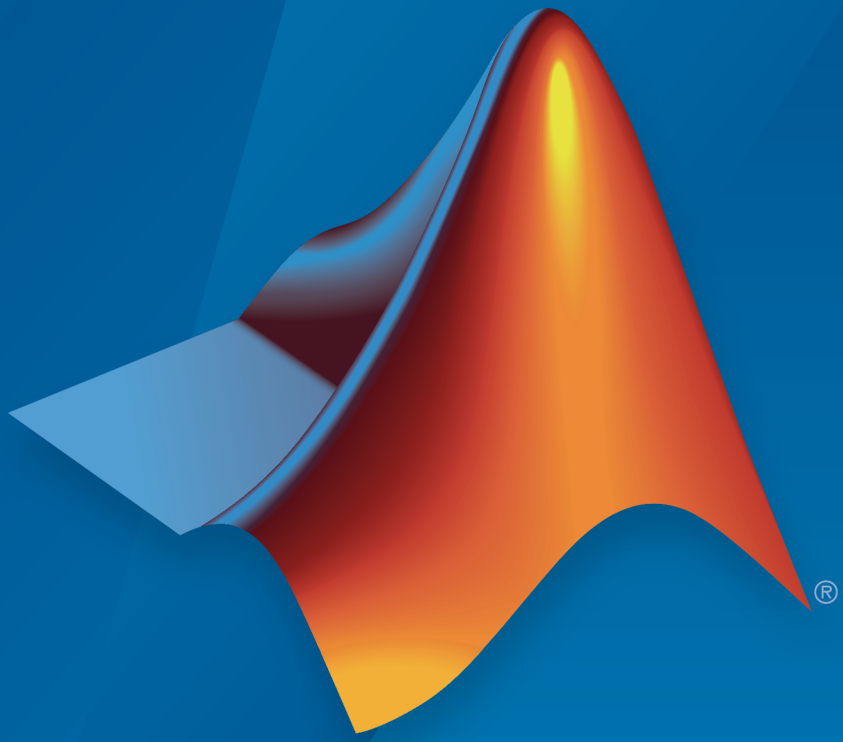
MATLAB®

Function Reference



MATLAB®

R2015a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

MATLAB Function Reference

Revision History

December 1996	First printing	For MATLAB 5.0 (Release 8)
June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online only	Revised for MATLAB 5.3 (Release 11)
June 1999	Second printing	For MATLAB 5.3 (Release 11)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for 6.5 (Release 13)
June 2004	Online only	Revised for 7.0 (Release 14)
September 2006	Online only	Revised for 7.3 (Release 2006b)
March 2007	Online only	Revised for 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)
March 2008	Online only	Revised for Version 7.6 (Release 2008a)
October 2008	Online only	Revised for Version 7.7 (Release 2008b)
March 2009	Online only	Revised for Version 7.8 (Release 2009a)
September 2009	Online only	Revised for Version 7.9 (Release 2009b)
March 2010	Online only	Revised for Version 7.10 (Release 2010a)
September 2010	Online only	Revised for Version 7.11 (Release 2010b)
April 2011	Online only	Revised for Version 7.12 (Release 2011a)
September 2011	Online only	Revised for Version 7.13 (Release 2011b)
March 2012	Online only	Revised for Version 7.14 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)

Contents

Alphabetical List

addframe (avifile)
addOptional
addParameter
addParamValue
addcats
addpath
addpref
addpoints
addprop (dynamicprops)
addproperty
addRequired
addsampletocollection
addtodate
addts
airy
align
alim
all
allchild
alpha
alphamap
alphaShape
alphaSpectrum
alphaTriangulation
area
boundaryFacets
criticalAlpha
inShape
nearestNeighbor
numRegions
perimeter
plot
surfaceArea
volume
amd
ancestor
and, &
angle
Using Animated Line Objects
animatedline

blkdiag
boundary
box
break
brighten
brush
bsxfun
builddocsearchdb
builtin
bvp4c
bvp5c
bvpget
bvpinit
bvpset
bvpxtend
caldays
caldiff
calendar
calendarDuration
calllib
callSoapService
calmonths
calquarters
calweeks
calyears
camdolly
cameratoolbar
camlight
camlookat
camorbit
campan
campos
camproj
camroll
camtarget
camup
camva
camzoom
cartToBary
cart2pol

cart2sph
cast
cat
categorical
categories
caxis
cd
convexHull
cd
cdf2rdf
cdfepoch
cdfinfo
cdflib
cdflib.close
cdflib.closeVar
cdflib.computeEpoch
cdflib.computeEpoch16
cdflib.create
cdflib.createAttr
cdflib.createVar
cdflib.delete
cdflib.deleteAttr
cdflib.deleteAttrEntry
cdflib.deleteAttrgEntry
cdflib.deleteVar
cdflib.deleteVarRecords
cdflib.epoch16Breakdown
cdflib.epochBreakdown
cdflib.getAttrEntry
cdflib.getAttrgEntry
cdflib.getAttrMaxEntry
cdflib.getAttrMaxgEntry
cdflib.getAttrName
cdflib.getAttrNum
cdflib.getAttrScope
cdflib.getCacheSize
cdflib.getChecksum
cdflib.getCompression
cdflib.getCompressionCacheSize
cdflib.getConstantNames

cdflib.getConstantValue
cdflib.getCopyright
cdflib.getFileBackward
cdflib.getFormat
cdflib.getLibraryCopyright
cdflib.getLibraryVersion
cdflib.getMajority
cdflib.getName
cdflib.getNumAttrEntries
cdflib.getNumAttrgEntries
cdflib.getNumAttributes
cdflib.getNumgAttributes
cdflib.getReadOnlyMode
cdflib.getStageCacheSize
cdflib.getValidate
cdflib.getVarAllocRecords
cdflib.getVarBlockingFactor
cdflib.getVarCacheSize
cdflib.getVarCompression
cdflib.getVarData
cdflib.getVarMaxAllocRecNum
cdflib.getVarMaxWrittenRecNum
cdflib.getVarsMaxWrittenRecNum
cdflib.getVarName
cdflib.getVarNum
cdflib.getVarNumRecsWritten
cdflib.getVarPadValue
cdflib.getVarRecordData
cdflib.getVarReservePercent
cdflib.getVarSparseRecords
cdflib.getVersion
cdflib.hyperGetVarData
cdflib.hyperPutVarData
cdflib.inquire
cdflib.inquireAttr
cdflib.inquireAttrEntry
cdflib.inquireAttrgEntry
cdflib.inquireVar
cdflib.open
cdflib.putAttrEntry

cdflib.putAttrgEntry
cdflib.putVarData
cdflib.putVarRecordData
cdflib.renameAttr
cdflib.renameVar
cdflib.setCacheSize
cdflib.setChecksum
cdflib.setCompression
cdflib.setCompressionCacheSize
cdflib.setFileBackward
cdflib.setFormat
cdflib.setMajority
cdflib.setReadOnlyMode
cdflib.setStageCacheSize
cdflib.setValidate
cdflib.setVarAllocBlockRecords
cdflib.setVarBlockingFactor
cdflib.setVarCacheSize
cdflib.setVarCompression
cdflib.setVarInitialRecs
cdflib.setVarPadValue
cdflib.SetVarReservePercent
cdflib.setVarsCacheSize
cdflib.setVarSparseRecords
cdfread
cdfwrite
ceil
cell
cell2mat
cell2struct
cell2table
celldisp
cellfun
cellplot
cellstr
cgs
char
checkcode
checkin
checkout

complex
computeStrip
computeTile
computer
cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contour Properties
contourslice
matlab.unittest.constraints
matlab.unittest.constraints.AbsoluteTolerance
matlab.unittest.constraints.AnyCellOf
matlab.unittest.constraints.AnyElementOf
matlab.unittest.constraints.BooleanConstraint
getNegativeDiagnosticFor
matlab.unittest.constraints.CellComparator
matlab.unittest.constraints.Constraint
getDiagnosticFor
satisfiedBy
matlab.unittest.constraints.ContainsSubstring
matlab.unittest.constraints.EndsWithSubstring
matlab.unittest.constraints.Eventually
matlab.unittest.constraints.EveryCellOf
matlab.unittest.constraints.EveryElementOf
matlab.unittest.constraints.HasElementCount
matlab.unittest.constraints.HasField
matlab.unittest.constraints.HasInf
matlab.unittest.constraints.HasLength
matlab.unittest.constraints.HasNaN
matlab.unittest.constraints.HasSize
matlab.unittest.constraints.IsAnything
matlab.unittest.constraints.IsEmpty
matlab.unittest.constraints.IsFalse

matlab.unittest.constraints.IsFinite
matlab.unittest.constraints.IsGreaterThan
matlab.unittest.constraints.IsGreaterThanOrEqualTo
matlab.unittest.constraints.IsEqualTo
matlab.unittest.constraints.IsInstanceOf
matlab.unittest.constraints.IsLessThan
matlab.unittest.constraints.IsLessThanOrEqualTo
matlab.unittest.constraints.IsOfClass
matlab.unittest.constraints.IsReal
matlab.unittest.constraints.IsSameHandleAs
matlab.unittest.constraints.IsScalar
matlab.unittest.constraints.IsSparse
matlab.unittest.constraints.IsSubstringOf
matlab.unittest.constraints.IssuesNoWarnings
matlab.unittest.constraints.IssuesWarnings
matlab.unittest.constraints.IsTrue
matlab.unittest.constraints.LogicalComparator
matlab.unittest.constraints.Matches
matlab.unittest.constraints.NumericComparator
matlab.unittest.constraints.ObjectComparator
matlab.unittest.constraints.PublicPropertyComparator
matlab.unittest.constraints.RelativeTolerance
matlab.unittest.constraints.ReturnsTrue
matlab.unittest.constraints.StartsWithSubstring
matlab.unittest.constraints.StringComparator
matlab.unittest.constraints.StructComparator
matlab.unittest.constraints.Throws
matlab.unittest.constraints.Tolerance
getDiagnosticFor
satisfiedBy
supports
contrast
conv
conv2
convhull
convhulln
convn
matlab.mixin.CustomDisplay
convertDimensionsToString
details

disp
display
displayEmptyObject
displayNonScalarObject
displayPropertyGroups
displayScalarHandleToDeletedObject
displayScalarObject
getClassNameForHeader
getDeletedHandleText
getDetailedFooter
getDetailedHeader
getFooter
getHandleText
getHeader
getPropertyGroups
getSimpleHeader
matlab.mixin.Copyable
copy
copyfile
copyobj
corrcoef
cos
cosd
cosh
cot
cotd
coth
countcats
cov
cplxpair
cputime
RandStream.create
createClassFromWsdl
createSoapMessage
cross
csc
cscd
csch
csvread
ctranspose, '

dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeget
ddensd
ddesd
ddeset
deal
deblank
dec2base
dec2bin
dec2hex
decic
deconv
del2
DelaunayTri
DelaunayTri
delaunay
delaunayn
delaunayTriangulation
convexHull
isInterior
voronoiDiagram
delete
delete (COM)
delete
delete (serial)
deleteproperty
delevent
delsamplefromcollection
demo
depdir
depfun
det
details
detrend

drawnow
dsearchn
duration
dynamicprops
echo
echodemo
edgeAttachments
edges
edit
eig
eigs
ellipj
ellipke
ellipsoid
empty
enableNETfromNetworkDrive
enableservice
end
EndInvoke
eomday
enumeration
eps
eq, ==
erf
erfc
erfcinv
erfcx
erfinv
error
errorbar
Errorbar Series Properties
errordlg
etime
etree
etreeplot
eval
evalc
evalin
event.EventData
event.listener

fft2
fftn
fftshift
fftw
fgetl
fgetl (serial)
fgets
fgets (serial)
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
filemarker
fileparts
fileread
filesep
fill
fill3
filter
filter2
find
findall
findfigs
findobj
findstr
finish
fitsdisp
fitsinfo
fitsread
fitswrite
fix
matlab.unittest.fixtures
matlab.unittest.fixtures.CurrentFolderFixture
matlab.unittest.fixtures.Fixture
log
setup
teardown
addTeardown

isCompatibile
matlab.unittest.fixtures.PathFixture
matlab.unittest.fixtures.SuppressedWarningsFixture
matlab.unittest.fixtures.TemporaryFolderFixture
flintmax
flip
flipdim
fliplr
flipud
floor
flow
fminbnd
fminsearch
fopen
fopen (serial)
for
format
fplot
fprintf
fprintf (serial)
frame2im
fread
fread (serial)
freeBoundary
freqspace
frewind
fscanf
fscanf (serial)
fseek
ftell
FTP
full
fullfile
func2str
function
function_handle (@)
functions
functiontests
funm
fwrite

h5readatt
h5write
h5writeatt
H5.close
H5.garbage_collect
H5.get_libversion
H5.open
H5.set_free_list_limits
H5A.close
H5A.create
H5A.delete
H5A.get_info
H5A.get_name
H5A.get_space
H5A.get_type
H5A.iterate
H5A.open
H5A.open_by_idx
H5A.open_by_name
H5A.read
H5A.write
H5D.close
H5D.create
H5D.get_access_plist
H5D.get_create_plist
H5D.get_offset
H5D.get_space
H5D.get_space_status
H5D.get_storage_size
H5D.get_type
H5D.open
H5D.read
H5D.set_extent
H5D.vlen_get_buf_size
H5D.write
H5DS.attach_scale
H5DS.detach_scale
H5DS.get_label
H5DS.get_num_scales
H5DS.get_scale_name

H5DS.is_scale
H5DS.iterate_scales
H5DS.set_label
H5DS.set_scale
H5E.clear
H5E.get_major
H5E.get_minor
H5E.walk
H5F.close
H5F.create
H5F.flush
H5F.get_access_plist
H5F.get_create_plist
H5F.get_filesize
H5F.get_freespace
H5F.get_info
H5F.get_mdc_config
H5F.get_mdc_hit_rate
H5F.get_mdc_size
H5F.get_name
H5F.get_obj_count
H5F.get_obj_ids
H5F.is_hdf5
H5F.mount
H5F.open
H5F.reopen
H5F.set_mdc_config
H5F.unmount
H5G.close
H5G.create
H5G.get_info
H5G.open
H5I.dec_ref
H5I.get_file_id
H5I.get_name
H5I.get_ref
H5I.get_type
H5I.inc_ref
H5I.is_valid
H5L.copy

H5L.create_external
H5L.create_hard
H5L.create_soft
H5L.delete
H5L.exists
H5L.get_info
H5L.get_name_by_idx
H5L.get_val
H5L.iterate
H5L.iterate_by_name
H5L.move
H5L.visit
H5L.visit_by_name
H5ML.compare_values
H5ML.get_constant_names
H5ML.get_constant_value
H5ML.get_function_names
H5ML.get_mem_datatype
H5ML.hoffset
H5ML.sizeof
H5O.close
H5O.copy
H5O.get_comment
H5O.get_comment_by_name
H5O.get_info
H5O.link
H5O.open
H5O.open_by_idx
H5O.set_comment
H5O.set_comment_by_name
H5O.visit
H5O.visit_by_name
H5P.close
H5P.copy
H5P.create
H5P.get_class
H5P.close_class
H5P.equal
H5P.exist
H5P.get

H5P.get_class_name
H5P.get_class_parent
H5P.get_nprops
H5P.get_size
H5P.isa_class
H5P.iterate
H5P.set
H5P.get_btree_ratios
H5P.get_chunk_cache
H5P.get_dxpl_multi
H5P.get_edc_check
H5P.get_hyper_vector_size
H5P.set_btree_ratios
H5P.set_chunk_cache
H5P.set_dxpl_multi
H5P.set_edc_check
H5P.set_hyper_vector_size
H5P.all_filters_avail
H5P.fill_value_defined
H5P.get_alloc_time
H5P.get_chunk
H5P.get_external
H5P.get_external_count
H5P.get_fill_time
H5P.get_fill_value
H5P.get_filter
H5P.get_filter_by_id
H5P.get_layout
H5P.get_nfilters
H5P.modify_filter
H5P.remove_filter
H5P.set_alloc_time
H5P.set_chunk
H5P.set_deflate
H5P.set_external
H5P.set_fill_time
H5P.set_fill_value
H5P.set_filter
H5P.set_fletcher32
H5P.set_layout

H5P.set_nbit
H5P.set_scaleoffset
H5P.set_shuffle
H5P.get_alignment
H5P.get_driver
H5P.get_family_offset
H5P.get_fapl_core
H5P.get_fapl_family
H5P.get_fapl_multi
H5P.get_fclose_degree
H5P.get_libver_bounds
H5P.get_gc_references
H5P.get_mdc_config
H5P.get_meta_block_size
H5P.get_multi_type
H5P.get_sieve_buf_size
H5P.get_small_data_block_size
H5P.set_alignment
H5P.set_family_offset
H5P.set_fapl_core
H5P.set_fapl_family
H5P.set_fapl_log
H5P.set_fapl_multi
H5P.set_fapl_sec2
H5P.set_fapl_split
H5P.set_fapl_stdio
H5P.set_fclose_degree
H5P.set_gc_references
H5P.set_libver_bounds
H5P.set_mdc_config
H5P.set_meta_block_size
H5P.set_multi_type
H5P.set_sieve_buf_size
H5P.set_small_data_block_size
H5P.get_istore_k
H5P.get_sizes
H5P.get_sym_k
H5P.get_userblock
H5P.get_version
H5P.set_istore_k

H5P.set_sizes
H5P.set_sym_k
H5P.set_userblock
H5P.get_attr_creation_order
H5P.get_attr_phase_change
H5P.get_copy_object
H5P.set_attr_creation_order
H5P.set_attr_phase_change
H5P.set_copy_object
H5P.get_create_intermediate_group
H5P.get_link_creation_order
H5P.get_link_phase_change
H5P.set_create_intermediate_group
H5P.set_link_creation_order
H5P.set_link_phase_change
H5P.get_char_encoding
H5P.set_char_encoding
H5R.create
H5R.dereference
H5R.get_name
H5R.get_obj_type
H5R.get_region
H5S.copy
H5S.create
H5S.close
H5S.create_simple
H5S.extent_copy
H5S.get_select_bounds
H5S.get_select_elem_npoints
H5S.get_select_elem_pointlist
H5S.get_select_hyper_blocklist
H5S.get_select_hyper_nblocks
H5S.get_select_npoints
H5S.get_select_type
H5S.get_simple_extent_dims
H5S.get_simple_extent_ndims
H5S.get_simple_extent_npoints
H5S.get_simple_extent_type
H5S.is_simple
H5S.offset_simple

H5S.select_all
H5S.select_elements
H5S.select_hyperslab
H5S.select_none
H5S.select_valid
H5S.set_extent_none
H5S.set_extent_simple
H5T.close
H5T.commit
H5T.committed
H5T.copy
H5T.create
H5T.detect_class
H5T.equal
H5T.get_class
H5T.get_create_plist
H5T.get_native_type
H5T.get_size
H5T.get_super
H5T.lock
H5T.open
H5T.array_create
H5T.get_array_dims
H5T.get_array_ndims
H5T.get_cset
H5T.get_ebias
H5T.get_fields
H5T.get_inpad
H5T.get_norm
H5T.get_offset
H5T.get_order
H5T.get_pad
H5T.get_precision
H5T.get_sign
H5T.get_strpad
H5T.set_cset
H5T.set_ebias
H5T.set_fields
H5T.set_inpad
H5T.set_norm

H5T.set_offset
H5T.set_order
H5T.set_pad
H5T.set_precision
H5T.set_sign
H5T.set_size
H5T.set_strpad
H5T.get_member_class
H5T.get_member_index
H5T.get_member_name
H5T.get_member_offset
H5T.get_member_type
H5T.get_nmembers
H5T.insert
H5T.pack
H5T.enum_create
H5T.enum_insert
H5T.enum_nameof
H5T.enum_valueof
H5T.get_member_value
H5T.get_tag
H5T.set_tag
H5T.is_variable_str
H5T.vlen_create
H5Z.filter_avail
H5Z.get_filter_info
hadamard
handle
addlistener
delete
findobj
findprop
isvalid
notify
relationaloperators
hankel
hasnext
hasFrame
hdf5info
hdf5read

hdf5write
hdfan
hdfdf24
hdfdfr8
hdfh
hdfhd
hdfhe
hdfhx
hdfinfo
hdfml
hdfpt
hdfread
hdftool
hdfv
hdfvf
hdfvh
hdfvs
height
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
matlab.mixin.Heterogeneous
cat
getDefaultScalarElement
horzcat
vertcat
hex2dec
hex2num
hgexport
hggroup
Group Properties
hgload
hgsave
hgsetget
hgtransform
Transform Properties
hidden

incenters
inOutStatus
ind2rgb
ind2sub
Inf
inferiorto
info
inline
inmem
innerjoin
inpolygon
input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8
int16
int32
int64
integral
integral2
integral3
interfaces
interp1
interp1q
interp2
interp3
interpft
interpn
interpstreamspeed
intersect
intmax
intmin
inv
invhilb

invoke
ipermute
is*
isa
isappdata
isbanded
isbetween
iscalendarduration
iscategorical
iscategory
iscell
iscellstr
ischar
iscolumn
iscom
isdatetime
isdiag
isdir
isdst
isduration
isEdge
isempty
isempty (tscollection)
isequal
isenum
isequaln
isequalwithequalnans
isevent
isfield
isfinite
isfloat
isglobal
isgraphics
ishandle
ishermitian
ishghandle
ishold
isinf
isinteger
isinterface

isjava
isKey
iskeyword
isletter
islogical
ismac
ismatrix
ismember
ismembertol
ismethod
ismethod (COM)
ismissing
isnan
isnat
isnumeric
isobject
isocaps
isocolors
isonormals
isordinal
isosurface
ispc
ispref
isprime
isprop
isprop (COM)
isprotected
isreal
isrow
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
issymmetric
isTiled
istable

length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
lib.pointer
disp
isNull
plus
reshape
setdatatype
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
LineSpec (Line Specification)
linkaxes
linkdata
linkprop
linsolve
linspace
RandStream.list
listdlg
listfonts
load
load (COM)
load (serial)
loadlibrary
loadobj
localfunctions
log
log10
log1p
log2
logical
loglog

logm
logspace
lookfor
lower
ls
lscov
lsqnonneg
lsqr
lt, <
lu
Primitive Line Properties
magic
makehgtform
mapreduce
mapreducer
containers.Map
mat2cell
mat2str
material
matfile
matlab.codetools.requiredFilesAndProducts
matlab.io.MatFile
matlab.io.fits.closeFile
matlab.io.fits.createFile
matlab.io.fits.deleteFile
matlab.io.fits.fileName
matlab.io.fits.fileMode
matlab.io.fits.openFile
matlab.io.fits.createImg
matlab.io.fits.getImgSize
matlab.io.fits.getImgType
matlab.io.fits.insertImg
matlab.io.fits.readImg
matlab.io.fits.setBscale
matlab.io.fits.writeImg
matlab.io.fits.deleteKey
matlab.io.fits.deleteRecord
matlab.io.fits.getHdrSpace
matlab.io.fits.readCard
matlab.io.fits.readKey

matlab.io.fits.readKeyCmplx
matlab.io.fits.readKeyDbl
matlab.io.fits.readKeyLongLong
matlab.io.fits.readKeyLongStr
matlab.io.fits.readKeyUnit
matlab.io.fits.readRecord
matlab.io.fits.writeComment
matlab.io.fits.writeDate
matlab.io.fits.writeKey
matlab.io.fits.writeKeyUnit
matlab.io.fits.writeHistory
matlab.io.fits.copyHDU
matlab.io.fits.deleteHDU
matlab.io.fits.getHDUnum
matlab.io.fits.getHDUtype
matlab.io.fits.getNumHDUs
matlab.io.fits.movAbsHDU
matlab.io.fits.movNamHDU
matlab.io.fits.movRelHDU
matlab.io.fits.writeChecksum
matlab.io.fits.imgCompress
matlab.io.fits.isCompressedImg
matlab.io.fits.setCompressionType
matlab.io.fits.setHCompScale
matlab.io.fits.setHCompSmooth
matlab.io.fits.setTileDim
matlab.io.fits.createTbl
matlab.io.fits.deleteCol
matlab.io.fits.deleteRows
matlab.io.fits.insertRows
matlab.io.fits.getAColParms
matlab.io.fits.getBColParms
matlab.io.fits.getColName
matlab.io.fits.getColType
matlab.io.fits.getEqColType
matlab.io.fits.getNumCols
matlab.io.fits.getNumRows
matlab.io.fits.insertCol
matlab.io.fits.insertATbl
matlab.io.fits.insertBTbl

matlab.io.fits.readATblHdr
matlab.io.fits.readBTblHdr
matlab.io.fits.readCol
matlab.io.fits.setTscale
matlab.io.fits.writeCol
matlab.io.fits.getConstantValue
matlab.io.fits.getVersion
matlab.io.fits.getOpenFiles
matlab.io.hdf4.sd
matlab.io.hdf4.sd.attrInfo
matlab.io.hdf4.sd.close
matlab.io.hdf4.sd.create
matlab.io.hdf4.sd.dimInfo
matlab.io.hdf4.sd.endAccess
matlab.io.hdf4.sd.fileInfo
matlab.io.hdf4.sd.findAttr
matlab.io.hdf4.sd.getCal
matlab.io.hdf4.sd.getChunkInfo
matlab.io.hdf4.sd.getCompInfo
matlab.io.hdf4.sd.getDataStrs
matlab.io.hdf4.sd.getDimID
matlab.io.hdf4.sd.getDimScale
matlab.io.hdf4.sd.getDimStrs
matlab.io.hdf4.sd.getFilename
matlab.io.hdf4.sd.getFillValue
matlab.io.hdf4.sd.getInfo
matlab.io.hdf4.sd.getRange
matlab.io.hdf4.sd.idToRef
matlab.io.hdf4.sd.idType
matlab.io.hdf4.sd.isCoordVar
matlab.io.hdf4.sd.isRecord
matlab.io.hdf4.sd.nameToIndex
matlab.io.hdf4.sd.nameToIndices
matlab.io.hdf4.sd.readAttr
matlab.io.hdf4.sd.readChunk
matlab.io.hdf4.sd.readData
matlab.io.hdf4.sd.refToIndex
matlab.io.hdf4.sd.select
matlab.io.hdf4.sd.setAttr
matlab.io.hdf4.sd.setCal

matlab.io.hdf4.sd.setChunk
matlab.io.hdf4.sd.setCompress
matlab.io.hdf4.sd.setDataStrs
matlab.io.hdf4.sd.setDimName
matlab.io.hdf4.sd.setDimScale
matlab.io.hdf4.sd.setDimStrs
matlab.io.hdf4.sd.setExternalFile
matlab.io.hdf4.sd.setFillMode
matlab.io.hdf4.sd.setFillValue
matlab.io.hdf4.sd.setNBitDataSet
matlab.io.hdf4.sd.setRange
matlab.io.hdf4.sd.start
matlab.io.hdf4.sd.writeChunk
matlab.io.hdf4.sd.writeData
matlab.io.hdfeos.gd
matlab.io.hdfeos.gd.attach
matlab.io.hdfeos.gd.close
matlab.io.hdfeos.gd.compInfo
matlab.io.hdfeos.gd.create
matlab.io.hdfeos.gd.defBoxRegion
matlab.io.hdfeos.gd.defComp
matlab.io.hdfeos.gd.defDim
matlab.io.hdfeos.gd.defField
matlab.io.hdfeos.gd.defOrigin
matlab.io.hdfeos.gd.defPixReg
matlab.io.hdfeos.gd.defProj
matlab.io.hdfeos.gd.defTile
matlab.io.hdfeos.gd.defVrtRegion
matlab.io.hdfeos.gd.detach
matlab.io.hdfeos.gd.dimInfo
matlab.io.hdfeos.gd.extractRegion
matlab.io.hdfeos.gd.fieldInfo
matlab.io.hdfeos.gd.getFillValue
matlab.io.hdfeos.gd.getPixels
matlab.io.hdfeos.gd.getPixValues
matlab.io.hdfeos.gd.gridInfo
matlab.io.hdfeos.gd.ij2ll
matlab.io.hdfeos.gd.inqAttrs
matlab.io.hdfeos.gd.inqDims
matlab.io.hdfeos.gd.inqFields

matlab.io.hdfeos.gd.inqGrid
matlab.io.hdfeos.gd.ll2ij
matlab.io.hdfeos.gd.nEntries
matlab.io.hdfeos.gd.open
matlab.io.hdfeos.gd.originInfo
matlab.io.hdfeos.gd.pixRegInfo
matlab.io.hdfeos.gd.projInfo
matlab.io.hdfeos.gd.readAttr
matlab.io.hdfeos.gd.readBlkSomOffset
matlab.io.hdfeos.gd.readField
matlab.io.hdfeos.gd.readTile
matlab.io.hdfeos.gd.regionInfo
matlab.io.hdfeos.gd.setFillValue
matlab.io.hdfeos.gd.setTileComp
matlab.io.hdfeos.gd.sphereCodeToName
matlab.io.hdfeos.gd.sphereNameToCode
matlab.io.hdfeos.gd.tileInfo
matlab.io.hdfeos.gd.writeAttr
matlab.io.hdfeos.gd.writeBlkSomOffset
matlab.io.hdfeos.gd.writeField
matlab.io.hdfeos.gd.writeTile
matlab.io.hdfeos.sw
matlab.io.hdfeos.sw.attach
matlab.io.hdfeos.sw.close
matlab.io.hdfeos.sw.compInfo
matlab.io.hdfeos.sw.create
matlab.io.hdfeos.sw.defBoxRegion
matlab.io.hdfeos.sw.defComp
matlab.io.hdfeos.sw.defDataField
matlab.io.hdfeos.sw.defDim
matlab.io.hdfeos.sw.defDimMap
matlab.io.hdfeos.sw.defGeoField
matlab.io.hdfeos.sw.defTimePeriod
matlab.io.hdfeos.sw.defVrtRegion
matlab.io.hdfeos.sw.detach
matlab.io.hdfeos.sw.dimInfo
matlab.io.hdfeos.sw.extractPeriod
matlab.io.hdfeos.sw.extractRegion
matlab.io.hdfeos.sw.fieldInfo
matlab.io.hdfeos.sw.geoMapInfo

matlab.io.hdfeos.sw.getFillValue
matlab.io.hdfeos.sw.idxMapInfo
matlab.io.hdfeos.sw.inqAttrs
matlab.io.hdfeos.sw.inqDataFields
matlab.io.hdfeos.sw.inqDims
matlab.io.hdfeos.sw.inqGeoFields
matlab.io.hdfeos.sw.inqIdxMaps
matlab.io.hdfeos.sw.inqMaps
matlab.io.hdfeos.sw.inqSwath
matlab.io.hdfeos.sw.mapInfo
matlab.io.hdfeos.sw.nEntries
matlab.io.hdfeos.sw.open
matlab.io.hdfeos.sw.periodInfo
matlab.io.hdfeos.sw.readAttr
matlab.io.hdfeos.sw.readField
matlab.io.hdfeos.sw.regionInfo
matlab.io.hdfeos.sw.setFillValue
matlab.io.hdfeos.sw.writeAttr
matlab.io.hdfeos.sw.writeField
matlab.io.saveVariablesToScript
matlab.lang.makeUniqueStrings
matlab.lang.makeValidName
matlabrc
matlabroot
matlabshared.supportpkg.checkForUpdate
matlabshared.supportpkg.getInstalled
matlab (Linux)
matlab (Mac)
matlab (Windows)
max
MaximizeCommandWindow
maxNumCompThreads
mean
median
memmapfile
memory
menu
mergecats
mesh
meshc

meshz
meshgrid
meta.class
meta.class.fromName
meta.DynamicProperty
meta.EnumeratedValue
meta.event
meta.MetaData
meta.method
meta.package
meta.abstractDetails
meta.package.fromName
meta.package.getAllPackages
meta.property
metaclass
methods
methodsview
mex
mex.getCompilerConfigurations
MException
addCause
getReport
last
rethrow
throw
throwAsCaller
mexext
mfilename
mget
milliseconds
min
MinimizeCommandWindow
minres
minus, -
minute
minutes
mislocked
mkdir
mkdir
mkpp

mldivide, \
mrdivide, /
mlint
mlintrpt
mlock
mmfileinfo
mmreader
mod
mode
month
more
morebins
move
movefile
movegui
movie
movie2avi
mpower, ^
mput
msgbox
mtimes, *
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin
narginchk
nargout
nargoutchk
native2unicode
nchoosek
ndgrid
ndims
ne, ~=
nearestNeighbor
neighbors
NET

NET.addAssembly
NET.Assembly
NET.convertArray
NET.createArray
NET.createGeneric
NET.disableAutoRelease
NET.enableAutoRelease
NET.GenericClass
NET.invokeGenericMethod
NET.isNETSupported
NET.NetException
NET.setStaticProperty
nccreate
ncdisp
ncinfo
ncread
ncreadatt
ncwrite
ncwriteatt
ncwriteschema
netcdf
netcdf.abort
netcdf.close
netcdf.copyAtt
netcdf.create
netcdf.defDim
netcdf.defGrp
netcdf.defVar
netcdf.defVarChunking
netcdf.defVarDeflate
netcdf.defVarFill
netcdf.defVarFletcher32
netcdf.delAtt
netcdf.endDef
netcdf.getAtt
netcdf.getChunkCache
netcdf.getConstant
netcdf.getConstantNames
netcdf.getVar
netcdf.inq

netcdf.inqDimIDs
netcdf.inqFormat
netcdf.inqGrpName
netcdf.inqGrpNameFull
netcdf.inqGrpParent
netcdf.inqGrps
netcdf.inqNcid
netcdf.inqUnlimDims
netcdf.inqVarIDs
netcdf.inqVarChunking
netcdf.inqVarDeflate
netcdf.inqVarFill
netcdf.inqVarFletcher32
netcdf.inqAtt
netcdf.inqAttID
netcdf.inqAttName
netcdf.inqDim
netcdf.inqDimID
netcdf.inqLibVers
netcdf.inqVar
netcdf.inqVarID
netcdf.open
netcdf.putAtt
netcdf.putVar
netcdf.reDef
netcdf.renameAtt
netcdf.renameDim
netcdf.renameVar
netcdf.setChunkCache
netcdf.setDefaultFormat
netcdf.setFill
netcdf.sync
newplot
nextDirectory
nextpow2
nnz
noanimate
nonzeros
norm
normest

not, ~
notebook
now
nthroot
null
num2cell
num2hex
num2str
numberOfStrips
numberOfTiles
numel
nzmax
matlab.lang.ObjectUpdateFailure
ode15i
ode15s
ode23
ode23s
ode23t
ode23tb
ode45
ode113
odeget
odeset
odextend
onCleanup
ones
open
open
openfig
opengl
openvar
optimget
optimset
or, |
ordeig
orderfields
ordqz
ordschur
orient
orth

outerjoin
pack
padecoef
pagesetupdlg
pan
matlab.unittest.parameters
matlab.unittest.parameters.EmptyParameter
matlab.unittest.parameters.ClassSetupParameter
matlab.unittest.parameters.TestParameter
matlab.unittest.parameters.MethodSetupParameter
pareto
parfor
parse
parseSoapResponse
pascal
patch
Patch Properties
path
path2rc
pathsep
pathtool
pause
pbaspect
pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie
pie3
pinv
planerot
play

play
playblocking
plot
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
matlab.unittest.plugins
matlab.unittest.plugins.CodeCoveragePlugin
matlab.unittest.plugins.DiagnosticsValidationPlugin
matlab.unittest.plugins.FailureDiagnosticsPlugin
matlab.unittest.plugins.LoggingPlugin
withVerbosity
matlab.unittest.plugins.OutputStream
print
matlab.unittest.plugins.StopOnFailuresPlugin
matlab.unittest.plugins.TAPPlugin
matlab.unittest.plugins.ToFile
matlab.unittest.plugins.ToStandardOutput
matlab.unittest.plugins.TestRunnerPlugin
runTestSuite
createSharedTestFixture
setupSharedTestFixture
runTestClass
createTestClassInstance
setupTestClass
runTest
createTestMethodInstance
setupTestMethod
runTestMethod
teardownTestMethod
teardownTestClass
teardownSharedTestFixture
matlab.unittest.plugins.TestRunProgressPlugin
matlab.unittest.plugins.plugindata
matlab.unittest.plugins.plugindata.ImplicitFixturePluginData
matlab.unittest.plugins.plugindata.PluginData
matlab.unittest.plugins.plugindata.SharedTestFixturePluginData

matlab.unittest.plugins.plugindata.TestSuiteRunPluginData
plus, +
pointLocation
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval
polyvalm
posixtime
pow2
power, .^
ppval
prefdir
preferences
primes
print
printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
properties
propertyeditor
matlab.mixin.util.PropertyGroup
psi
publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
pyargs
matlab.exception.PyException

pyversion
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quad2d
quadgk
quadl
quadv
matlab.unittest.qualifications
matlab.unittest.qualifications.Assertable
assertClass
assertEmpty
assertEqual
assertError
assertFail
assertFalse
assertGreaterThan
assertGreaterThanOrEqual
assertInstanceOf
assertLength
assertLessThan
assertLessThanOrEqual
assertMatches
assertNotEmpty
assertNotEqual
assertNotSameHandle
assertNumElements
assertReturnsTrue
assertSameHandle
assertSize
assertSubstring
assertThat
assertTrue
assertWarning
assertWarningFree
matlab.unittest.qualifications.AssertionFailedException
matlab.unittest.qualifications.Assumable

assumeClass
assumeEmpty
assumeEqual
assumeError
assumeFail
assumeFalse
assumeGreaterThan
assumeGreaterThanOrEqual
assumeInstanceOf
assumeLength
assumeLessThan
assumeLessThanOrEqual
assumeMatches
assumeNotEmpty
assumeNotEqual
assumeNotSameHandle
assumeNumElements
assumeReturnsTrue
assumeSameHandle
assumeSize
assumeSubstring
assumeThat
assumeTrue
assumeWarning
assumeWarningFree
matlab.unittest.qualifications.AssumptionFailedException
matlab.unittest.qualifications.ExceptionEventData
matlab.unittest.qualifications.FatalAssertable
fatalAssertClass
fatalAssertEmpty
fatalAssertEqual
fatalAssertError
fatalAssertFail
fatalAssertFalse
fatalAssertGreaterThan
fatalAssertGreaterThanOrEqual
fatalAssertInstanceOf
fatalAssertLength
fatalAssertLessThan
fatalAssertLessThanOrEqual

fatalAssertMatches
fatalAssertNotEmpty
fatalAssertNotEqual
fatalAssertNotSameHandle
fatalAssertNumElements
fatalAssertReturnsTrue
fatalAssertSameHandle
fatalAssertSize
fatalAssertSubstring
fatalAssertThat
fatalAssertTrue
fatalAssertWarning
fatalAssertWarningFree
matlab.unittest.qualifications.FatalAssertionFailedException
matlab.unittest.qualifications.QualificationEventData
matlab.unittest.qualifications.Verifiable
verifyClass
verifyEmpty
verifyEqual
verifyError
verifyFail
verifyFalse
verifyGreaterThan
verifyGreaterThanOrEqual
verifyInstanceOf
verifyLength
verifyLessThan
verifyLessThanOrEqual
verifyMatches
verifyNotEmpty
verifyNotEqual
verifyNotSameHandle
verifyNumElements
verifyReturnsTrue
verifySameHandle
verifySize
verifySubstring
verifyThat
verifyTrue
verifyWarning

verifyWarningFree
quarter
questdlg
quit
Quit (COM)
quiver
quiver3
Quiver Series Properties
qz
rand
rand (RandStream)
randi
randi (RandStream)
randn
randn (RandStream)
randperm
randperm (RandStream)
RandStream
RandStream constructor
RandStream.getGlobalStream
RandStream.setGlobalStream
rank
rat
rats
rbbox
rcond
rdivide, ./
read
readFrame
read
read
readasync
readEncodedStrip
readEncodedTile
readRGBAImage
readRGBAStrip
readRGBATile
Remove
RemoveAll
timeseries

addsample
append
ctranspose
delsample
detrend
filter
get
getabstime
getdatasamples
getdatasamplesize
getinterpmethod
getqualitydesc
getsamples
getsampleusingtime
idealfilter
iqr
max
mean
median
min
plot
resample
set
setabstime
setinterpmethod
setuniformtime
synchronize
transpose
std
sum
var
triangulation
barycentricToCartesian
cartesianToBarycentric
circumcenter
edgeAttachments
edges
faceNormal
featureEdges
freeBoundary

incenter
isConnected
nearestNeighbor
neighbors
pointLocation
size
vertexAttachments
vertexNormal
readtable
real
reallog
realmax
realmin
realpow
realsqrt
record
record
recordblocking
rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp
regexpi
regexprep
regexptranslate
registerevent
rehash
release
rem
remove
removecats
removets
renamecats
reordercats
rename

repelem
repmat
resample (tscollection)
reset
reset (RandStream)
reshape
residue
restoredefaultpath
rethrow
return
rewriteDirectory
rgb2gray
rgb2hsv
rgb2ind
rgbplot
ribbon
rmappdata
rmdir
rmdir
rmfield
rmpath
rmpref
rng
Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rowfun
rref
rsf2csf
run
runtests
Chart Surface Properties
Primitive Surface Properties
save
save (COM)

save (serial)
saveas
savefig
saveobj
savepath
scatter
scatter3
Scatter Series Properties
schur
script
scatteredInterpolant
sec
secd
sech
second
seconds
selectmoveresize
matlab.unittest.selectors
matlab.unittest.selectors.HasParameter
matlab.unittest.selectors.HasSharedTestFixture
matlab.unittest.selectors.HasBaseFolder
matlab.unittest.selectors.HasName
matlab.unittest.selectors.HasTag
semilogx
semilogy
sendmail
serial
serialbreak
set
set
set
set (COM)
matlab.mixin.SetGet
set
setdisp
get
getdisp
set
set (RandStream)
set (serial)

set (tscollection)
setabstime (tscollection)
setappdata
setcats
setdiff
setDirectory
setenv
setfield
setpixelposition
setpref
setstr
setSubDirectory
setTag
settimeseriesnames
setxor
shading
shg
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size
size
size (serial)
size
size (tscollection)
slice
smooth3
snapnow
sort
sortrows
sound
soundsc
spalloc
sparse

spaugment
spconvert
spdiags
specular
speye
spfun
sph2cart
sphere
spinmap
spline
split
spones
spparms
sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stack
stairs
Stair Properties
standardizeMissing
startup
std
stem
stem3
Stem Series Properties
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp

strcmpi
stream2
stream3
streamline
streamparticles
streamribbon
streamslice
streamtube
strfind
strings
strjoin
strjust
strmatch
strncmp
strncmpi
strread
strrep
strsplit
strtok
strtrim
struct
struct2cell
struct2table
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substruct
subvolume
sum
summary
superclasses
superiorto
support
Support Package Installer
supportPackageInstaller

surf
surfc
surf2patch
surface
surfl
surfnorm
svd
svds
swapbytes
switch, case, otherwise
sylvester
symamd
symbfact
symmlq
symrcm
symvar
syntax
system
table
table2array
table2cell
table2struct
Table Properties
TabularTextDatastore Properties
tan
tand
tanh
tar
targetupdater
tcpclient
tempdir
tempname
tetramesh
matlab.unittest.Test
matlab.unittest.TestCase
addTeardown
applyFixture
forInteractiveUse
getSharedTestFixtures
log

run
matlab.unittest.TestResult
matlab.unittest.TestRunner
addPlugin
run
runInParallel
withNoPlugins
withTextOutput
matlab.unittest.TestSuite
fromClass
fromFile
fromFolder
fromMethod
fromName
fromPackage
run
selectIf
texlabel
text
Text Properties
textread
textscan
textwrap
tfqmr
tic
timeit
toc
Tiff
time
timeofday
timer
delete
get
isvalid
set
start
startat
stop
timerfind
timerfindall

wait
times, .*
title
todatenum
toeplitz
toolboxdir
trace
transpose, .'
trapz
treelayout
treeplot
tril
trimesh
triplequad
triplot
TriRep
TriRep
TriScatteredInterp
TriScatteredInterp
trisurf
triu
true
try, catch
tscollection
tsdata.event
tsearchn
type
typecast
tzoffset
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties
uigetdir
uigetfile
uigetpref
uiimport
uimenu

unregisterevent
unstack
untar
unwrap
unzip
uplus, +
upper
urlread
urlwrite
usejava
userpath
Using ValueIterator Objects
validateattributes
validatestring
values
vander
var
varargin
varargout
varfun
vectorize
ver
matlab.unittest.Verbosity
verctrl
verLessThan
version
vertcat
vertcat (tscollection)
vertexAttachments
VideoReader
VideoWriter
view
viewmtx
visdiff
volumebounds
voronoi
voronoiDiagram
voronoin
waitbar
waitfor

waitforbuttonpress
warndlg
warning
waterfall
wavfinfo
wavplay
wavread
wavrecord
wavwrite
web
weboptions
webread
websave
webwrite
week
weekday
what
whatsnew
which
while
whitebg
who
who
whos
whos
width
wilkinson
winopen
winqueryreg
workspace
write
writetable
write
writeDirectory
writeEncodedStrip
writeEncodedTile
writeVideo
xlabel
matlab.wsdl.createWSDLClient
matlab.wsdl.setWSDLToolPath

year
years
ylabel
ymd
yyyymmdd
zlabel
xlim
ylim
zlim
xlsfinfo
xlsread
xlswrite
xmlread
xmlwrite
xor
xslt
zeros
zip
zoom

Relational Operators < > <= >= == ~=

Relational operations

Syntax

```
A < B

A > B

A <= B

A >= B

A == B

A ~= B
```

Description

The relational operators are <, >, <=, >=, ==, and ~=. Relational operators perform element-by-element comparisons between two arrays. They return a `logical` array of the same size, with elements set to logical `1` (`true`) where the relation is true, and elements set to logical `0` (`false`) where it is not.

The operators <, >, <=, and >= use only the real part of their operands for the comparison. The operators == and ~= test real and imaginary parts.

To test if two strings are equivalent, use `strcmp`, which allows vectors of dissimilar length to be compared.

Note For some toolboxes, the relational operators are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given operator, type `help` followed by the operator name. For example, type `help lt`. The toolboxes that overload `lt (<)` are listed. For information about using the operator in that toolbox, see the documentation for the toolbox.

Examples

If one of the operands is a scalar and the other a matrix, the scalar expands to the size of the matrix. For example, the two pairs of statements

```
X = 5; X >= [1 2 3; 4 5 6; 7 8 10]
X = 5*ones(3,3); X >= [1 2 3; 4 5 6; 7 8 10]
```

produce the same result:

```
ans =

     1     1     1
     1     1     0
     0     0     0
```

See Also

`all` | `any` | `find` | `Logical Operators: Short Circuit` | `strcmp`

Logical Operators: Short-Circuit && ||

Logical operations with short-circuiting

Syntax

```
expr1 && expr2
expr1 || expr2
```

Description

`expr1 && expr2` represents a logical `AND` operation that employs short-circuiting behavior. That is, `expr2` is not evaluated if `expr1` is logical `0` (`false`). Each expression must evaluate to a scalar logical result.

`expr1 || expr2` represents a logical `OR` operation that employs short-circuiting behavior. That is, `expr2` is not evaluated if `expr1` is logical `1` (`true`). Each expression must evaluate to a scalar logical result.

Examples

Use Scalar Logical Conditions

Create two vectors.

```
X = [1 0 0 1 1];
Y = [0 0 0 0 0];
```

Use the short-circuit OR operator with `X` and `Y`.

```
X || Y
```

```
Operands to the || and && operators must be convertible to logical scalar values.
```

The short-circuit operators operate only with scalar logical conditions.

Use the `any` and `all` functions to reduce each vector to a single logical condition.

```
any(X) || any(Y)

ans =

     1
```

The expression is equivalent to `1 || 0`, so it evaluates to logical `1` (`true`) after computing only the first condition, `any(X)`.

Specify Dependent Logical Conditions

Specify a logical statement where the second condition depends on the first. In the following statement, it doesn't make sense to evaluate the relation on the right if the divisor, `b`, is zero.

```
b = 1;
a = 20;
x = (b ~= 0) && (a/b > 18.5)

x =

     1
```

The result is logical `1` (`true`). However, if `(b ~= 0)` evaluates to `false`, MATLAB® assumes the entire expression to be `false` and terminates its evaluation of the expression early.

Specify `b = 0` and evaluate the same expression.

```
b = 0;
x = (b ~= 0) && (a/b > 18.5)

x =

     0
```

The result is logical `0` (`false`). The first statement evaluates to logical `0` (`false`), so the expression short-circuits.

Change Structure Field Value

Create a structure with fields named `'File'` and `'Format'`.

```
S = struct('File',{'myGraph'},'Format',[])
```

```
S =

      File: 'myGraph'
    Format: []
```

Short-circuit expressions are useful in *if* statements when you want multiple conditions to be true. The conditions can build on one another in such a way that it only makes sense to evaluate the second expression if the first expression is true.

Specify an *if* statement that executes only when `S` contains an empty field named `'Format'`.

```
if isfield(S,'Format') && isempty(S.Format)
  S.Format = '.png';
end
S
```

```
S =

      File: 'myGraph'
    Format: '.png'
```

The first condition tests if the string `'Format'` is the name of a field in structure `S`. The second statement then tests whether the `Format` field is empty. The truth of the second condition depends on the first. The second condition can never be true if the first condition is not true. Since `S` has an empty field named `'Format'`, the body statement executes and assigns `S.Format` the value `'.png'`.

More About

Logical Short-Circuiting

With logical short-circuiting, the second operand, `expr2`, is evaluated only when the result is not fully determined by the first operand, `expr1`.

Due to the properties of logical AND and OR, the result of a logical expression is sometimes fully determined before evaluating all of the conditions. The logical `and` operator returns logical `0` (`false`) if even a single condition in the expression is false. The logical `or` operator returns logical `1` (`true`) if even a single condition in the expression is true. When the evaluation of a logical expression terminates early by encountering one of these values, the expression is said to have *short-circuited*.

For example, in the expression `A && B`, MATLAB does not evaluate condition `B` at all if condition `A` is false. If `A` is false, then the value of `B` does not change the outcome of the operation.

When you use the element-wise `&` and `|` operators in the context of an `if` or `while` loop expression (and *only* in that context), they use short-circuiting to evaluate expressions.

Note: Always use the `&&` and `||` operators to enable short-circuit evaluation. Using the `&` and `|` operators for short-circuiting can yield unexpected results when the expressions do not evaluate to logical scalars.

- "Reduce Logical Arrays to Single Value"

See Also

`all` | `and` | `any` | `false` | `find` | `logical` | `or` | `true` | `xor`

Special Characters [ ] ( ) {} = ' . ... , ; : % ! @

Special characters

Syntax

```
[ ]
{ }
( )
=
'
.
.
.( )
..
...
,
;
:
%
%{ %}
!
@
```

Description

`[ ]` Brackets are used to form vectors and matrices. `[6.9 9.64 sqrt(-1)]` is a vector with three elements separated by blanks. `[6.9, 9.64, i]` is the same thing. `[1+j 2-j 3]` and `[1 +j 2 -j 3]` are not the same. The first has three elements, the second has five.

`[11 12 13; 21 22 23]` is a 2-by-3 matrix. The semicolon ends the first row.

Vectors and matrices can be used inside `[ ]` brackets. `[A B;C]` is allowed if the number of rows of `A` equals the number of rows of `B` and the number of columns of `A` plus the number of columns of `B` equals the number of columns of `C`. This rule generalizes to allow fairly complicated constructions.

`A = [ ]` stores an empty matrix in `A`. `A(m,:) = [ ]` deletes row `m` of `A`. `A(:,n) = [ ]` deletes column `n` of `A`. `A(n) = [ ]` reshapes `A` into a column vector and deletes the `n`th element.

`[A1,A2,A3...] = function` assigns function output to multiple variables.

For the use of `[` and `]` on the left of an "=" in multiple assignment statements, see `lu`, `eig`, `svd`, and so on.

`{ }` Curly braces are used in cell array assignment statements. For example, `A(2,1) = {[1 2 3; 4 5 6]}`, or `A{2,2} = ('str')`. See `help paren` for more information about `{ }`.

`( )` Parentheses are used to indicate precedence in arithmetic expressions in the usual way. They are used to enclose arguments of functions in the usual way. They are also used to enclose subscripts of vectors and matrices in a manner somewhat more general than usual. If `X` and `V` are vectors, then `X(V)` is `[X(V(1)), X(V(2)), ..., X(V(n))]`. The components of `V` must be integers to be used as subscripts. An error occurs if any such subscript is less than 1 or greater than the size of `X`. Some examples are

- `X(3)` is the third element of `X`.
- `X([1 2 3])` is the first three elements of `X`.

  See `help paren` for more information about `( )`.

If `X` has `n` components, `X(n:-1:1)` reverses them. The same indirect subscripting works in matrices. If `V` has `m` components and `W` has `n` components, then `A(V,W)` is the `m`-by-`n` matrix formed from the elements of `A` whose subscripts are the elements of `V` and `W`. For example, `A([1,5],:) = A([5,1],:)` interchanges rows 1 and 5 of `A`.

`=` Used in assignment statements. `B = A` stores the elements of `A` in `B`. == is the relational equals operator. See the Relational Operators page.

`'` Matrix transpose. `X'` is the complex conjugate transpose of `X`. `X.'` is the nonconjugate transpose.

Quotation mark. `'any text'` is a vector whose components are the ASCII codes for the characters. A quotation mark within the text is indicated by two quotation marks.

`.` Decimal point. `314/100`, `3.14`, and `.314e1` are all the same.

Element-by-element operations. These are obtained using `.*` , `.^`, `./`, or `.\`. See the Arithmetic Operators page.

`.` Field access. `S(m).f` when `S` is a structure, accesses the contents of field `f` of that structure.

`.( )` Dynamic Field access. `S.(df)` when `S` is a structure, accesses the contents of dynamic field `df` of that structure. Dynamic field names are defined at runtime.

`..` Parent folder. See `cd`.

`...` Continuation. Three or more periods at the end of a line continue the current function on the next line. Three or more periods before the end of a line cause the MATLAB software to ignore the remaining text on the current line and continue the function on the next line. This effectively makes a comment out of anything on the current line that follows the three periods. For an example, see "Continue Long Statements on Multiple Lines".

`,` Comma. Used to separate matrix subscripts and function arguments. Used to separate statements in multistatement lines. For multistatement lines, the comma can be replaced by a semicolon to suppress printing.

`;` Semicolon. Used inside brackets to end rows. Used after an expression or statement to suppress printing or to separate statements.

`:` Colon. Create vectors, array subscripting, and `for` loop iterations. See `colon (:)` for details.

`%` Percent. The percent symbol denotes a comment; it indicates a logical end of line. Any following text is ignored. MATLAB displays the first contiguous comment lines in a function or script file in response to a `help` command.

`%{ %}` Percent-brace. The text enclosed within the `%{` and `%}` symbols is a comment block. Use these symbols to insert comments that take up more than a single line in your script of function code. Any text between these two symbols is ignored by MATLAB.

With the exception of whitespace characters, the %{ and %} operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

`!` Exclamation point. Indicates that the rest of the input line is issued as a command to the operating system. See "Run External Commands, Scripts, and Programs" for more information.

`@`	Function handle. MATLAB data type that is a handle to a function. See `function_handle (@)` for details.

More About

Tips

Some uses of special characters have function equivalents, as shown:

Horizontal concatenation	`[A,B,C...]`	`horzcat(A,B,C...)`
Vertical concatenation	`[A;B;C...]`	`vertcat(A,B,C...)`
Subscript reference	`A(i,j,k...)`	`subsref(A,S)`. See help `subsref`.
Subscript assignment	`A(i,j,k...)= B`	`subsasgn(A,S,B)`. See help `subsasgn`.

Note For some toolboxes, the special characters are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given character, type `help` followed by the character name. For example, type `help transpose`. The toolboxes that overload `transpose` (`.'`) are listed. For information about using the character in that toolbox, see the documentation for the toolbox.

- "Logical Operations"
- "Relational Operations"

See Also

"Array vs. Matrix Operations"

colon (:)

Create vectors, array subscripting, and `for`-loop iterators

Description

The colon is one of the most useful operators in MATLAB. It can create vectors, subscript arrays, and specify `for` iterations.

The colon operator uses the following rules to create regularly spaced vectors for scalar values `i`, `j`, and `k`:

`j:k`	is the same as `[j,j+1,j+2,...,j+m]`, where `m = fix(k-j)`. In the case where both `j` and `k` are integers, this is simply `[j,j+1,...,k]`. This syntax returns an empty matrix when `j > k`.
`j:i:k`	is the same as `[j,j+i,j+2i, ...,j+m*i]`, where `m = fix((k-j)/i)`. This syntax returns an empty matrix when `i == 0`, `i > 0` and `j > k`, or `i < 0` and `j < k`.

If `i`, `j`, or `k` is an empty input, then the colon operator returns an empty 1-by-0 matrix. If you specify nonscalar arrays, MATLAB interprets `j:i:k` as `j(1):i(1):k(1)`.

You can use the colon to create a vector of indices to select rows, columns, or elements of arrays, where:

`A(:,j)`	is the `j`th column of `A`.
`A(i,:)`	is the `i`th row of `A`.
`A(:,:)`	is the equivalent two-dimensional array. For matrices this is the same as `A`.
`A(j:k)`	is `A(j), A(j+1),...,A(k)`.
`A(:,j:k)`	is `A(:,j), A(:,j+1),...,A(:,k)`.
`A(:,:,k)`	is the `k`th page of three-dimensional array `A`.
`A(i,j,k,:)`	is a vector in four-dimensional array `A`. The vector includes `A(i,j,k,1)`, `A(i,j,k,2)`, `A(i,j,k,3)`, and so on.

`A(:)` is all the elements of `A`, regarded as a single column. On the left side of an assignment statement, `A(:)` fills `A`, preserving its shape from before. In this case, the right side must contain the same number of elements as `A`.

When you create a vector to index into a cell array or structure array (such as *`cellName`*`{:}` or *`structName`*`(:).`*`fieldName`*), MATLAB returns multiple outputs in a comma-separated list. For more information, see "How to Use the Comma-Separated Lists" in the MATLAB Programming Fundamentals documentation.

Examples

Using the colon with integers,

```
D = 1:4
```

results in

```
D =
    1    2    3    4
```

Using two colons to create a vector with arbitrary real increments between the elements,

```
E = 0:.1:.5
```

results in

```
E =
    0    0.1000    0.2000    0.3000    0.4000    0.5000
```

The command

```
A(:,:,2) = pascal(3)
```

generates a three-dimensional array whose first page is all zeros.

```
A(:,:,1) =
     0     0     0
     0     0     0
     0     0     0

A(:,:,2) =
     1     1     1
```

```
     1     2     3
     1     3     6
```

Using a colon with characters to iterate a for-loop,

```
for x='a':'d',x,end
```

results in

```
x =
    a
x =
    b
x =
    c
x =
    d
```

See Also

`for` | `linspace` | `logspace` | `reshape` | `varargin`

abs

Absolute value and complex magnitude

Syntax

```
Y = abs(X)
```

Description

`Y = abs(X)` returns the absolute value of each element in array `X`.

If `X` is complex, `abs(X)` returns the complex magnitude.

Examples

Absolute Value of Scalar

```
y = abs(-5)
```

```
y =

     5
```

Absolute Value of Vector

Create a numeric vector of real values.

```
x = [1.3 -3.56 8.23 -5 -0.01]'
```

```
x =

    1.3000
   -3.5600
    8.2300
   -5.0000
   -0.0100
```

Find the absolute value of the elements of the vector.

```
y = abs(x)

y =

    1.3000
    3.5600
    8.2300
    5.0000
    0.0100
```

Magnitude of Complex Number

```
y = abs(3+4i)

y =

     5
```

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `X` can be a `single` array, `double` array, signed integer array, or `duration` array. The size and data type of the output array is the same as the input array.
Complex Number Support: Yes

More About

Absolute Value

The absolute value (or modulus) of a real number is the corresponding nonnegative value that disregards the sign.

For a real value, `a`, the absolute value is:

- `a`, if `a` is greater than or equal to zero
- `-a`, if `a` is less than zero

`abs(-0)` returns `0`.

Complex Magnitude

The complex magnitude (or modulus) is the length of a vector from the origin to a complex value plotted in the complex plane.

For a complex value, $|a+bi|$ is defined as $\sqrt{a^2+b^2}$.

See Also

`angle` | `hypot` | `sign` | `unwrap`

Introduced before R2006a

accumarray

Construct array with accumulation

Syntax

```
A = accumarray(subs,val)
A = accumarray(subs,val,sz)
A = accumarray(subs,val,sz,fun)
A = accumarray(subs,val,sz,fun,fillval)
A = accumarray(subs,val,sz,fun,fillval,issparse)
```

Description

`A = accumarray(subs,val)` returns an array, `A`, by accumulating elements of vector `val` using the subscripts in `subs`. The values in each row of the *m*-by-*n* matrix `subs` define an *n*-dimensional subscript into the output, `A`. If `subs` is a column vector, the value in each row defines a row subscript into the output, which is also a column vector.

The ith row of `subs` corresponds to the ith element in the vector, `val`. The function collects all elements of `val` that have identical subscripts in `subs`, applies the function `@sum`, and then stores the result in the location of `A` corresponding to the subscript. Elements of `A` that are not referred to by any row of `subs` contain the value `0`.

`A = accumarray(subs,val,sz)` returns an array, `A`, with size `sz`. Specify `sz` as a vector of positive integers to define the size of the output, or as `[]` to let the subscripts in `subs` determine the size of the output. Use `sz` when `subs` does not reference trailing rows, columns, or dimensions that you would like to be present in the output.

`A = accumarray(subs,val,sz,fun)` applies the function `fun` to each subset of elements in `val` that have identical subscripts in `subs`. Specify `fun` using the `@` symbol (e.g., `@mean`), or as `[]` to use the default function, `@sum`.

`A = accumarray(subs,val,sz,fun,fillval)` fills all elements of `A` that are not referred to by any subscript in `subs` with the scalar value, `fillval`. The `fillval` input must have the same class as the values returned by `fun`. Specify `fillval` as `[]` to use the default value, `0`.

`A = accumarray(subs,val,sz,fun,fillval,issparse)` returns an array, `A`, that is sparse if the scalar `issparse` is `true` or `1`, and full if `issparse` is `false` or `0`. The output, `A`, is full by default.

Examples

Find Bin Counts

Create a vector of subscripts, `subs`.

```
subs = [1; 2; 4; 2; 4]
```

```
subs =
     1
     2
     4
     2
     4
```

Use `accumarray` with `val = 1` to count the number of identical subscripts in `subs`.

```
A = accumarray(subs,1)
```

```
A =

     1
     2
     0
     2
```

The result is a vector of bin counts. You can obtain the same answer with `histcounts(subs,'BinMethod','integers')`. However, `accumarray` also can compute bin counts over higher dimensional grids.

Accumulate Data

Create a vector of data, `val`, and a vector of subscript values with the same length, `subs`.

```
val = 101:105';
subs = [1; 3; 4; 3; 4]
```

```
subs =
```

```
     1
     3
     4
     3
     4
```

Use `accumarray` to sum the values in `val` that have identical subscripts in `subs`.

```
A = accumarray(subs,val)
```

```
A =
   101
     0
   206
   208
```

The result is a vector of accumulated values. Since the second and fourth elements of `subs` are equal to 3, `A(3)` is the sum of the second and fourth elements of `val`, that is, `A(3) = 102 + 104 = 206`. Also, `A(2) = 0` because `subs` does not contain the value `2`. Since `subs` is a vector, the output, `A`, is also a vector. The length of `A` is `max(subs, [],1)`.

Specify Output Size

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 101:106';
subs = [1 1; 2 2; 3 2; 1 1; 2 2; 4 1]
```

```
subs =

     1     1
     2     2
     3     2
     1     1
     2     2
     4     1
```

The subscripts in `subs` define a 4-by-2 matrix for the output.

Use `accumarray` to sum the values in `val` that have identical subscripts in `subs`.

```
A = accumarray(subs,val)
```

```
A =
```

```
   205     0
     0   207
     0   103
   106     0
```

The result is a 4-by-2 matrix of accumulated values.

Use the `sz` input of `accumarray` to return a 4-by-4 matrix. You can specify a size with each dimension equal to or greater than the default size, in this case 4-by-2, but not smaller.

```
A = accumarray(subs,val,[4 4])
```

```
A =

   205     0     0     0
     0   207     0     0
     0   103     0     0
   106     0     0     0
```

The result is a 4-by-4 matrix of accumulated values.

Use Custom Functions

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = [100.1 101.2 103.4 102.8 100.9 101.5]';
subs = [1 1; 1 1; 2 2; 3 2; 2 2; 3 2]
```

```
subs =

     1     1
     1     1
     2     2
     3     2
     2     2
     3     2
```

The subscripts in `subs` define a 3-by-2 matrix for the output.

Use the `fun` input of `accumarray` to calculate the within-group variances of data in `val` that have identical subscripts in `subs`. Specify `fun` as `@var`.

```
A1 = accumarray(subs,val,[],@var)
```

```
A1 =

    0.6050         0
         0    3.1250
         0    0.8450
```

The result is a 3-by-2 matrix of variance values.

Alternatively, you can specify `fun` as an anonymous function so long as it accepts vector inputs and returns a scalar. A common situation where this is useful is when you want to pass additional parameters to a function. In this case, use the `var` function with a normalization parameter.

```
A2 = accumarray(subs,val,[],@(x) var(x,1))
```

```
A2 =

    0.3025         0
         0    1.5625
         0    0.4225
```

The result is a 3-by-2 matrix of normalized variance values.

Sum Values Natively

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = int8(10:15);
subs = [1 1 1; 1 1 1; 1 1 2; 1 1 2; 2 3 1; 2 3 2]
```

```
subs =

     1     1     1
     1     1     1
     1     1     2
     1     1     2
     2     3     1
     2     3     2
```

The subscripts in `subs` define a 2-by-3-by-2 multidimensional array for the output.

Use `accumarray` to sum the data values in `val` that have identical subscripts in `subs`. You can use a function handle to sum the values in their native, `int8`, integer class by using the `'native'` option of the `sum` function.

```
A = accumarray(subs,val,[],@(x) sum(x,'native'))

A(:,:,1) =

   21    0    0
    0    0   14


A(:,:,2) =

   25    0    0
    0    0   15
```

The result is a 2-by-3-by-2 multidimensional array of class `int8`.

Group Values in Cell Array

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 1:10;
subs = [1 1;1 1;1 1;1 1;2 1;2 1;2 1;2 1;2 1;2 2]

subs =

     1     1
     1     1
     1     1
     1     1
     2     1
     2     1
     2     1
     2     1
     2     1
     2     2
```

The subscripts in `subs` define a 2-by-2 matrix for the output.

Use `accumarray` to group the elements of `val` into a cell array.

```
A = accumarray(subs,val,[],@(x) {x})

A =

    [4x1 double]      []
```

```
    [5x1 double]    [10]
```

The result is a 2-by-2 cell array.

Verify that the vector elements are in the same order as they appear in `val`.

```
A{2,1}
```

```
ans =

     5
     6
     7
     8
     9
```

Since the subscripts in `subs` are sorted, the elements of the numeric vectors in the cell array are in the same order as they appear in `val`.

Using Functions That Depend on Data Order

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 1:5;
subs = [1 2; 1 1; 1 2; 1 1; 2 3]
```

```
subs =

     1     2
     1     1
     1     2
     1     1
     2     3
```

The subscripts in `subs` define a 2-by-3 matrix for the output, but are unsorted with respect to the linear indices in the output, `A`.

Group the values in `val` into a cell array by specifying `fun = @(x) {x}`.

```
A = accumarray(subs,val,[],@(x) {x})
```

```
A =

    [2x1 double]    [2x1 double]     []
```

```
    []        []    [5]
```

The result is a 2-by-3 cell array.

Examine the vector in `A{1,2}`.

```
A{1,2}

ans =

     3
     1
```

The elements of the `A{1,2}` vector are in a different order than in `val`. The first element of the vector is 3 instead of 1. If the subscripts in `subs` are not sorted with respect to their linear indices, then `accumarray` might not always preserve the order of the data in `val` when it passes them to `fun`. In the unusual case that `fun` requires that its input values be in the same order as they appear in `val`, sort the indices in `subs` with respect to the linear indices of the output.

In this case, use the `sortrows` function with two inputs and two outputs to reorder `subs` and `val` concurrently with respect to the linear indices of the output.

```
[S,I] = sortrows(subs,[2,1]);
A = accumarray(S,val(I),[],@(x) {x});
A{1,2}

ans =

     1
     3
```

The elements of the `A{1,2}` vector are now in sorted order.

Fill Output with NaN Values

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 101:106';
subs = [1 1; 2 2; 3 3; 1 1; 2 2; 4 4]

subs =
```

```
     1     1
     2     2
     3     3
     1     1
     2     2
     4     4
```

The subscripts in `subs` define a 4-by-4 matrix for the output, but only reference 4 out of the 16 elements. By default, the other 12 elements are `0` in the output.

Use the `fillval` input of `accumarray` to fill in the extra output elements with NaN values.

```
A = accumarray(subs,val,[],[],NaN)
```

```
A =

   205   NaN   NaN   NaN
   NaN   207   NaN   NaN
   NaN   NaN   103   NaN
   NaN   NaN   NaN   106
```

The result is a 4-by-4 matrix padded with NaN values.

Change Output Sparsity

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = [34 22 19 85 53 77 99 6];
subs = [1 1; 400 400; 80 80; 1 1; 400 400; 400 400; 80 80; 1 1]
```

```
subs =

     1     1
   400   400
    80    80
     1     1
   400   400
   400   400
    80    80
     1     1
```

The subscripts in `subs` define a 400-by-400 matrix for the output, but only reference 3 out of the 160,000 elements. When the result of an operation with `accumarray` leads to

a large output array with low density of nonzero elements, you can save storage space by storing the output as a sparse matrix.

Use the `issparse` input of `accumarray` to return a sparse matrix.

```
A = accumarray(subs,val,[],[],[],true)
```

```
A =

   (1,1)       125
  (80,80)      118
 (400,400)     152
```

The result is a sparse matrix. You can obtain the same answer with `sparse(subs(:,1),subs(:,2),val)`.

Input Arguments

`subs` — Subscript matrix

vector of indices | matrix of indices | cell array of index vectors

Subscript matrix, specified as a vector of indices, matrix of indices, or cell array of index vectors. The indices must be positive integers:

- The value in each row of the m-by-n matrix, `subs`, specifies an n-dimensional index into the output, `A`. For example, if `subs` is a 3-by-2 matrix, it contains three 2-D subscripts. `subs` also can be a column vector of indices, in which case the output, `A`, is also a column vector.
- The ith row in `subs` corresponds to the ith data value in `val`.

Thus, `subs` determines which data in `val` to group, as well as its final destination in the output. If `subs` is a cell array of index vectors, each vector must have the same length, and the function treats the vectors as columns of a subscript matrix.

`val` — Data

vector | scalar

Data, specified as a vector or scalar:

- If `val` is a vector, it must have the same length as the number of rows in `subs`.

- If `val` is a scalar, it is scalar expanded.

In both cases, a one-to-one pairing is present between the subscripts in each row of `subs` and the data values in `val`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

`sz` — Size of output array

`[]` (default) | vector of positive integers

Size of output array, specified as a vector of positive integers or `[]` (default). When you specify `[]` for the default size, the values in `subs` determine the size of the output array, `A`.

When you specify `sz` as a vector of positive integers, it must satisfy these properties:

- If `subs` is a nonempty *m*-by-*n* matrix with `n > 1` columns, then `sz` must have *n* elements and pass the logical test `all(sz >= max(subs,[],1))`.
- If `subs` is a nonempty column vector, then `sz` must be `[m 1]` where `m >= max(subs)`.

Example: `sz = [3 3]`

`fun` — Function

`[]` (default) | function handle

Function, specified as a function handle or `[]` (default). The default function is `@sum`. The `fun` function must accept a column vector and return a numeric, `logical`, or `char` scalar, or a scalar `cell`. If the subscripts in `subs` are not sorted with respect to their linear indices, `fun` should not depend on the order of the values in its input data.

Example: `fun = @max`

Data Types: `function_handle`

`fillval` — Fill value

`[]` (default) | scalar

Fill value, specified as a scalar or `[]` (default). The default value of `fillval` is `0`. If `subs` does not reference each element in the output, `accumarray` fills in the output with

the value specified by `fillval`. The class of `fillval` must be the same as the values returned by `fun`.

`issparse` — Output sparsity

`false` (default) | `true` | `1` | `0`

Output sparsity, specified as `true`, `1`, `0`, or `false` (default). Specify `true` or `1` when you want the output array to be sparse. If `issparse` is `true` or `1`:

- `fillval` must be `0` or `[]`.
- The values in `val` and the output values of `fun` must both have type `double`.

Output Arguments

`A` — Output array

vector | matrix | multidimensional array

Output array, returned as a vector, matrix, or multidimensional array. `A` has the same class as the values returned by `fun`.

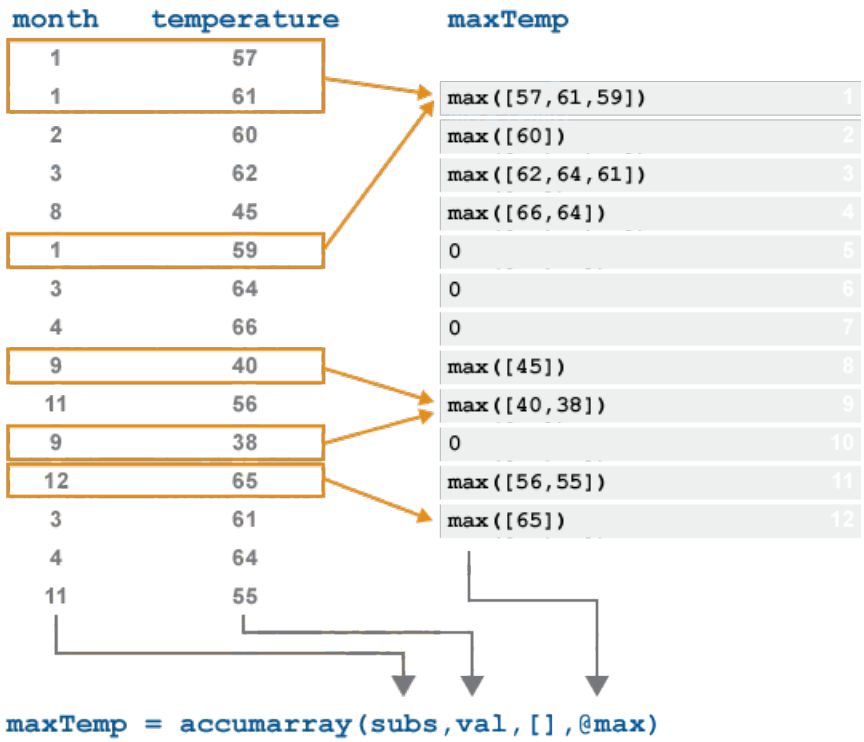
When `sz` is not specified, the size of `A` depends on the input `subs`:

- If `subs` is a nonempty matrix with `n > 1` columns, then `A` is an *n*-dimensional array of size `max(subs,[],1)`.
- If `subs` is an empty matrix with `n > 1` columns, then `A` is an *n*-dimensional empty array with size 0-by-0-by-...-by-0.
- If `subs` is a nonempty column vector, then `A` is a column vector of length `max(subs,[],1)`. The length of `A` is `0` when `subs` is empty.

More About

Accumulating Elements

The following graphic illustrates the behavior of `accumarray` on a vector of temperature data taken over a 12-month period. To find the maximum temperature reading for each month, `accumarray` applies the `max` function to each group of values in `temperature` that have identical subscripts in `month`.



No values in `month` point to the 5, 6, 7, or 10 positions of the output. These elements are `0` in the output by default, but you can specify a value to fill in using `fillval`.

Tips

- The behavior of `accumarray` is similar to that of the `histcounts` function. Both functions group data into bins.

  - `histcounts` groups continuous values into a 1-D range using bin edges. `accumarray` groups data using *n*-dimensional subscripts.
  - `histcounts` returns the bin counts and/or bin placement. However, `accumarray` can apply any function to the binned data.

  You can mimic the behavior of `histcounts` using `accumarray` with `val = 1`.

- The `sparse` function also has accumulation behavior similar to that of `accumarray`.

- `sparse` groups data into bins using 2-D subscripts, whereas `accumarray` groups data into bins using *n*-dimensional subscripts.
- `sparse` adds elements that have identical subscripts into the output. `accumarray` adds elements that have identical subscripts into the output by default, but can optionally apply any function to the bins.

See Also

`full` | `function_handle` | `histcounts` | `sparse` | `sum`

Introduced before R2006a

acos

Inverse cosine in radians

Syntax

```
y = acos(x)
```

Description

`y = acos(x)` returns the "Inverse Cosine" on page 1-101 ($\cos^{-1}$) of the elements of x. The `acos` function operates element-wise on arrays. For real elements of x in the interval `[-1,1]`, `acos(x)` returns real values in the interval `[0,pi]`. For real values of x outside the interval `[-1,1]` and for complex values of x, `acos(x)` returns complex values. All angles are in radians.

Examples

Inverse Cosine of a Value

```
acos(0.3)
```

```
ans =
    1.2661
```

Inverse Cosine of a Vector of Complex Values

Find the inverse cosine of the elements of vector `x`. The `acos` function acts on `x` element-wise.

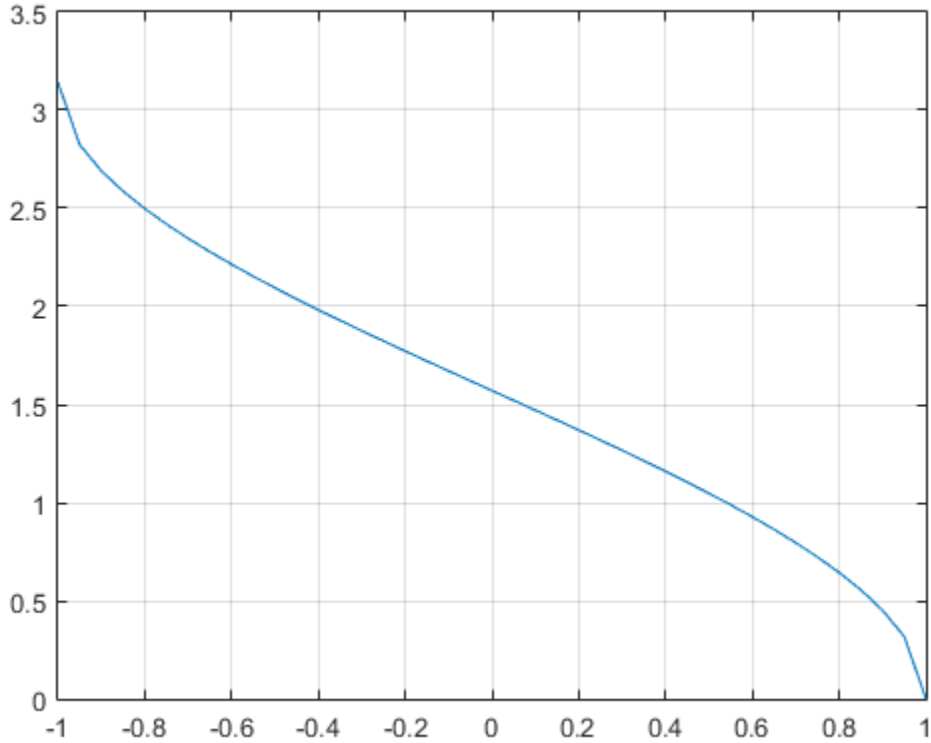
```
x = [0.5i 1+3i -2.2+i];
y = acos(x)
```

```
y =
   1.5708 - 0.4812i   1.2632 - 1.8642i   2.6799 - 1.5480i
```

Plot the Inverse Cosine Function

Plot the inverse cosine function over the domain $-1 \le x \le 1$.

```
x = -1:.05:1;
plot(x,acos(x))
grid on
```



Input Arguments

x — Numeric input

number | vector | matrix | multidimensional array

Numeric input, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Complex Number Support: Yes

More About

Inverse Cosine

The inverse cosine is defined as

$$\cos^{-1}(z) = -i\log\left[z + i\left(1 - z^2\right)^{1/2}\right].$$

See Also

acosd | acosh | cos

Introduced before R2006a

acosd

Inverse cosine in degrees

Syntax

```
Y = acosd(X)
```

Description

`Y = acosd(X)` returns the inverse cosine ($\cos^{-1}$) of the elements of X in degrees. The function's domain and range include complex values. For real elements of X in the domain [-1,1], `acosd` returns values in the range [0, 180]. For values of X outside this range, `acosd` returns complex values.

Examples

Inverse Cosine of 0

Verify that inverse cosine of 0 is exactly 90.

```
acosd(0)
```

```
ans =

     90
```

Round-Trip Calculation for Complex Angles

Show that the inverse cosine, followed by cosine, returns the original values of X.

```
cosd(acosd([2 3]))
```

```
ans =

    2.0000    3.0000
```

`acosd([2 3])` returns two complex angles, which are then passed to the `cosd` function. `cosd` returns the original values, `2` and `3`.

Input Arguments

X — Cosine of angle

scalar value | vector | matrix | N-D array

Cosine of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `acosd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

See Also

`acos` | `cos` | `cosd`

Introduced before R2006a

acosh

Inverse hyperbolic cosine

Syntax

```
Y = acosh(X)
```

Description

`Y = acosh(X)` returns the inverse hyperbolic cosine for each element of `X`.

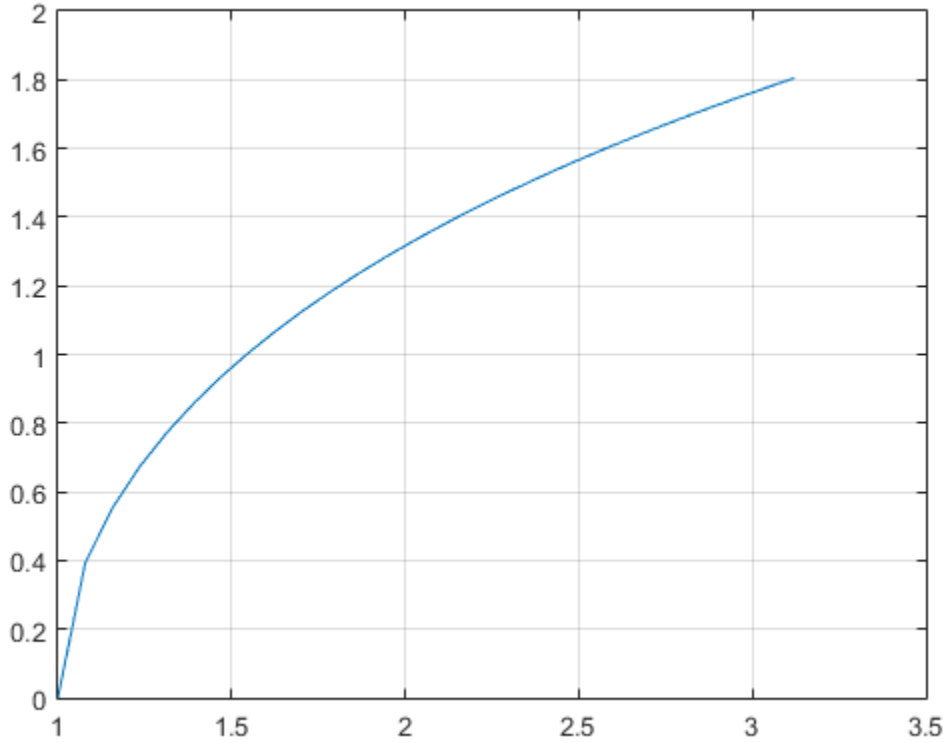
The `acosh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph of Inverse Hyperbolic Cosine Function

Graph the inverse hyperbolic cosine function over the domain $-1 \le x \le \pi$.

```
x = 1:pi/40:pi;
plot(x,acosh(x)), grid on
```



More About

Inverse Hyperbolic Cosine

The inverse hyperbolic cosine can be defined as

$$\cosh^{-1}(z) = \log\left[z + \left(z^2 - 1\right)^{1/2}\right].$$

See Also

`acos` | `cosh` | `asinh` | `atanh`

Introduced before R2006a

acot

Inverse cotangent in radians

Syntax

```
Y = acot(X)
```

Description

`Y = acot(X)` returns the "Inverse Cotangent" on page 1-109 ($\cot^{-1}$) of the elements of X. The `acot` function operates element-wise on arrays. For real elements of X, `acot(X)` returns values in the interval `[-pi/2,pi/2]`. For complex values of X, `acot(X)` returns complex values. All angles are in radians.

Examples

Inverse Cotangent of a Value

```
acot(2.6)
```

```
ans =
    0.3672
```

Inverse Cotangent of a Vector of Complex Values

Find the inverse cotangent of the elements of vector `x`. The `acot` function acts on `x` element-wise.

```
x = [0.5i 1+3i -2.2+i];
Y = acot(x)
```

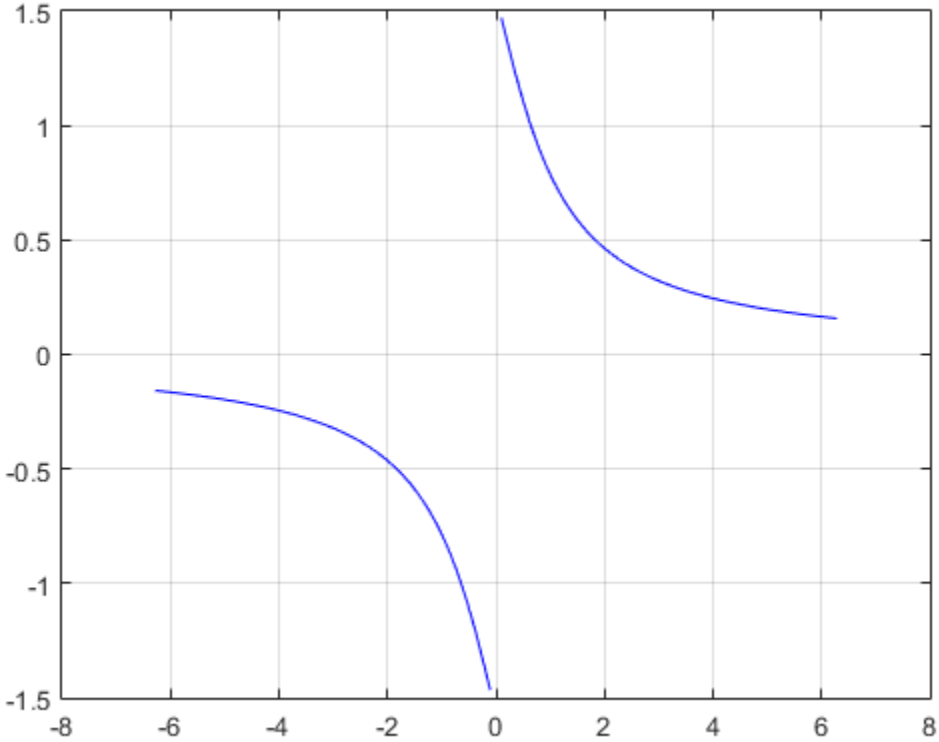
```
Y =
   1.5708 - 0.5493i   0.1093 - 0.3059i  -0.3689 - 0.1506i
```

Plot the Inverse Cotangent Function

Plot the inverse cotangent function over the intervals $-2\pi \le x < 0$ and $0 < x \le 2\pi$.

```
x1 = -2*pi:pi/30:-0.1;
```

```
x2 = 0.1:pi/30:2*pi;
plot(x1,acot(x1),'b')
hold on
plot(x2,acot(x2),'b')
grid on
```



Input Arguments

X — Numeric input

number | vector | matrix | multidimensional array

Numeric input, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Cotangent

The inverse cotangent is defined as

$$\cot^{-1}(z) = \tan^{-1}\left(\frac{1}{z}\right).$$

See Also

`acotd` | `acoth` | `cot`

Introduced before R2006a

acotd

Inverse cotangent in degrees

Syntax

```
Y = acotd(X)
```

Description

`Y = acotd(X)` returns the inverse cotangent ($\cot^{-1}$) of the elements of X in degrees. The function's domain and range include complex values. For real elements of X in the range [-Inf,Inf], `acotd` returns values in the range [-90,90]. For complex values of X, `acotd` returns complex values.

Examples

Inverse Cotangent of Vector

```
x = [0 20 Inf];
y = acotd(x)
```

```
y =

   90.0000    2.8624         0
```

The `acotd` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Cotangent of Complex Value

```
acotd(1+i)
```

```
ans =
```

```
31.7175 -23.0535i
```

Input Arguments

X — Cotangent of angle

scalar value | vector | matrix | N-D array

Cotangent of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `acotd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

See Also

`acot` | `cot` | `cotd`

Introduced before R2006a

acoth

Inverse hyperbolic cotangent

Syntax

```
Y = acoth(X)
```

Description

`Y = acoth(X)` returns the inverse hyperbolic cotangent for each element of `X`.

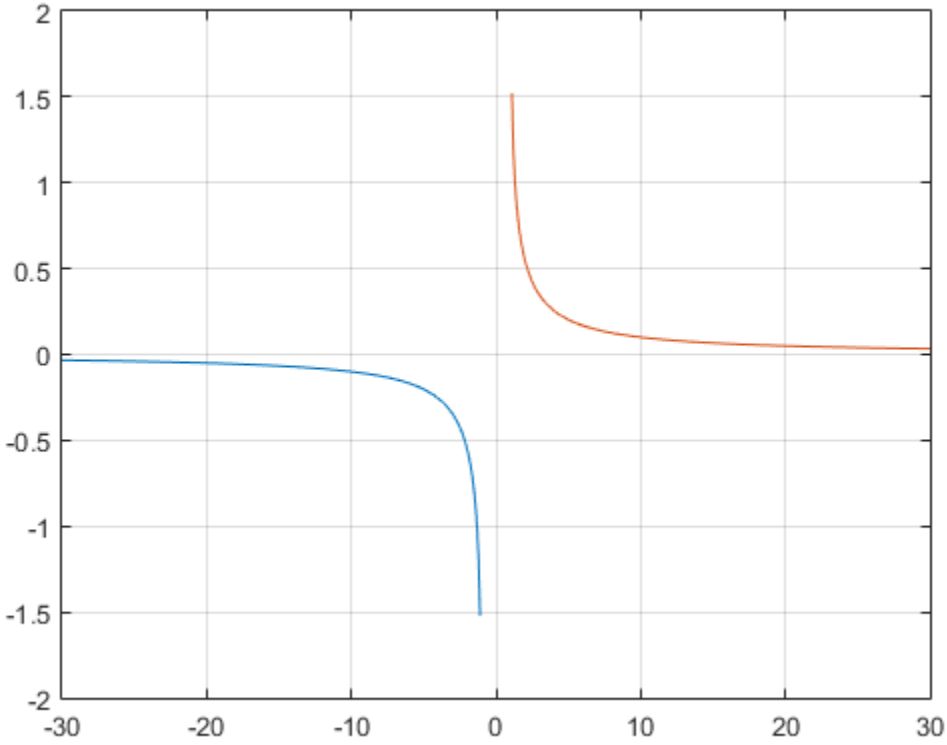
The `acoth` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph of Inverse Hyperbolic Cotangent Function

Graph the inverse hyperbolic cotangent function over the domains $-30 \le x < -1$ and $1 < x \le 30$.

```
x1 = -30:0.1:-1.1;
x2 = 1.1:0.1:30;
plot(x1,acoth(x1),x2,acoth(x2)), grid on
```



More About

Inverse Hyperbolic Cotangent

The inverse hyperbolic cotangent can be defined as

$$\coth^{-1}(z) = \tanh^{-1}\left(\frac{1}{z}\right).$$

See Also

`acot` | `coth` | `atanh` | `asinh` | `acosh`

Introduced before R2006a

acsc

Inverse cosecant in radians

Syntax

```
Y = acsc(X)
```

Description

`Y = acsc(X)` returns the "Inverse Cosecant" on page 1-117 ($\csc^{-1}$) of the elements of X. The `acsc` function operates element-wise on arrays. For real elements of X in intervals `[-Inf,-1]` and `[1,Inf]`, `acsc` returns real values in the interval `[-pi/2,pi/2]`. For real values of X in the interval `[-1,1]` and for complex values of X, `acsc` returns complex values. All angles are in radians.

Examples

Inverse Cosecant of a Value

```
acsc(3)
```

```
ans =
    0.3398
```

Inverse Cosecant of a Vector of Complex Angles

Find the inverse cosecant of the elements of vector `x`. The `acsc` function acts on `x` element-wise.

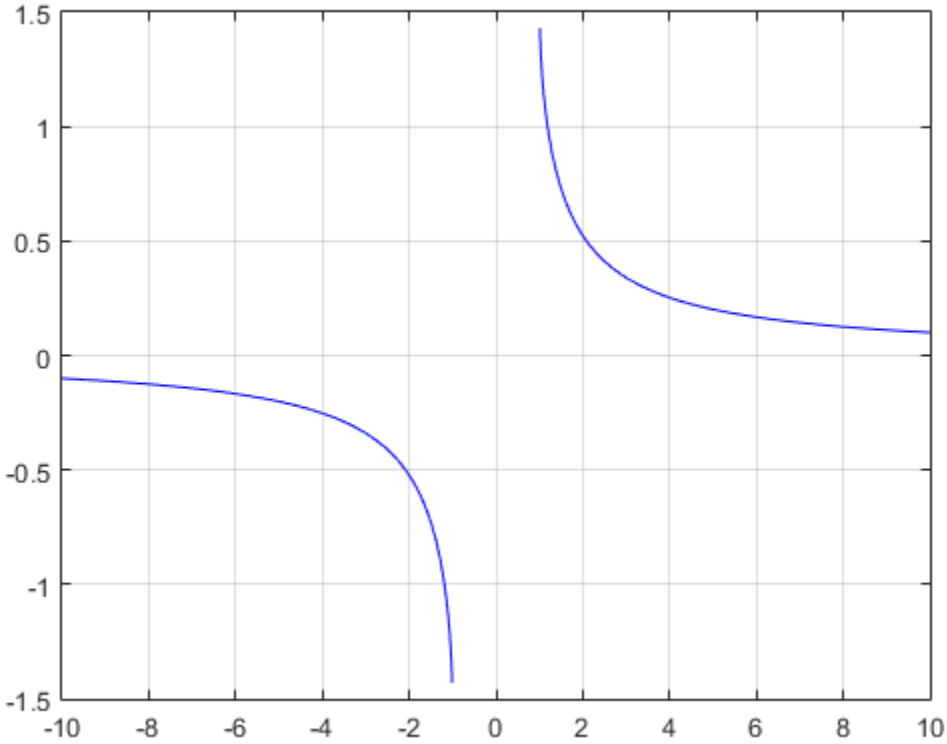
```
x = [0.5i 1+3i -2.2+i];
Y = acsc(x)
```

```
Y =
   0.0000 - 1.4436i   0.0959 - 0.2970i  -0.3795 - 0.1833i
```

Plot the Inverse Cosecant Function

Plot the inverse cosecant function over the intervals $-10 \le x < -1$ and $1 < x \le 10$.

```
x1 = -10:0.01:-1.01;
x2 = 1.01:0.01:10;
plot(x1,acsc(x1),'b')
hold on
plot(x2,acsc(x2),'b')
grid on
```



Input Arguments

X — Numeric input

number | vector | matrix | multidimensional array

Numeric input, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Cosecant

The inverse cosecant is defined as

$$\csc^{-1}(z) = \sin^{-1}\left(\frac{1}{z}\right).$$

See Also

`acscd` | `acsch` | `csc`

Introduced before R2006a

acscd

Inverse cosecant in degrees

Syntax

```
Y = acscd(X)
```

Description

`Y = acscd(X)` returns the inverse cosecant (cosec^{-1}) of the elements of X in degrees. The function's domain and range include complex values. For real elements of X in the domain [-Inf,1] and [1,Inf], `acscd` returns values in the range [-90,90]. For values of X outside this range, `acscd` returns complex values.

Examples

Inverse Cosecant of Vector

```
x = [20 10 Inf];
y = acscd(x)
```

```
y =

    2.8660    5.7392         0
```

The `acscd` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Cosecant of Complex Value

```
acscd(1+i)
```

```
ans =
```

```
25.9136 -30.4033i
```

Input Arguments

X — Cosecant of angle

scalar value | vector | matrix | N-D array

Cosecant of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `acscd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

See Also

`acsc` | `csc` | `cscd`

Introduced before R2006a

acsch

Inverse hyperbolic cosecant

Syntax

```
Y = acsch(X)
```

Description

`Y = acsch(X)` returns the inverse hyperbolic cosecant for each element of `X`.

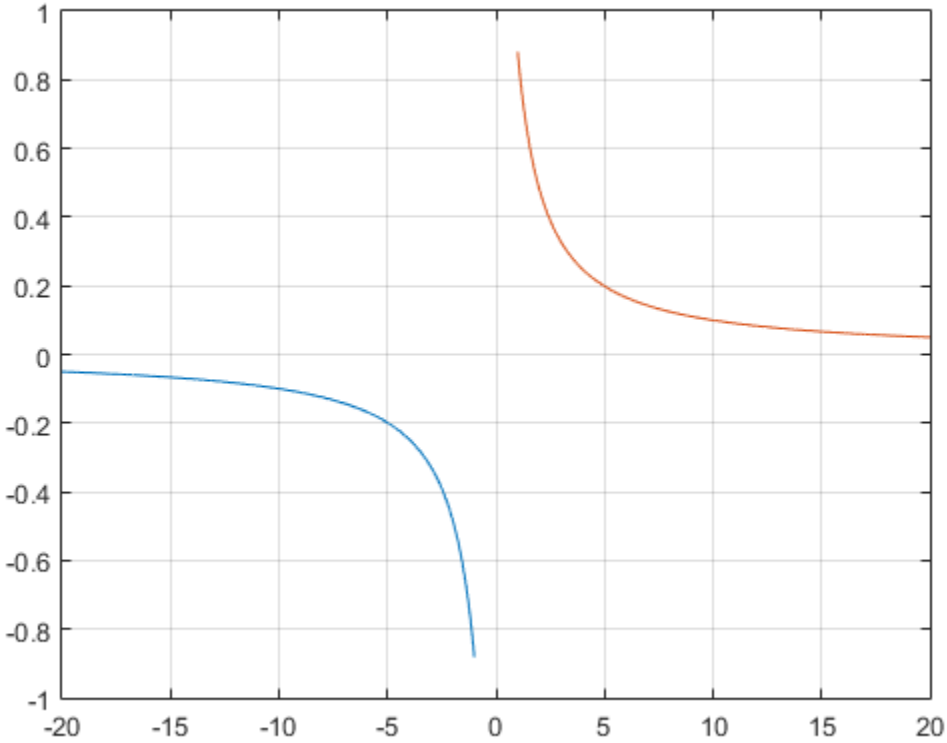
The `acsch` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph of Inverse Hyperbolic Cosecant Function

Graph the inverse hyperbolic cosecant function over the domains $-20 \le x \le -1$ and $1 \le x \le 20$.

```
x1 = -20:0.01:-1;
x2 = 1:0.01:20;
plot(x1,acsch(x1),x2,acsch(x2)), grid on
```



More About

Inverse Hyperbolic Cosecant

The inverse hyperbolic cosecant can be defined as

$$\operatorname{csch}^{-1}(z) = \sinh^{-1}\left(\frac{1}{z}\right).$$

See Also

`acsc` | `csch` | `asinh` | `acosh`

Introduced before R2006a

actxcontrol

Create Microsoft ActiveX control in figure window

Syntax

```
h = actxcontrol('progid')
h = actxcontrol('progid','param1',value1,...)
h = actxcontrol('progid',position)
h = actxcontrol('progid', position, fig_handle)
h = actxcontrol('progid',position,fig_handle,event_handler)
h =
actxcontrol('progid',position,fig_handle,event_handler,'filename')
```

Description

`h = actxcontrol('progid')` creates an ActiveX® control in a figure window. The programmatic identifier (`progid`) for the control determines the type of control created. (See the documentation provided by the control vendor to get this string.) The returned object, `h`, represents the default interface for the control.

You cannot use an ActiveX server for the `progid` because MATLAB software cannot insert ActiveX servers in a figure. See `actxserver` for use with ActiveX servers.

`h = actxcontrol('progid','param1',value1,...)` creates an ActiveX control using the optional parameter name/value pairs. Parameter names include:

- `position` — MATLAB position vector specifying the position of the control. The format is [left, bottom, width, height] using pixel units.
- `parent` — Handle to parent figure, model, or Command Window.
- `callback` — Name of event handler. To use the same handler for all events, specify a single name. To handle specific events, specify a cell array of event name/event handler pairs.
- `filename` — Sets the initial conditions to the previously saved control.
- `licensekey` — License key to create licensed ActiveX controls that require design-time licenses. See “Deploy ActiveX Controls Requiring Run-Time Licenses” for information on how to use controls that require run-time licenses.

One possible format is:

```
h = actxcontrol('myProgid','newPosition',[0 0 200 200],...
 'myFigHandle',gcf,...
 'myCallback',{'Click' 'myClickHandler';...
 'DblClick' 'myDblClickHandler';...
 'MouseDown' 'myMouseDownHandler'});
```

The following syntaxes are deprecated and will not become obsolete. They are included for reference, but the previous syntaxes are preferred.

`h = actxcontrol('progid',position)` creates an ActiveX control having the location and size specified in the vector, `position`. The format of this vector is:

```
[x y width height]
```

The first two elements of the vector determine where the control is placed in the figure window, with `x` and `y` being offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control. The last two elements, `width` and `height`, determine the size of the control itself.

The default `position` vector is `[20 20 60 60]`.

`h = actxcontrol('progid', position, fig_handle)` creates an ActiveX control at the specified `position` in an existing figure window. The Handle Graphics® handle, `fig_handle`, identifies this window.

The `gcf` command returns the current figure handle.

Note If the figure window designated by `fig_handle` is invisible, the control is invisible. If you want the control you are creating to be invisible, use the handle of an invisible figure window.

`h = actxcontrol('progid',position,fig_handle,event_handler)` creates an ActiveX control that responds to events. Controls respond to events by invoking a MATLAB function whenever an event (such as clicking a mouse button) is fired. The `event_handler` argument identifies one or more functions to be used in handling events. For more information, see “Specifying Event Handlers” on page 1-125.

```
h =
actxcontrol('progid',position,fig_handle,event_handler,'filename')
```

creates an ActiveX control with the first four arguments, and sets its initial state to that of a previously saved control. MATLAB loads the initial state from the file specified in the string `filename`.

If you do not want to specify an `event_handler`, you can use an empty string (`''`) as the fourth argument.

The `progid` argument must match the `progid` of the saved control.

Specifying Event Handlers

There is more than one valid format for the `event_handler` argument. Use this argument to specify one of the following:

- A different event handler routine for each event supported by the control
- One common routine to handle selected events
- One common routine to handle all events

In the first case, use a cell array for the `event_handler` argument, with each row of the array specifying an event and handler pair:

```
{'event' 'eventhandler'; 'event2' 'eventhandler2'; ...}
```

`event` is either a string containing the event name or a numeric event identifier (see Example 2). `eventhandler` is a string identifying the function you want the control to use in handling the event. Include only those events that you want enabled.

In the second case, use the same cell array syntax described, but specify the same `eventhandler` for each event. Again, include only those events that you want enabled.

In the third case, make `event_handler` a string (instead of a cell array) that contains the name of the one function that is to handle all events for the control.

There is no limit to the number of event and handler pairs you can specify in the `event_handler` cell array. However, if you register the same event name to the same callback handler multiple times, MATLAB executes the event only once.

Event handler functions accept a variable number of arguments.

Strings used in the `event_handler` argument are not case-sensitive.

Note Although using a single handler for all events might be easier in some cases, specifying an individual handler for each event creates more efficient code, resulting in better performance.

Examples

Handling Events

The `event_handler` argument specifies how you want the control to handle any events that occur. The control can handle all events with one common handler function, selected events with a common handler function, or each type of event can be handled by a separate function.

This command creates an `mwsamp` control that uses one event handler, `sampev`, to respond to all events:

```
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],...
   gcf,'sampev');
```

The next command also uses a common event handler, but will only invoke the handler when selected events, `Click` and `DblClick` are fired:

```
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],...
   gcf,{'Click' 'sampev'; 'DblClick' 'sampev'});
```

This command assigns a different handler routine to each event. For example, `Click` is an event, and `myclick` is the routine that executes whenever a `Click` event is fired:

```
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],...
   gcf,{'Click', 'myclick'; 'DblClick' 'my2click'; ...
   'MouseDown' 'mymoused'});
```

The next command does the same thing, but specifies the events using numeric event identifiers:

```
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],...
gcf,{-600, 'myclick'; -601 'my2click'; -605 'mymoused'});
```

For examples of event handler functions and how to register them with MATLAB, see "Sample Event Handlers".

More About

Tips

If the control implements any custom interfaces, use the `interfaces` function to list them, and the `invoke` function to get a handle to a selected interface.

When you no longer need the control, call `release` to release the interface and free memory and other resources used by the interface. Releasing the interface does not delete the control itself. To release the interface, use the `delete` function.

For more information on handling control events, see Writing Event Handlers.

For an example event handler, see the file `sampev.m` in the `toolbox\matlab\winfun\comcli` folder.

COM functions are available on Microsoft® Windows® systems only.

Note If you encounter problems creating Microsoft Forms 2.0 controls in MATLAB or other non-VBA container applications, see "Use Microsoft Forms 2.0 Controls".

See Also

`actxserver` | `delete (COM)` | `save (COM)` | `release` | `load (COM)` | `interfaces`

Introduced before R2006a

actxcontrollist

List currently installed Microsoft ActiveX controls

Syntax

```
info = actxcontrollist
```

Description

`info = actxcontrollist` returns a list of controls in `info`, a 1-by-3 cell array containing the name, programmatic identifier (ProgID), and file name for the control. Each control has one row, which MATLAB software sorts by file name.

COM functions are available on Microsoft Windows systems only.

Examples

Show information for two controls:

```
list = actxcontrollist;
for k = 1:2
  sprintf(' Name = %s\n ProgID = %s\n File = %s\n', list{k,:})
end
```

```
ans =
  Name = Calendar Control 11.0
  ProgID = MSCAL.Calendar.7
  File = C:\Program Files\MSOffice\OFFICE11\MSCAL.OCX

ans =
  Name = CTreeView Control
  ProgID = CTREEVIEW.CTreeViewCtrl.1
  File = C:\WINNT\system32\dmocx.dll
```

MATLAB displays controls specific to your system.

See Also

`actxcontrolselect` | `actxcontrol`

Introduced before R2006a

actxcontrolselect

Create Microsoft ActiveX control from UI

Syntax

```
h = actxcontrolselect
[h, info] = actxcontrolselect
```

Description

`h = actxcontrolselect` displays a UI listing all ActiveX controls installed on the system and creates the one you select from the list. Returns handle `h` for the object. Use the handle to identify this control when calling MATLAB COM functions.

`[h, info] = actxcontrolselect` returns a 1-by-3 cell array `info` containing the name, programmatic identifier (ProgID), and file name for the control.

COM functions are available on Microsoft Windows systems only.

More About

- "Creating Control Objects Using a UI"

See Also

`actxcontrollist` | `actxcontrol`

Introduced before R2006a

actxGetRunningServer

Handle to running instance of Automation server

Syntax

```
h = actxGetRunningServer('progid')
```

Description

`h = actxGetRunningServer('progid')` gets a reference to a running instance of the OLE Automation server. `progid` is the programmatic identifier of the Automation server object and `h` is the handle to the default interface of the server object.

The function returns an error if the server specified by `progid` is not currently running or if the server object is not registered. When multiple instances of the Automation server are running, the operating system controls the behavior of this function.

COM functions are available on Microsoft Windows systems only.

Examples

Get a handle to the Microsoft Excel® program:

```
h = actxGetRunningServer('Excel.Application')
```

More About

- "MATLAB COM Automation Server Interface"

See Also

`actxcontrol` | `actxserver`

Introduced in R2007a

actxserver

Create COM server

Syntax

```
h = actxserver('progid')
h = actxserver('progid','machine','machineName')
h = actxserver('progid','interface','interfaceName')
h =
actxserver('progid','machine','machineName','interface','interfaceName')
h = actxserver('progid',machine)
```

Description

`h = actxserver('progid')` creates a local OLE Automation server, where `progid` is the programmatic identifier of an OLE-compliant COM server, and `h` is the handle of the server's default interface.

Get `progid` from the control or server vendor's documentation. To see the `progid` values for MATLAB software, refer to “Programmatic Identifiers”.

`h = actxserver('progid','machine','machineName')` creates an OLE Automation server on a remote machine, where `machineName` is a string specifying the name of the machine on which to start the server.

`h = actxserver('progid','interface','interfaceName')` creates a Custom interface server, where `interfaceName` is a string specifying the interface name of the COM object. Values for `interfaceName` are

- `IUnknown` — Use the IUnknown interface.
- The Custom interface name

You must know the name of the interface and have the server vendor's documentation in order to use the `interfaceName` value. For information about custom COM servers and interfaces, see “COM Server Types”.

Note: The MATLAB COM Interface does not support invoking functions with optional parameters.

`h = actxserver('progid','machine','machineName','interface','interfaceName')` creates a Custom interface server on a remote machine.

The following syntaxes are deprecated and will not become obsolete. They are included for reference, but the syntaxes described earlier are preferred:

`h = actxserver('progid',machine)` creates a COM server running on the remote system named by the `machine` argument. This can be an IP address or a DNS name. Use this syntax only in environments that support Distributed Component Object Model (for more information, see "Using MATLAB Application as DCOM Server").

COM functions are available on Microsoft Windows systems only.

Examples

Microsoft Excel Workbook Example

This example creates an OLE Automation server, Excel version 9.0, and manipulates a workbook in the application.

Create a COM server running Microsoft Excel.

```
e = actxserver('Excel.Application')
```

```
e =
    COM.Excel.application
```

Make the Excel frame window visible.

```
e.Visible = 1;
```

Use the `get` method on the Excel object `e` to list all properties of the application.

```
get(e)
```

Create an interface `eWorkBooks`.

```
eWorkbooks = e.Workbooks
```

```
eWorkbooks =
   Interface.Microsoft_Excel_9.0_Object_Library.Workbooks
```

List all methods for that interface.

```
invoke(eWorkbooks)
```

```
       Add: 'handle Add(handle, [Optional]Variant)'
     Close: 'void Close(handle)'
      Item: 'handle Item(handle, Variant)'
      Open: 'handle Open(handle, string, [Optional]Variant)'
  OpenText: 'void OpenText(handle, string, [Optional]Variant)'
          .
          .
          .
```

Add a new workbook `w`, also creating a new interface.

```
w = Add(eWorkbooks)
```

```
w =
   Interface.Microsoft_Excel_9.0_Object_Library._Workbook
```

Close Excel and delete the server.

```
Quit(e);
delete(e);
```

More About

Tips

For components implemented in a dynamic link library (DLL), `actxserver` creates an in-process server. For components implemented as an executable (EXE), `actxserver` creates an out-of-process server. Out-of-process servers can be created either on the client system or on any other system on a network that supports DCOM.

If the control implements any Custom interfaces, use the `interfaces` function to list them, and the `invoke` function to get a handle to a selected interface.

You can register events for COM servers.

See Also

`actxcontrol` | `delete (COM)` | `save (COM)` | `actxGetRunningServer` | `release` | `load (COM)` | `interfaces` | `invoke`

Related Examples

- "Read Excel Spreadsheet Data"
- "Use MATLAB Application as Automation Client"

Introduced before R2006a

add

Add single key-value pair to KeyValueStore

Syntax

```
add(KVStore,key,value)
```

Description

`add(KVStore,key,value)` adds a single key-value pair to `KVStore`, which is a `KeyValueStore` created during `mapreduce` execution. Use `add` in a map or reduce function written for use with `mapreduce` to store intermediate or final key-value pair information.

Input Arguments

`KVStore` — Key-value pair storage object

`KeyValueStore` object

Key-value pair storage object, specified as a `KeyValueStore` object. The `mapreduce` function automatically creates the `KeyValueStore` object during execution:

- In the map function, the name of the intermediate `KeyValueStore` object is the third input argument to the map function, `myMapper(data, info, intermKVStore)`. Use that same variable name to add intermediate key-value pairs with `add` or `addmulti` in the map function.
- In the reduce function, the name of the final `KeyValueStore` object is the third input argument to the reduce function, `myReducer(intermKey, intermValIter, outKVStore)`. Use that same variable name to add final key-value pairs with `add` or `addmulti` in the reduce function.

For more information, see Using KeyValueStore Objects.

`key` — Key

numeric scalar | string

Key, specified as a numeric scalar or string.

All of the keys added by the map function must have the same class. The keys added by the reduce function also must have the same class, but that class can differ from the class of the keys added by the map function.

Numeric keys cannot be `NaN`, complex, logical, or sparse.

Example: `add(intermKVStore,'Sum',sum(X))` adds a key-value pair to an intermediate `KeyValueStore` object (named `intermKVStore`) in a map function.

Example: `add(outKVStore,'Stats',[mean(X) max(X) min(X) var(X) std(X)])` adds a key-value pair to a final `KeyValueStore` object (named `outKVStore`) in a reduce function.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char`

`value` — Value

any MATLAB object

Value, specified as any MATLAB object. This includes all valid MATLAB data types.

The `OutputType` argument of `mapreduce` affects the type of values that the reduce function can add:

- If the `OutputType` is `'Binary'` (the default), then a value added by the reduce function can be any MATLAB object.
- If the `OutputType` is `'TabularText'`, then a value added by the reduce function can be a numeric scalar or string when using the `add` function. Additionally, you can use the `addmulti` function to add multiple values with a numeric vector, cell vector of strings, or cell vector of numeric scalars. In each case, the numeric values cannot be `NaN`, complex, logical, or sparse.

Note: The above key-value pair requirements may differ when using other products with mapreduce. See the documentation for the appropriate product to get product-specific key-value pair requirements.

Example: `add(intermKVStore,'Sum',sum(X))` specifies a single scalar value to pair with a key.

Example: `add(outKVStore,'Stats',[mean(X) max(X) min(X) var(X) std(X)])` specifies a numeric array as the value to pair with a key.

More About

Tips

- Avoid using `add` in a loop, as it can negatively affect `mapreduce` execution time. Instead, use cell arrays to collect multiple values (using vectorized operations if possible) and use a single call to `addmulti`.

- Using KeyValueStore Objects
- "Build Effective Algorithms with MapReduce"

See Also

`addmulti`

Introduced in R2014b

matlab.apputil.create

Create or modify app project file for packaging app into `.mlappinstall` file using interactive dialog box

Syntax

```
matlab.apputil.create
matlab.apputil.create(prjfile)
```

Description

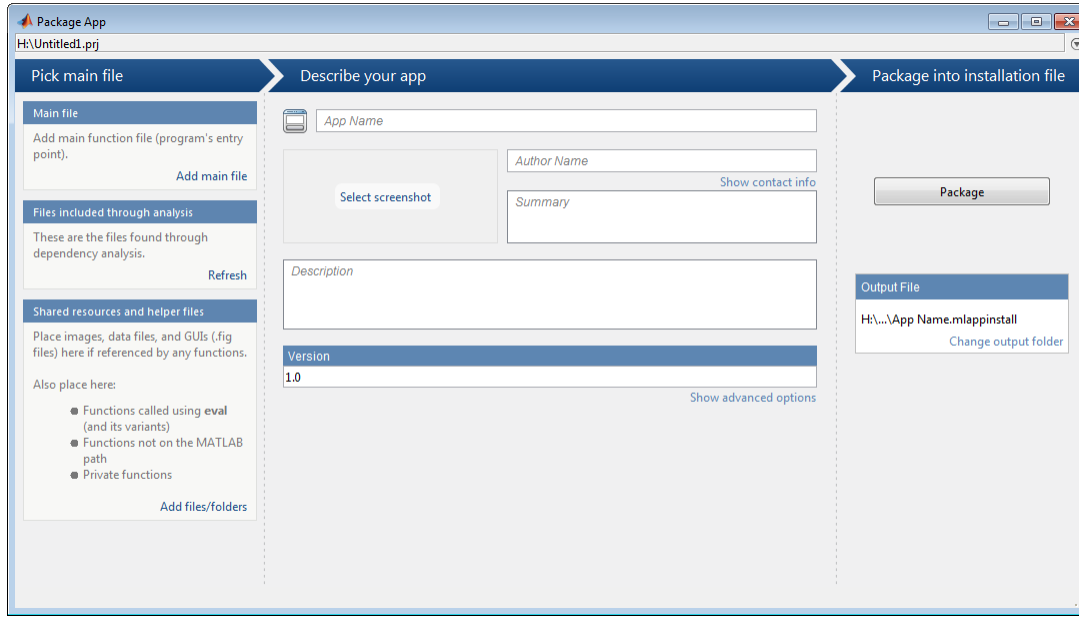
`matlab.apputil.create` opens the Package App dialog box that steps you through the process of creating an `.mlappinstall` file.

`matlab.apputil.create(prjfile)` loads the specified `.prj` file and populates the Package App dialog box with the information from the specified project file. Use this option if you need to update an existing app.

Examples

Open Dialog Box for Creating an App Package

```
matlab.apputil.create
```



Minimally, add a main file and specify an app name. MATLAB creates and continuously saves a `.prj` file, regardless of whether you click **Package**. However, MATLAB does not create a `.mlappinstall` file if you do not click **Package**.

Update Existing App Package

Assume you have an existing project file, `myapp.prj`. You want to add a file and update the description.

Open the Package App dialog box, specifying the previously created `.prj` file:

```
matlab.apputil.create('myapp.prj')
```

The dialog box opens populated with the data you previously specified for `myapp`. Adjust the information in the dialog box, as needed.

Input Arguments

`prjfile` — Full or partial path to the `.prj` file

string

Full or partial path to the `.prj` file you created previously with the Package App dialog box, specified as a string

Example: `'C:\myapp.prj'`

More About

- "MATLAB App Installer File — mlappinstall"

See Also

`matlab.apputil.package`

matlab.apputil.getInstalledAppInfo

List installed app information

Syntax

```
matlab.apputil.getInstalledAppInfo
```

```
appinfo = matlab.apputil.getInstalledAppInfo
```

Description

`matlab.apputil.getInstalledAppInfo` displays the ID and name of all installed custom apps. It does not display this information for apps packaged with MathWorks® products.

`appinfo = matlab.apputil.getInstalledAppInfo` returns structure to `appinfo`, which includes the status, ID, location, and name of all installed custom apps. It does not return this information for apps packaged with MathWorks products.

Examples

Display Installed Apps Information in the Command Window

Assume you installed two apps, `LinePlotter` and `PlotRandNumbers`. Display the app information in the Command Window.

```
 matlab.apputil.getInstalledAppInfo
```

```
ID                    Name
------------------    --------------
LinePlotterAPP        LinePlotter
PlotRandNumbersAPP    PlotRandNumbers
```

Store Installed App Information in a Variable

Assume you installed an app, `ColorPalette`. Get the app information and store it in a variable, `myappinfo`.

```
myappinfo = matlab.apputil.getInstalledAppInfo;
```

Store Installed App Information in a Variable and Display IDs

Assume you installed two apps, `LinePlotter` and `PlotRandNumbers`. Get and store the app information for both installed apps in a variable, `myappinfo`. Then, get the `id` for each app.

```
myappinfo = matlab.apputil.getInstalledAppInfo
```

```
myappinfo =

1x2 struct array with fields:
    id
    name
    status
    location
```

Get the id of each installed app:

```
appids={myappinfo.id}
```

```
appids =

    'LinePlotterAPP'    'PlotRandNumbersAPP'
```

Output Arguments

`appinfo` — Information about installed apps

structure array

Information about the installed app, returned as a structure array, with one element for each installed app. Each element of the structure array has the following fields:

`status` — Installation status

`'installed'`

Status of the installation, returned as the string `'installed'`.

`id` — Unique identifier for the installed app

string

Unique identifier for the installed app, returned as a string

The ID is for use when running or uninstalling the app programmatically.

`location` — Folder where the app is installed
string

Folder where the app is installed, returned as a string

`name` — Name of the installed app
string

Name of the installed app as it appears in the apps gallery, returned as a string

More About

- "MATLAB App Installer File — mlappinstall"

See Also

`matlab.apputil.install` | `matlab.apputil.run` | `matlab.apputil.uninstall`

matlab.apputil.install

Install app from a `.mlappinstall` file

Syntax

```
appinfo = matlab.apputil.install(appfile)
```

Description

`appinfo = matlab.apputil.install(appfile)` installs the specified app file and returns information about the app.

Examples

Install App and Display Information About the Installation

Assume you have downloaded an app from File Exchange named `EmployeeData`. Install it and return information about the installation to the variable `appinfo`. Later, if you decide to deinstall the app programmatically, you have the app id required to do so.

```
appinfo = matlab.apputil.install...
   ('C:\myguis\myapps\EmployeeData.mlappinstall')
```

```
appinfo =

          id: 'EmployeeDataApp'
        name: 'EmployeeData'
      status: 'installed'
    location: 'C:\myguis\myapps\EmployeeData.mlappinstall'
```

Input Arguments

`appfile` — Full or partial path to `.mlappinstall` file

string

Full or partial path of the app file you want to install, specified as a string.

Example: `'C:\myguis\myapps\myapp.mlappinstall'`

Output Arguments

`appinfo` — Information about installed app

structure

Information about the installed app, returned as a structure with the fields:

`status` — Installation status

`'installed' | 'updated'`

Installation status, returned as a string:

- `'installed'` — New app is installed.
- `'updated'` — Previously installed app is updated.

`id` — Unique identifier

string

Unique identifier for the installed app, returned as a string

The ID is for use when running or uninstalling the app programmatically.

`location` — Folder where app is installed

string

Folder where app is installed, returned as a string

`name` — Name of installed app

string

Name of installed app as it appears in the apps gallery, returned as a string

More About

- "MATLAB App Installer File — mlappinstall"

See Also

`matlab.apputil.getInstalledAppInfo` | `matlab.apputil.package` | `matlab.apputil.uninstall`

matlab.apputil.package

Package app files into `.mlappinstall` file

Syntax

```
matlab.apputil.package(prjfile)
```

Description

`matlab.apputil.package(prjfile)` creates a `.mlappinstall` file based on the information in the specified prjfile.

Examples

Create mlappinstall File for Previously Created Project File

Assume you previously created `myprjfile.prj` using `matlab.apputil.create`. The following command creates the corresponding `.mlappinstall` file.

```
matlab.apputil.package('myprjfile.prj')
```

Input Arguments

`prjfile` — Full or partial path to app project (`.prj`) file

string

Full or partial path to app project (`.prj`) file, specified as a string.

Example: `'plotdata.prj'`

More About

Tips

- To create a `.prj` file, use `matlab.apputil.create`.

See Also

`matlab.apputil.create` | `matlab.apputil.install` | `matlab.apputil.run`

matlab.apputil.run

Run app programmatically

Syntax

```
matlab.apputil.run(appid)
```

Description

`matlab.apputil.run(appid)` runs the custom app specified by the unique identifier, appid.

Examples

Run Previously Installed App

Assume you installed two apps, `PlotData` and `setslider`. Run `PlotData` programmatically, using its ID.

Get IDs of all installed apps.

```
matlab.apputil.getInstalledAppInfo
```

```
ID                    Name
------------------    ----------------
setsliderAPP          setslider
PlotDataAPP           PlotData
```

Run `PlotData`.

```
matlab.apputil.run('PlotDataAPP')
```

Input Arguments

`appid` — ID of custom app

string

ID of custom app you want to run, specified as a string.

Example: `'DataExplorationAPP'`

More About

Tips

- The ID of a custom app is returned when you install it. You can use `matlab.apputil.getInstalledAppInfo` to get the ID after you have installed an app.
- When a custom app runs, MATLAB adds any folders it needs to have added to the path, as identified when the app was packaged. When the app exits, MATLAB removes those folders from the path.
- You can run multiple, different custom apps concurrently. However, you cannot run two instances of the same app concurrently.

See Also

`matlab.apputil.create` | `matlab.apputil.getInstalledAppInfo` | `matlab.apputil.install`

matlab.apputil.uninstall

Uninstall app

Syntax

```
matlab.apputil.uninstall(appid)
```

Description

`matlab.apputil.uninstall(appid)` removes the app specified by the unique identifier, appid. MATLAB removes all files corresponding to the app and removes the app from the app gallery.

Examples

Uninstall App

Assume you previously installed two apps, `setslider` and `simplegui`. Get the IDs of all installed apps, and then use the ID for `simplegui` to uninstall it.

View the IDs of all apps

```
matlab.apputil.getInstalledAppInfo
```

```
ID                    Name
--------------------  ----------------
setsliderAPP          setslider
simpleguiAPP          simplegui
```

Uninstall the `simplegui` app.

```
matlab.apputil.uninstall('simpleguiAPP')
```

Confirm the app was removed, by running `matlab.apputil.getInstalledAppInfo` again.

```
matlab.apputil.getInstalledAppInfo
```

```
ID                    Name
--------------------  ----------------
setsliderAPP          setslider
```

Input Arguments

`appid` — ID of app

string

ID of app to be uninstalled, specified as a string.

Example: `'DataExplorationAPP'`

More About

Tips

- To determine the appid of an installed app, preserve the value returned when you install the app programmatically with `matlab.apputil.install`, or use `matlab.apputil.getInstalledAppInfo`.

See Also

`matlab.apputil.getInstalledAppInfo` | `matlab.apputil.install`

addevent

Add event to `timeseries` object

Syntax

```
ts = addevent(ts,e)
ts = addevent(ts,Name,Time)
```

Description

`ts = addevent(ts,e)` adds one or more `tsdata.event` objects, `e`, to the `timeseries` object `ts`. `e` is either a single `tsdata.event` object or an array of `tsdata.event` objects.

`ts = addevent(ts,Name,Time)` constructs one or more `tsdata.event` objects and adds them to the `Events` property of `ts`. `Name` is a cell array of event name strings. `Time` is a cell array of event times.

Examples

Create a time-series object and add an event to this object.

```
%% Import the sample data
load count.dat

%% Create time-series object
count1=timeseries(count(:,1),1:24,'name', 'data');

%% Modify the time units to be 'hours' ('seconds' is default)
count1.TimeInfo.Units = 'hours';

%% Construct and add the first event at 8 AM
e1 = tsdata.event('AMCommute',8);

%% Specify the time units of the time
e1.Units = 'hours';
```

View the properties (`EventData`, `Name`, `Time`, `Units`, and `StartDate`) of the event object.

```
get(e1)
```

MATLAB software responds with

```
    EventData: []
         Name: 'AMCommute'
         Time: 8
        Units: 'hours'
    StartDate: ''
%% Add the event to count1
count1 = addevent(count1,e1);
```

An alternative syntax for adding two events to the time series `count1` is as follows:

```
count1 = addevent(count1,{'AMCommute' 'PMCommute'},{8 18})
```

See Also

`timeseries` | `tsdata.event`

Introduced before R2006a

addmulti

Add multiple key-value pairs to KeyValueStore

Syntax

```
addmulti(KVStore,keys,values)
```

Description

`addmulti(KVStore,keys,values)` adds multiple key-value pairs to `KVStore`, which is a `KeyValueStore` created during `mapreduce` execution. Use `addmulti` in a map or reduce function written for use with `mapreduce` to store intermediate or final key-value pair information.

Input Arguments

`KVStore` — Key-value pair storage object

`KeyValueStore` object

Key-value pair storage object, specified as a `KeyValueStore` object. The `mapreduce` function automatically creates the `KeyValueStore` object during execution:

- In the map function, the name of the intermediate `KeyValueStore` object is the third input argument to the map function, `myMapper(data, info, intermKVStore)`. Use that same variable name to add intermediate key-value pairs with `add` or `addmulti` in the map function.
- In the reduce function, the name of the final `KeyValueStore` object is the third input argument to the reduce function, `myReducer(intermKey, intermValIter, outKVStore)`. Use that same variable name to add final key-value pairs with `add` or `addmulti` in the reduce function.

For more information, see Using KeyValueStore Objects.

`keys` — Keys

numeric scalar | numeric vector | string | cell vector of strings | cell vector of numeric scalars

Keys, specified as a numeric scalar, numeric vector, string, cell vector of strings, or cell vector of numeric scalars. If the keys are a numeric vector or cell vector, then each entry specifies a different key.

All of the keys added by the map function must have the same class. The keys added by the reduce function must also have the same class, but that class can differ from the class of the keys added by the map function.

Numeric keys cannot be `NaN`, complex, logical, or sparse.

Example: `addmulti(intermKVStore,{'Sum'; 'Count'; 'Variance'},{sum(X); numel(X); var(X)})` adds three key-value pairs to an intermediate `KeyValueStore` object (named `intermKVStore`) using a cell vector of strings to specify the keys.

Example: `addmulti(intermKVStore,[1 2 3 4],{sum(X); mean(X); max(X); min(X)})` adds four key-value pairs to an intermediate `KeyValueStore` object using a numeric vector to specify the keys.

Example: `addmulti(outKVStore,'Stats',{[mean(X) max(X) min(X) var(X) std(X)]})` adds a single key-value pair to a final `KeyValueStore` object (named `outKVStore`) using a string as the key.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char`

`values` — Values

cell array

Values, specified as a cell array. Each entry in the cell array specifies the value in a key-value pair, so `numel(values)` must be equal to the number of keys. The entries in the cell array can be any MATLAB object, including all valid MATLAB data types.

The `OutputType` argument of `mapreduce` affects the type of values that the reduce function can add:

- If the `OutputType` is `'Binary'` (the default), then a value added by the reduce function can be any MATLAB object.
- If the `OutputType` is `'TabularText'`, then a value added by the reduce function can be a numeric scalar or string when using the `add` function. Additionally, you can use the `addmulti` function to add multiple values with a numeric vector, cell vector of strings, or cell vector of numeric scalars. In each case, the numeric values cannot be `NaN`, complex, logical, or sparse.

Note: The above key-value pair requirements may differ when using other products with mapreduce. See the documentation for the appropriate product to get product-specific key-value pair requirements.

Example: `addmulti(intermKVStore,{'Sum'; 'Count'; 'Variance'},{sum(X); numel(X); var(X)})` adds three key-value pairs to an intermediate `KeyValueStore` object named `intermKVStore`.

Example: `addmulti(intermKVStore,[1 2 3 4],{sum(X); mean(X); max(X); min(X)})` adds four key-value pairs to an intermediate `KeyValueStore` object using a cell vector.

Example: `addmulti(outKVStore,'Stats',{[mean(X) max(X) min(X) var(X) std(X)]})` adds a single key-value pair to a final `KeyValueStore` object named `outKVStore`.

Example: `addmulti(outKVStore,{'Distance' 'Time'},{table.Distance table.Time})` adds two key-value pairs using variables in a table to specify the values.

More About

Tips

- Avoid using `add` in a loop, as it can negatively affect `mapreduce` execution time. Instead, use cell arrays to collect multiple values (using vectorized operations if possible) and use a single call to `addmulti`.

- Using KeyValueStore Objects
- "Build Effective Algorithms with MapReduce"

See Also

`add`

Introduced in R2014b

audioinfo

Information about audio file

Syntax

```
info = audioinfo(filename)
```

Description

`info = audioinfo(filename)` returns information about the contents of the audio file specified by `filename`.

Examples

Get Information About Audio File

Create a WAVE file from the example file `handel.mat`, and get information about the file.

Create a WAVE (.wav) file in the current folder.

```
load handel.mat
filename = 'handel.wav';
audiowrite(filename,y,Fs);
clear y Fs
```

Use `audioinfo` to return information about the WAVE file.

```
info = audioinfo(filename)
```

```
info =
             Filename: 'S:\handel.wav'
    CompressionMethod: 'Uncompressed'
          NumChannels: 1
           SampleRate: 8192
         TotalSamples: 73113
             Duration: 8.9249
```

```
          Title: []
        Comment: []
         Artist: []
  BitsPerSample: 16
```

Input Arguments

`filename` — Name of file

string

Name of file, specified as a string. If a path is specified, it can be absolute, relative, or partial.

Example: `'myFile.mp3'`

Example: `'../myFile.mp3'`

Example: `'C:\temp\myFile.mp3'`

`audioinfo` supports the following file formats.

Platform Support	File Format
All platforms	WAVE (`.wav`)
	OGG (`.ogg`)
	FLAC (`.flac`)
	AU (`.au`)
Windows 7 (or later), Macintosh, and Linux®	MP3 (`.mp3`)
	MPEG-4 AAC (`.m4a`, `.mp4`)

On Windows 7 platforms (or later), `audioinfo` might also return information about the contents of any files supported by Windows Media® Foundation.

On Linux platforms, `audioinfo` might also return information about the contents of any files supported by GStreamer.

`audioinfo` can extract audio metadata from MPEG-4 (`.mp4`, `.m4v`) video files on Windows 7 or later, Mac OS X 10.7 Lion or higher, and Linux, and from Windows Media Video (`.wmv`) and AVI (`.avi`) files on Windows 7 (or later) and Linux platforms.

Output Arguments

`info` — Information about audio file

structure

Information about audio file, returned as a structure. `info` can contain the following fields.

Field Name	Description	Data Type
`Filename`	Filename including the absolute path to the file and the file extension.	`string`
`CompressionMethod`	Compression method used.	`string`
`NumChannels`	Number of audio channels encoded in the audio file.	`double`
`SampleRate`	Sample rate of the audio data in the file, in hertz.	`double`
`TotalSamples`	Total number of audio samples in the file.	`double`
`Duration`	Duration of the file, in seconds.	`double`
`BitsPerSample`	Number of bits per sample encoded in the audio file. Only valid for WAVE (`.wav`) and FLAC (`.flac`) files.	`double`
`BitRate`	Number of kilobits per second (kbit/s) used for compressed audio files. Only valid for MP3 (`.mp3`) and MPEG-4 Audio (`.m4a`, `.mp4`) files.	`double`
`Title`	Value of `'Title'`, if any.	`string`
`Artist`	Value of `'Artist'`, if any.	`string`
`Comment`	Value of `'Comment'`, if any.	`string`

Note: The `BitRate` property returns the actual bit rate on Mac platforms, and not the encoded bit rate. This means that bit rate values might be lower than specified at the time of the encoding, depending on the source data.

Note: On Mac platforms, `audioinfo` returns metadata from `.m4a` and `.mp4` files only on Mac OS X 10.7 Lion or higher. Previous versions of Mac OS X will not read the `'Title'`, `'Author'`, or `'Comment'` fields.

Limitations

- For MP3 and MPEG-4 AAC audio files on Windows 7 or later and Linux platforms, `audioinfo` might report fewer samples than expected. On Linux platforms, this is due to a limitation in the underlying GStreamer framework.
- On Linux platforms, `audioinfo` interprets single channel data in MPEG-4 AAC files as stereo data.

See Also

`audioread` | `audiowrite`

Introduced in R2012b

audioread

Read audio file

Syntax

```
[y,Fs] = audioread(filename)
[y,Fs] = audioread(filename,samples)

[y,Fs] = audioread( ___ ,dataType)
```

Description

`[y,Fs] = audioread(filename)` reads data from the file named `filename`, and returns sampled data,`y`, and a sample rate for that data, `Fs`.

`[y,Fs] = audioread(filename,samples)` reads the selected range of audio samples in the file, where `samples` is a vector of the form `[start,finish]`.

`[y,Fs] = audioread( ___ ,dataType)` returns sampled data in the data range corresponding to the `dataType` of `'native'` or `'double'`, and can include any of the input arguments in previous syntaxes.

Examples

Read Complete Audio File

Create a WAVE file from the example file `handel.mat`, and read the file back into MATLAB.

Create a WAVE (`.wav`) file in the current folder.

```
load handel.mat

filename = 'handel.wav';
audiowrite(filename,y,Fs);
```

```
clear y Fs
```

Read the data back into MATLAB using `audioread`.

```
[y,Fs] = audioread('handel.wav');
```

Play the audio.

```
sound(y,Fs);
```

Read Portion of Audio File

Create a FLAC file from the example file `handel.mat`, and then read only the first 2 seconds.

Create a FLAC (`.flac`) file in the current folder.

```
load handel.mat

filename = 'handel.flac';
audiowrite(filename,y,Fs);
```

Read only the first 2 seconds.

```
samples = [1,2*Fs];
clear y Fs
[y,Fs] = audioread(filename,samples);
```

Play the samples.

```
sound(y,Fs);
```

Return Audio in Native Integer Format

Create a FLAC file and read the first 2 seconds according to the previous Example. Then, view the data type of the sampled data `y`.

```
whos y
```

```
  Name          Size              Bytes  Class     Attributes

  y         16384x1             131072  double
```

The data type of `y` is `double`.

Request audio data in the native format of the file, and then view the data type of the sampled data `y`.

```
[y,Fs] = audioread(filename,'native');
whos y
```

```
  Name          Size             Bytes  Class    Attributes

  y         16384x1              32768  int16
```

The data type of `y` is now `int16`.

Input Arguments

`filename` — Name of file to read

string

Name of file to read, specified as a string that includes the file extension. If a path is specified, it can be absolute, relative or partial.

Example: `'myFile.mp3'`

Example: `'../myFile.mp3'`

Example: `'C:\temp\myFile.mp3'`

`audioread` supports the following file formats.

Platform Support	File Format
All platforms	WAVE (`.wav`)
	OGG (`.ogg`)
	FLAC (`.flac`)
	AU (`.au`)
Windows 7 (or later), Macintosh, and Linux	MP3 (`.mp3`)
	MPEG-4 AAC (`.m4a`, `.mp4`)

On Windows platforms prior to Windows 7, `audioread` does not read WAVE files with MP3 encoded data.

On Windows 7 (or later) platforms, `audioread` might also read any files supported by Windows Media Foundation.

On Linux platforms, `audioread` might also read any files supported by GStreamer.

`audioread` can extract audio from MPEG-4 (`.mp4`, `.m4v`) video files on Windows 7 or later, Macintosh, and Linux, and from Windows Media Video (`.wmv`) and AVI (`.avi`) files on Windows 7 (or later) and Linux platforms.

`samples` — Audio samples to read

`[1,inf]` (default) | two-element vector of positive scalar integers

Audio samples to read, specified as a two-element vector of the form `[start,finish]`, where `start` and `finish` are the first and last samples to read, and are positive scalar integers.

- `start` must be less than or equal to `finish`.
- `start` and `finish` must be less than the number of audio samples in the file,
- You can use `inf` to indicate the last sample in the file.

Note: When reading a portion of some MP3 files on Windows 7 platforms, `audioread` might read a shifted range of samples. This is due to a limitation in the underlying Windows Media Foundation framework.

When reading a portion of MP3 and M4A files on Linux platforms, `audioread` might read a shifted range of samples. This is due to a limitation in the underlying GStreamer framework.

Example: `[1,100]`

Data Types: `double`

`dataType` — Data format of audio data, `y`

`'double'` (default) | `'native'`

Data format of audio data,`y`, specified as one of the following strings:

`'double'`	Double-precision normalized samples.

`'native'` Samples in the native format found in the file.

For compressed audio formats, such as MP3 and MPEG-4 AAC that do not store data in integer form, `'native'` defaults to `'single'`.

Output Arguments

y — Audio data

matrix

Audio data in the file, returned as an `m`-by-`n` matrix, where `m` is the number of audio samples read and `n` is the number of audio channels in the file.

- If you do not specify `dataType`, or `dataType` is `'double'`, then `y` is of type `double`, and matrix elements are normalized values between −1.0 and 1.0.
- If `dataType` is `'native'`, then `y` can be one of several MATLAB data types, depending on the file format and the `BitsPerSample` value of the input file. Call `audioinfo` to determine the `BitsPerSample` value of the file.

File Format	BitsPerSample	Data Type of y	Data Range of y
WAVE (`.wav`)	8	`uint8`	$0 \le y \le 255$
	16	`int16`	$-32768 \le y \le +32767$
	24	`int32`	$-2^{32} \le y \le 2^{32}-1$
	32	`int32`	$-2^{32} \le y \le 2^{32}-1$
	32	`single`	$-1.0 \le y \le +1.0$
	64	`double`	$-1.0 \le y \le +1.0$
FLAC (`.flac`)	8	`uint8`	$0 \le y \le 255$
	16	`int16`	$-32768 \le y \le +32767$
	24	`int32`	$-2^{32} \le y \le 2^{32}-1$
MP3 (`.mp3`), MPEG-4 AAC (`.m4a`, `.mp4`), OGG (`.ogg`), and certain compressed WAVE files	N/A	`single`	$-1.0 \le y \le +1.0$

Note: Where `y` is `single` or `double` and the `BitsPerSample` is 32 or 64, values in `y` might exceed −1.0 or +1.0.

`Fs` — Sample rate

positive scalar

Sample rate, in hertz, of audio data `y`, returned as a positive scalar.

Limitations

- For MP3 and MPEG-4 AAC audio files on Windows 7 or later and Linux platforms, `audioread` might read fewer samples than expected. On Windows 7 platforms, this is due to a limitation in the underlying Media Foundation framework. On Linux platforms, this is due to a limitation in the underlying GStreamer framework. If you require sample-accurate reading, work with WAV or FLAC files.
- On Linux platforms, `audioread` reads MPEG-4 AAC files that contain single-channel data as stereo data.

See Also

`audioinfo` | `audiowrite`

Introduced in R2012b

audiowrite

Write audio file

Syntax

```
audiowrite(filename,y,Fs)
audiowrite(filename,y,Fs,Name,Value)
```

Description

`audiowrite(filename,y,Fs)` writes a matrix of audio data, `y`, with sample rate `Fs` to a file called `filename`. The `filename` input also specifies the output file format. The output data type depends on the output file format and the data type of the audio data, `y`.

`audiowrite(filename,y,Fs,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Write an Audio File

Create a WAVE file from the example file `handel.mat`, and read the file back into MATLAB.

Write a WAVE (`.wav`) file in the current folder.

```
load handel.mat

filename = 'handel.wav';
audiowrite(filename,y,Fs);
clear y Fs
```

Read the data back into MATLAB using `audioread`.

```
[y,Fs] = audioread(filename);
```

Listen to the audio.

```
sound(y,Fs);
```

Specify Bits Per Sample and Metadata

Create a FLAC file from the example file `handel.mat` and specify the number of output bits per sample and a comment.

```
load handel.mat

filename = 'handel.flac';
audiowrite(filename,y,Fs,'BitsPerSample',24,...
'Comment','This is my new audio file.');
clear y Fs
```

View information about the new FLAC file.

```
info = audioinfo(filename)
```

```
info =

             Filename: 'S:\handel.flac'
    CompressionMethod: 'FLAC'
          NumChannels: 1
           SampleRate: 8192
         TotalSamples: 73113
             Duration: 8.9249
                Title: []
              Comment: 'This is my new audio file.'
               Artist: []
        BitsPerSample: 24
```

Input Arguments

`filename` — Name of file to write

string

Name of file to write, or the full path to the file, specified as a string that includes the file extension. If a path is specified, it can be absolute or relative. If you do not specify the path, then the destination directory is the current working directory.

`audiowrite` supports the following file formats.

Platform Support	File Format
All platforms	WAVE (`.wav`)
	OGG (`.ogg`)
	FLAC (`.flac`)
Windows and Mac	MPEG-4 AAC (`.m4a`, `.mp4`)

Example: `'myFile.m4a'`

Example: `'../myFile.m4a'`

Example: `'C:\temp\myFile.m4a'`

When writing AAC files on Windows, `audiowrite` pads the front and back of the output signal with extra samples of silence. The Windows AAC encoder also places a very sharp fade-in and fade-out on the audio. This results in audio with an increased number of samples after being written to disk.

y — Audio data to write

matrix

Audio data to write, specified as an `m`-by-`n` matrix, where `m` is the number of audio samples to write and `n` is the number of audio channels to write.

If either `m` or `n` is 1, then `audiowrite` assumes that this dimension specifies the number of audio channels, and the other dimension specifies the number of audio samples.

The maximum number of channels depends on the file format.

File Format	Maximum Number of Channels
WAVE (`.wav`)	256
OGG (`.ogg`)	255
FLAC (`.flac`)	8
MPEG-4 AAC (`.m4a`, `.mp4`)	2

The valid range for the data in `y` depends on the data type of `y`.

Data Type of y	Valid Range for y
`uint8`	$0 \le y \le 255$

Data Type of y	Valid Range for y
int16	-32768 ≤ y ≤ +32767
int32	-2^32 ≤ y ≤ 2^32–1
single	-1.0 ≤ y ≤ +1.0
double	-1.0 ≤ y ≤ +1.0

Data beyond the valid range is clipped.

If y is single or double, then audio data in y should be normalized to values in the range −1.0 and 1.0, inclusive.

Data Types: single | double | int16 | int32 | uint8

Fs — Sample rate
positive scalar

Sample rate, in hertz, of audio data y, specified as a positive scalar greater than 0. Values of Fs are truncated to integer boundaries. When writing to .m4a or .mp4 files on Windows platforms, audiowrite supports only samples rates of 44100 and 48000.

Example: 44100

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Title','Symphony No. 9','Artist','My Orchestra' instructs audiowrite to write an audio file with the title "Symphony No. 9" and the artist information "My Orchestra."

'BitsPerSample' — Number of output bits per sample
16 (default) | 8 | 24 | 32 | 64

Number of output bits per sample, specified as the comma-separated pair consisting of 'BitsPerSample' and a number.

Only available for WAVE (`.wav`) and FLAC (`.flac`) files. For FLAC files, only 8, 16, or 24 bits per sample are supported.

Example: `'BitsPerSample',32`

`'BitRate'` — Kilobits per second (kbit/s)

128 (default) | 64 | 96 | 160 | 192 | 256 | 320

Number of kilobits per second (kbit/s) used for compressed audio files, specified as the comma-separated pair consisting of `'BitRate'` and an integer. Noninteger values are truncated. On Windows 7 or later, the only valid values are 96, 128, 160, and 192.

In general, a larger `BitRate` value results in higher compression quality.

Only available for MPEG-4 (`.m4a`, `.mp4`) files.

Example: `'BitRate',96`

`'Quality'` — Quality setting for the Ogg Vorbis Compressor

75 (default) | value in the range [0 100]

Quality setting for the Ogg Vorbis Compressor, specified as the comma-separated pair consisting of `'Quality'` and a number in the range [0 100], where 0 is lower quality and higher compression, and 100 is higher quality and lower compression.

Only available for OGG (`.ogg`) files.

Example: `'Quality',25`

`'Title'` — Title information

[ ] (default) | string

Title information, specified as the comma-separated pair consisting of `'Title'` and a string.

`'Artist'` — Artist information

[ ] (default) | string

Artist information, specified as the comma-separated pair consisting of `'Artist'` and a string.

`'Comment'` — Additional information

[ ] (default) | string

Additional information, specified as the comma-separated pair consisting of `'Comment'` and a string.

Note: On Mac platforms, `audiowrite` writes metadata to WAVE, OGG, and FLAC files only, and will not write the `'Title'`, `'Author'`, or `'Comment'` fields to MPEG-4 AAC files.

More About

Algorithms

The output data type is determined by the file format, the data type of `y`, and the specified output `BitsPerSample`.

File Formats	Data Type of y	Output BitsPerSample	Output Data Type
WAVE (`.wav`),	`uint8`, `int16`, `int32`, `single`, `double`	8	`uint8`
		16	`int16`
		24	`int32`
	`uint8`, `int16`, `int32`	32	`int32`
	`single`, `double`	32	`single`
	`single`, `double`	64	`double`
FLAC (`.flac`)	`uint8`, `int16`, `int32`, `single`, `double`	8	`int8`
		16	`int16`
		24	`int32`
MPEG-4 (`.m4a`, `.mp4`), OGG (`.ogg`)	`uint8`, `int16`, `int32`, `single`, `double`	N/A	`single`

See Also

`audioinfo` | `audioread`

Introduced in R2012b

addframe (avifile)

Add frame to Audio/Video Interleaved (AVI) file

Note: `avifile` has been removed. Use `VideoWriter` instead.

Syntax

```
aviobj = addframe(aviobj,frame)
aviobj = addframe(aviobj,frame1,frame2,frame3,...)
aviobj = addframe(aviobj,mov)
aviobj = addframe(aviobj,h)
```

Description

`aviobj = addframe(aviobj,frame)` appends the data in *`frame`* to the AVI file identified by *`aviobj`*, which was created by a previous call to `avifile`. *`frame`* can be either an indexed image (`m`-by-`n`) or a truecolor image (`m`-by-`n`-by-3) of `double` or `uint8` precision. If *`frame`* is not the first frame added to the AVI file, it must be consistent with the dimensions of the previous frames.

`addframe` returns a handle to the updated AVI file object, *`aviobj`*. For example, `addframe` updates the `TotalFrames` property of the AVI file object each time it adds a frame to the AVI file.

`aviobj = addframe(aviobj,frame1,frame2,frame3,...)` adds multiple frames to an AVI file.

`aviobj = addframe(aviobj,mov)` appends the frames contained in the MATLAB movie *`mov`* to the AVI file *`aviobj`*. MATLAB movies that store frames as indexed images use the colormap in the first frame as the colormap for the AVI file, unless the colormap has been previously set.

`aviobj = addframe(aviobj,h)` captures a frame from the figure or axis handle *`h`* and appends this frame to the AVI file. `addframe` renders the figure into an offscreen array

before appending it to the AVI file. This ensures that the figure is written correctly to the AVI file even if the figure is obscured on the screen by another window or screen saver.

Examples

This example calls `addframe` to add frames to the AVI file object `aviobj`.

```
aviobj = avifile('example.avi','compression','None');

t = linspace(0,2.5*pi,40);
fact = 10*sin(t);
fig=figure;
[x,y,z] = peaks;
for k=1:length(fact)
    h = surf(x,y,fact(k)*z);
    axis([-3 3 -3 3 -80 80])
    axis off
    caxis([-90 90])

    F = getframe(fig);
    aviobj = addframe(aviobj,F);
end
close(fig);
aviobj = close(aviobj);
```

More About

Tips

- `avifile` cannot write files larger than 2 GB. Use `VideoWriter` and `writeVideo` to create larger files.
- If you are creating an animation from a figure with an `EraseMode` property set to `'xor'`, you must use `getframe` to capture the graphics into a frame of a MATLAB movie. Add the frame to the AVI file using the syntax *`aviobj`* `= addframe(`*`aviobj`*`,`*`mov`*`)`. See the example for an illustration.

See Also

`VideoWriter` | `close (avifile)` | `avifile`

Introduced before R2006a

addOptional

Class: inputParser

Add optional positional argument to input parser scheme

Syntax

```
addOptional(p,argName,default)
addOptional(p,argName,default,validationFcn)
```

Description

`addOptional(p,argName,default)` adds optional input, `argName`, to the input parser scheme of `inputParser` object, `p`. When the inputs that you are checking do not include a value for this optional input, the input parser assigns the `default` value to the input.

`addOptional(p,argName,default,validationFcn)` specifies a validation function for the input argument.

Tips

- For optional string inputs, specify a validation function. Without a validation function, the input parser interprets valid string inputs as invalid parameter names and throws an error.
- Use `addOptional` to add an individual argument into the input parser scheme. If you want to parse an optional name-value pair, use `addParameter`.

Input Arguments

`p`

Object of class `inputParser`.

`argName`

String that specifies the internal name for the input argument.

Arguments added with `addOptional` are positional. When you call a function with positional inputs:

- Specify inputs in the same order that they are added into the input parser scheme.
- Specify a value for the `k`th argument by assigning values for the first `(k–1)` arguments in the input scheme.

`default`

Default value for the input. This value can be of any data type.

`validationFcn`

Handle to a function that checks if the input argument is valid.

`inputParser` accepts two types of validation functions: functions that return `true` or `false`, and functions that pass or throw an error. Both types of validation functions must accept a single input argument.

Examples

Add Optional Input

Create an `inputParser` object and add an optional input named `myinput` with a default value of `0` to the input scheme.

```
p = inputParser;
argName = 'myinput';
default = 0;
addOptional(p,argName,default);
```

Validate Optional Input

Check whether an optional input named `num` with a default value of `1` is a numeric scalar greater than zero.

```
p = inputParser;
argName = 'num';
```

```
default = 1;
validationFcn = @(x) isnumeric(x) && isscalar(x) && (x > 0);
addOptional(p,argName,default,validationFcn)
```

The syntax `@(x)` creates a handle to an anonymous function with one input.

Attempt to parse an invalid input, such as `-1`:

```
parse(p,-1)
```

```
The value of 'num' is invalid. It must satisfy the function: @(x)isnumeric(x)&&isscalar(x)&&(x>0).
```

Validate Optional Input with `validateattributes`

Create an `inputParser` object and define a validation function using `validateattributes` to test that optional input is numeric, positive, and even. Add the required input to the scheme.

```
p = inputParser;
argName = 'evenPosNum';
default = 1;
validationFcn = @(x) validateattributes(x,{'numeric'},...
    {'even','positive'});
addOptional(p,argName,default,validationFcn)
```

Parse an input string. Parse will fail.

```
parse(p,'hello')
```

```
The value of 'evenPosNum' is invalid. Expected input to be one of these types:

double, single, uint8, uint16, uint32, uint64, int8, int16, int32, int64

Instead its type was char.
```

Parse an odd number. Parse will fail.

```
 parse(p,13)
```

```
The value of 'evenPosNum' is invalid. Expected input to be even.
```

Parse an even, positive number. Parse will pass.

```
parse(p,42)
```

See Also

`addParameter` | `addRequired` | `function_handle` | `inputParser` | `validateattributes`

More About

- “Anonymous Functions”

addParameter

Class: inputParser

Add optional parameter name-value pair argument to input parser scheme

Syntax

```
addParameter(p,paramName,default)
addParameter(p,paramName,default,validationFcn)
addParameter(___,'PartialMatchPriority', matchPriorityValue)
```

Description

`addParameter(p,paramName,default)` adds parameter name and value argument `paramName` to the input parser scheme of `inputParser` object, `p`. When the inputs that you are checking do not include a value for this optional parameter, the input parser assigns the `default` value.

`addParameter(p,paramName,default,validationFcn)` specifies a validation function for the input argument.

`addParameter(___,'PartialMatchPriority', matchPriorityValue)` allows specification of matchPriorityValue to indicate priority for matching of conflicting partial parameter names and can include any of the input arguments in the previous syntaxes.

Input Arguments

`p`

Object of class `inputParser`.

`paramName`

String that specifies the internal name for the input parameter.

Parameter names and values are optional inputs. When calling the function, name and value pairs can appear in any order, with the general form `Name1,Value1,...,NameN,ValueN`.

`default`

Default value for the input. This value can be of any data type.

`matchPriorityValue`

Positive integer value indicating the priority for matching of conflicting partial parameter names. The input parser selects lower priority values over higher ones. If, within an input parser scheme, partial parameter names are ambiguous and have the same priority, `inputParser` will throw an error. If the names are ambiguous, but have different `PartialMatchPriority` values, `inputParser` issues a warning indicating the matched name.

Default: 1

`validationFcn`

Handle to a function that checks if the input argument is valid.

`inputParser` accepts two types of validation functions: functions that return `true` or `false`, and functions that pass or throw an error. Both types of validation functions must accept a single input argument.

Examples

Add Optional Parameter Value Input

Create an `inputParser` object and add an optional input named `myparam` with a default value of `0` to the input scheme.

```
p = inputParser;
paramName = 'myparam';
default = 0;
addParameter(p,paramName,default);
```

Unlike the positional inputs added with the `addRequired` and `addOptional` methods, each parameter added with `addParameter` corresponds to two input arguments: one for the name and one for the value of the parameter.

Pass both the parameter name and value to the `parse` method.

```
parse(p,'myparam',100);
p.Results
```

```
ans =

    myparam: 100
```

Validate Parameter Value

Check whether the value corresponding to `myparam` is a numeric scalar greater than zero.

```
p = inputParser;
paramName = 'myparam';
default = 1;
errorStr = 'Value must be positive, scalar, and numeric.';
validationFcn = @(x) assert(isnumeric(x) && isscalar(x) ...
    && (x > 0),errorStr);
addParameter(p,paramName,default,validationFcn)
```

The syntax `@(x)` creates a handle to an anonymous function with one input.

Attempt to parse an invalid value, such as `-1`:

```
parse(p,'myparam',-1)
```

```
The value of 'myparam' is invalid. Value must be positive, scalar, and numeric.
```

Validate Parameter Value Input with `validateattributes`

Create an `inputParser` object and define a validation function using `validateattributes` to test that input parameter `myName` is a nonempty string.

```
p = inputParser;
paramName = 'myName';
default = 'John Doe';
validationFcn = @(x) validateattributes(x,{'char'},{'nonempty'});
addParameter(p,paramName,default,validationFcn)
```

Define `myName` as a number. The attempt to parse fails.

```
parse(p,'myName',1138)
```

```
The value of 'myName' is invalid. Expected input to be one of these types:

char

Instead its type was double.
```

Parse a string. The attempt to parse passes.

```
parse(p,'myName','George')
```

See Also

`addOptional` | `addRequired` | `function_handle` | `inputParser` | `validateattributes`

More About

- “Anonymous Functions”

addParamValue

Class: inputParser

(Not recommended) Add parameter name and value argument to Input Parser scheme

Compatibility

`addParamValue` is not recommended. Use `addParameter` instead.

Syntax

```
addParamValue(p,paramName,default)
addParamValue(p,paramName,default,validationFcn)
```

Description

`addParamValue(p,paramName,default)` adds parameter name and value argument `paramName` to the input scheme of `inputParser` object `p`. When the inputs that you are checking do not include a value for this optional parameter, the parser assigns the `default` value.

`addParamValue(p,paramName,default,validationFcn)` specifies a validation function for the input argument.

Tips

- If your parameter values do not require validation, you do not have to include them in the input scheme with `addParamValue`. As an alternative, set the `KeepUnmatched` property of the `inputParser` object to `true`. The parser stores extra parameter names and values in the `Unmatched` property rather than in the `Results` property of the object. For an example, see the `inputParser` reference page.

Input Arguments

p

Object of class `inputParser`.

paramName

String that specifies the internal name for the input parameter.

Parameter names and values are optional inputs. When calling the function, name and value pairs can appear in any order, with the general form `Name1,Value1,...,NameN,ValueN`.

default

Default value for the input. This value can be of any data type.

validationFcn

Handle to a function that checks if the input argument is valid.

`inputParser` accepts two types of validation functions: functions that return `true` or `false`, and functions that pass or throw an error. Both types of validation functions must accept a single input argument.

Examples

Add Optional Parameter Value Input

Create an `inputParser` object and add an optional input named `myparam` with a default value of `0` to the input scheme.

```
p = inputParser;
paramName = 'myparam';
default = 0;
addParameter(p,paramName,default);
```

Unlike the positional inputs added with the `addRequired` and `addOptional` methods, each parameter added with `addParameter` corresponds to two input arguments: one for the name and one for the value of the parameter.

Pass both the parameter name and value to the `parse` method.

```
parse(p,'myparam',100);
p.Results
```

```
ans =

    myparam: 100
```

Validate Parameter Value

Check whether the value corresponding to `myparam` is a numeric scalar greater than zero.

```
p = inputParser;
paramName = 'myparam';
default = 1;
validationFcn = @(x) isnumeric(x) && isscalar(x) && (x > 0);
addParamValue(p,paramName,default,validationFcn)
```

The syntax `@(x)` creates a handle to an anonymous function with one input.

Attempt to parse an invalid value, such as `-1`:

```
parse(p,'myparam',-1)
```

```
Argument 'myparam' failed validation @(x)isnumeric(x)&&isscalar(x)&&(x>0).
```

See Also

`addOptional` | `addParameter` | `addRequired` | `function_handle` | `inputParser`

More About

- “Anonymous Functions”

addcats

Add categories to categorical array

Syntax

```
B = addcats(A,newcats)
B = addcats(A,newcats,'Before',beforewhere)
B = addcats(A,newcats,'After',afterwhere)
```

Description

`B = addcats(A,newcats)` adds categories to the end of the category list for the input categorical array, `A`. The output categorical array, `B`, contains the same values as `A`. The output, `B`, does not contain any elements equal to the new categories until you assign values from `newcats` to elements in `B`.

If `A` is an ordinal categorical array, you must specify the `'Before',beforewhere` or `'After',afterwhere` input arguments.

`B = addcats(A,newcats,'Before',beforewhere)` adds categories before the category specified by beforewhere.

`B = addcats(A,newcats,'After',afterwhere)` adds categories after the category specified by afterwhere.

Examples

Add Categories at End

Create a nonordinal categorical array.

```
A = categorical({'republican' 'democrat' 'republican';...
    'democrat' 'republican' 'democrat'})
```

```
A =
```

```
     republican      democrat        republican
     democrat        republican      democrat
```

Display the categories of A.

```
categories(A)
```

```
ans =

    'democrat'
    'republican'
```

A is a 2-by-3 categorical array with two categories.

Add the categories, `independent` and `undeclared`, to the end of the category list

```
B = addcats(A,{'independent' 'undeclared'})
```

```
B =

     republican      democrat        republican
     democrat        republican      democrat
```

B contains the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans =

    'democrat'
    'republican'
    'independent'
    'undeclared'
```

B is a 2-by-3 categorical array with four categories.

Add Categories and Specify Category to Precede

Create an ordinal categorical array.

```
A = categorical({'medium' 'large'; 'small' 'xlarge'; 'large' 'medium'},...
```

```
    {'small' 'medium' 'large' 'xlarge'},'Ordinal',true)

A =

     medium      large
     small       xlarge
     large       medium
```

Display the categories of A.

```
categories(A)

ans =

    'small'
    'medium'
    'large'
    'xlarge'
```

Since A is ordinal, the categories have the mathematical ordering `small < medium < large < xlarge`.

Add the category `xsmall` before `small`.

```
B = addcats(A,'xsmall','Before','small')

B =

     medium      large
     small       xlarge
     large       medium
```

B contains the same values as A.

Display the categories of B.

```
categories(B)

ans =

    'xsmall'
    'small'
    'medium'
    'large'
    'xlarge'
```

The categories have the mathematical ordering `xsmall < small < medium < large < xlarge`.

Input Arguments

`A` — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`newcats` — New categories

string | cell array of strings

New categories, specified as a string or a cell array of strings.

`beforewhere` — Category to precede

string

Category to precede, specified as a string.

`afterwhere` — Category to follow

string

Category to follow, specified as a string.

See Also

`categories` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `reordercats` | `setcats`

addpath

Add folders to search path

Syntax

```
addpath(folderName1,...,folderNameN)
addpath(folderName1,...,folderNameN,position)

addpath( ___ ,'-frozen')

oldpath = addpath( ___ )
```

Description

`addpath(folderName1,...,folderNameN)` adds the specified folders to the top of the search path. Use `addpath` statements in a `startup.m` file to modify the search path programmatically at startup.

`addpath(folderName1,...,folderNameN,position)` adds the specified folders to the top or bottom of the search path, as specified by `position`.

`addpath( ___ ,'-frozen')` additionally disables folder change detection on Windows for folders being added, which conserves Windows change notification resources (Windows only). Type `help changenotification` in the Command Window for more information.

Add `'-frozen'` to the input arguments in any of the previous syntaxes. You can specify `'-frozen'` and `position` in either order.

`oldpath = addpath( ___ )` additionally returns the path prior to adding the specified folders.

Examples

Add Folder to Top of Search Path

If you do not have a folder called `c:/matlab/myfiles`, create the folder.

```
mkdir('c:/matlab/myfiles')
```

Add `c:/matlab/myfiles` to the top of the search path.

```
addpath('c:/matlab/myfiles')
```

Add Folder to End of Search Path

Add `c:/matlab/myfiles` to the end of the search path.

```
addpath('c:/matlab/myfiles','-end')
```

Add Folder and Its Subfolders to Search Path

Add `c:/matlab/myfiles` and its subfolders to the search path.

Call `genpath` inside of `addpath` to add all subfolders of `c:/matlab/myfiles` to the search path.

```
addpath(genpath('c:/matlab/myfiles'))
```

Add Folder to Search Path and Disable Folder Change Notification

On Windows, add the folder `c:/matlab/myfiles` to the top of the search path, disable folder change notification, and return the search path before adding the folder.

```
oldpath = addpath('c:/matlab/myfiles','-frozen');
```

Input Arguments

`folderName1,...,folderNameN` — Names of folders to add to search path

string

Names of folders to add to the search path, specified as strings. Use the full path name for each folder. Use `genpath` with `addpath` to add all subfolders of `folderName`.

Example: `'c:\matlab\work'`

Example: `'/home/user/matlab'`

Example: `'/home/user/matlab','/home/user/matlab/test'`

`position` — Position on the search path

`'-begin'` (default) | `'-end'`

Position on the search path, specified as one of the following strings.

Value of `position`	Description
`'-begin'`	Add specified folders to the top of the search path.
`'-end'`	Add specified folders to the bottom of the search path.

Output Arguments

`oldpath` — Path prior to addition of folders

string

Path prior to the addition of folders, returned as a string.

More About

Tips

- If you use `addpath` within a local function, the path change persists after program control returns from the function. That is, the scope of the path change is global.

- "What Is the MATLAB Search Path?"
- "Files and Folders that MATLAB Accesses"
- "Specify File Names"
- "Startup Options in MATLAB Startup File"

See Also

`genpath` | `path` | `pathsep` | `rmpath` | `savepath`

Introduced before R2006a

addpref

Add preference

Syntax

```
addpref('group','pref',val)
addpref('group',{'pref1','pref2',...'prefn'},{val1,val2,...valn})
```

Description

`addpref('group','pref',val)` creates the preference specified by `group` and `pref` and sets its value to `val`. It is an error to add a preference that already exists. Individual preference values can be any MATLAB data type, including numeric types, strings, cell arrays, structures, and objects.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g. `'ApplicationOnePrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`addpref('group',{'pref1','pref2',...'prefn'},{val1,val2,...valn})` creates the preferences specified by the cell array of names `'pref1'`, `'pref2',...,'prefn'`, setting each to the corresponding value.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

Add a preference called `version` to the `mytoolbox` group of preferences, setting its value to the cell array `{'1.0','beta'}`.

```
addpref('mytoolbox','version',{'1.0','beta'})
```

Add a preference called `documentation` to the `mytoolbox` group of preferences, setting its value to a struct you define as having four fields:

```
mydoc.docexists = 1;
mydoc.docpath = fullfile(docroot,'techdoc',...
   'matlab_env','examples');
mydoc.demoexists = 1;
mydoc.demopath = fullfile(docroot,'techdoc',...
   'matlab_env','examples','demo_examples');
addpref('mytoolbox','documentation',mydoc)
% Retrieve the preference with GETPREF
p = getpref('mytoolbox','documentation')

p =
     docexists: 1
       docpath: [1x109 char]
    demoexists: 1
      demopath: [1x123 char]
```

See Also

`getpref` | `ispref` | `rmpref` | `setpref` | `uigetpref` | `uisetpref`

Introduced before R2006a

addpoints

Add points to animated line

To use this function, you must first create an animated line with the `animatedline` function. For more information on line animations, see Using Animated Line Objects.

Syntax

```
addpoints(h,x,y)
addpoints(h,x,y,z)
```

Description

`addpoints(h,x,y)` adds points defined by `x` and `y` to the animated line specified by `h`. To display the updates on the screen, use `drawnow` or `drawnow update`. New points automatically connect to previous points.

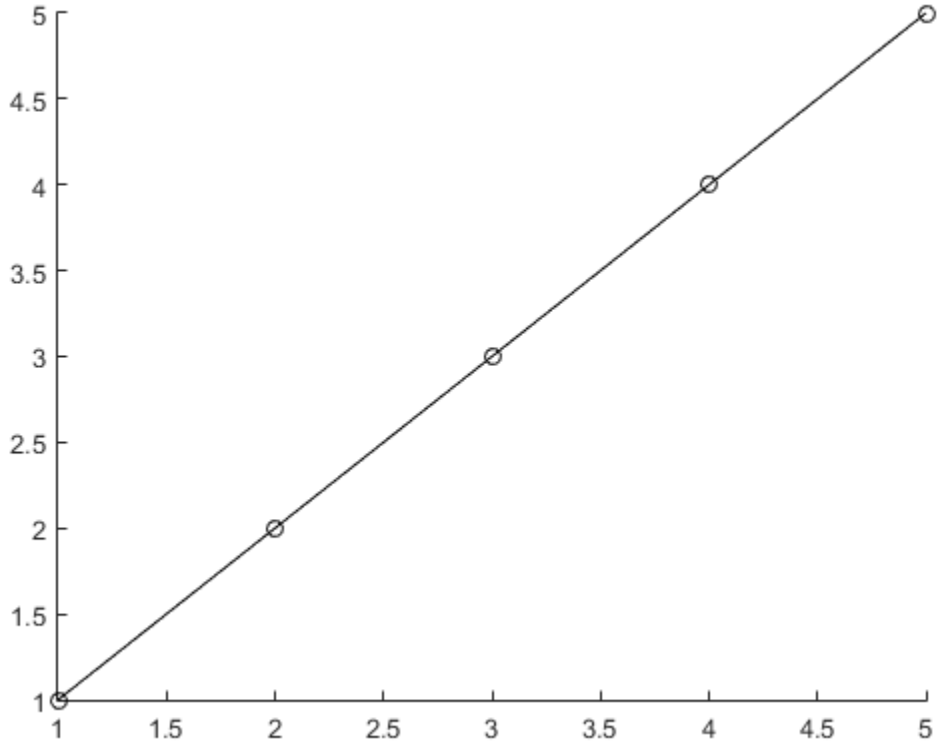
`addpoints(h,x,y,z)` adds points defined by `x`, `y`, and `z` to the 3-D animated line specified by `h`.

Examples

Add Five Points to Animated Line

Create an animated line object with no data. Then, add five points to the line. Use a circle to mark each point.

```
h = animatedline('Marker','o');
x = 1:5;
y = 1:5;
addpoints(h,x,y)
```

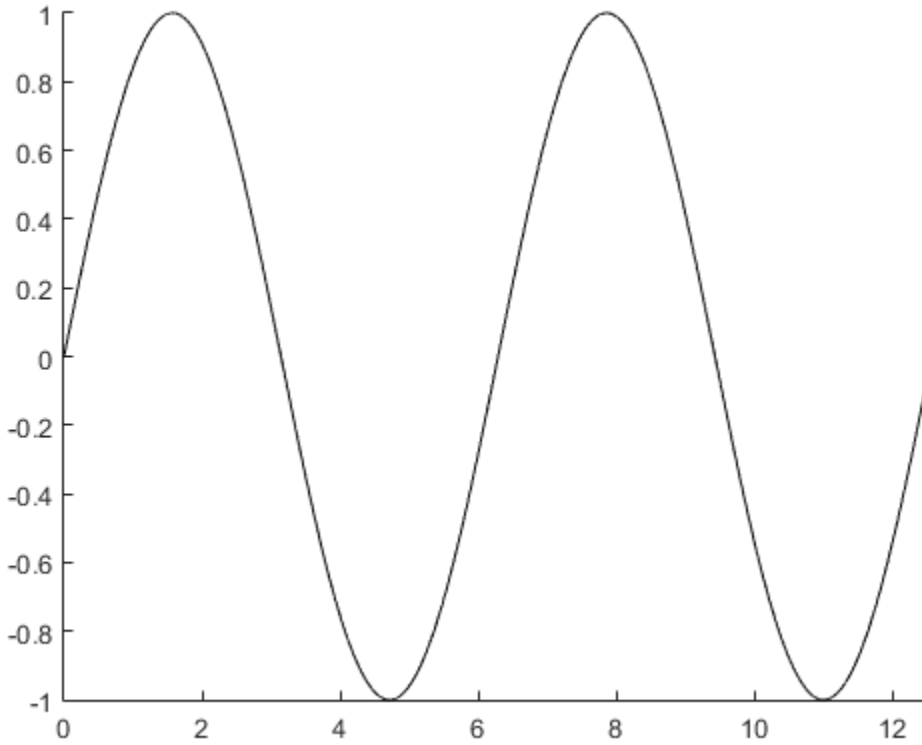


Add Points Within Loop to Animated Line

Create an animated line using the `animatedline` function. Then, add points to the line within a loop to create an animation. Set the axis limits before the loop to prevent the limits from changing.

```
figure
h = animatedline;
axis([0 4*pi -1 1])

for x = linspace(0,4*pi,10000)
    y = sin(x);
    addpoints(h,x,y)
    drawnow limitrate
```

```
end
```



Input Arguments

h — Animated line object

animated line object

Animated line object. Create an animated line using the `animatedline` function.

x — x values

scalar | vector

x values, specified as a scalar or a vector. The length of `x` must equal the length of `y`.

Example: `11:20`

Data Types: `double`

y — y values

scalar | vector

y values, specified as a scalar or a vector. The length of `y` must equal the length of `x`.

Example: `11:20`

Data Types: `double`

z — z values

scalar | vector

z values, specified as a scalar or a vector. The length of `z` must equal the length of `x` and `y`.

Example: `11:20`

Data Types: `double`

See Also

Functions

`animatedline` | `clearpoints` | `getpoints`

Using Objects

Using Animated Line Objects

addprop (dynamicprops)

Add dynamic property

Syntax

```
P = addprop(Hobj,'PropName')
```

Description

`P = addprop(Hobj,'PropName')` adds a property named `PropName` to each object in array `Hobj`. The class definition is not affected by the addition of dynamic properties. Note that you can add dynamic properties only to objects derived from the `dynamicprops` class. You can set and retrieve the data in dynamic properties as you would any property.

The output argument `P` is an array the same size as `Hobj` of `meta.DynamicProperty` objects, which you can use to assign `SetMethod` and `GetMethod` functions to the property. These functions operate just like property set and get access methods.

See "Dynamic Properties — Adding Properties to an Instance" for more information and examples.

See Also

`handle` | `dynamicprops`

addproperty

Add custom property to COM object

Syntax

```
addproperty(h,'name')
```

Description

`addproperty(h,'name')` adds custom property specified in string, `name`, to object or interface, `h`.

COM functions are available on Microsoft Windows systems only.

Examples

This example adds a custom property to the MATLAB sample control.

1 Create an instance of the control:

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f);
get(h)
```

```
     Label: 'Label'
    Radius: 20
```

MATLAB also displays interfaces.

2 Add a custom property named `Position` and assign a value:

```
addproperty(h,'Position');
h.Position = [200 120];
get(h)
```

```
       Label: 'Label'
      Radius: 20
    Position: [200 120]
```

3 Delete the custom property `Position`:

```
deleteproperty(h,'Position');
get(h)
```

```
     Label: 'Label'
    Radius: 20
```

MATLAB displays the original list of properties:

More About

- "Use Object Properties"

See Also

`deleteproperty` | `set (COM)` | `inspect` | `get (COM)`

Introduced before R2006a

addRequired

Class: inputParser

Add required positional argument to input parser scheme

Syntax

```
addRequired(p,argName)
addRequired(p,argName,validationFcn)
```

Description

`addRequired(p,argName)` adds required argument, `argName`, to the input parser scheme of `inputParser` object, `p`.

`addRequired(p,argName,validationFcn)` includes a validation function for the input argument.

Input Arguments

`p`

Object of class `inputParser`.

`argName`

String that specifies the internal name for the input argument.

Arguments added with `addRequired` are positional. When you call a function with positional inputs, you must specify inputs in the order that they are added to the input parser scheme.

`validationFcn`

Handle to a function that checks if the input argument is valid.

`inputParser` accepts two types of validation functions: functions that return `true` or `false`, and functions that pass or throw an error. Both types of validation functions must accept a single input argument.

Examples

Add Required Input

Create an `inputParser` object and add a required input named `myinput` to the input scheme.

```
p = inputParser;
argName = 'myinput';
addRequired(p,argName);
```

Validate Required Input

Check whether a required input named `num` is a numeric scalar greater than zero.

```
p = inputParser;
argName = 'num';
validationFcn = @(x) isnumeric(x) && isscalar(x) && (x > 0);
addRequired(p,argName,validationFcn)
```

The syntax `@(x)` creates a handle to an anonymous function with one input.

Attempt to parse an invalid input, such as `-1`:

```
parse(p,-1)
```

```
The value of 'num' is invalid. It must satisfy the function: @(x)isnumeric(x)&&isscalar(x)&&(x>0).
```

Validate Required Input with `validateattributes`

Create an `inputParser` object and define a validation function using `validateattributes` to test that required input is numeric, positive, and even. Add the required input to the scheme.

```
p = inputParser;
argName = 'evenPosNum';
validationFcn = @(x) validateattributes(x,{'numeric'},...
    {'even','positive'});
addRequired(p,argName,validationFcn)
```

Parse an input string. Parse will fail.

```
parse(p,'hello')
```

```
The value of 'evenPosNum' is invalid. Expected input to be one of these types:

double, single, uint8, uint16, uint32, uint64, int8, int16, int32, int64

Instead its type was char.
```

Parse an odd number. Parse will fail.

```
 parse(p,13)
```

```
The value of 'evenPosNum' is invalid. Expected input to be even.
```

Parse an even, positive number. Parse will pass.

```
parse(p,42)
```

See Also

`addOptional` | `addParameter` | `function_handle` | `inputParser` | `validateattributes`

More About

- “Anonymous Functions”

addsampletocollection

Add sample to `tscollection` object

Syntax

```
tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data,
TSnName,TSnData)
```

Description

`tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data, TSnName,TSnData)` adds data samples `TSnData` to the collection member `TSnName` in the `tscollection` object `tsc` at one or more `Time` values. Here, `TSnName` is the string that represents the name of a time series in `tsc`, and `TSnData` is an array containing data samples.

Examples

The following example shows how to create a `tscollection` that consists of two `timeseries` objects, where one `timeseries` does not have quality codes and the other does. The final step of the example adds a sample to the `tscollection`.

1. Create two `timeseries` objects, `ts1` and `ts2`.

   ```
   ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                    'name','acceleration');
   ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
                    'name','speed');
   ```

2. Define a dictionary of quality codes and descriptions for `ts2`.

   ```
   ts2.QualityInfo.Code = [0 1];
   ts2.QualityInfo.Description = {'bad','good'};
   ```

3. Assign a quality of code of `1`, which is equivalent to `'good'`, to each data value in `ts2`.

   ```
   ts2.Quality = ones(5,1);
   ```

4 Create a time-series collection `tsc`, which includes time series `ts1` and `ts2`.

```
tsc = tscollection({ts1,ts2});
```

5 Add a data sample to the collection `tsc` at `3.5` seconds.

```
tsc = addsampletocollection(tsc,'time',3.5,'acceleration',10,'speed',{5 1});
```

The cell array for the `timeseries` object `'speed'` specifies both the data value 5 and the quality code `1`.

Note If you do not specify a quality code when adding a data sample to a time series that has quality codes, then the lowest quality code is assigned to the new sample by default.

More About

Tips

If you do not specify data samples for a time-series member in tsc, that time-series member will contain missing data at the times given by Time (for numerical time-series data), `NaN` values, or (for logical time-series data) `false` values.

When a time-series member requires `Quality` values, you can specify data quality codes together with the data samples by using the following syntax:

```
tsc = addsampletocollection(tsc,'time',time,TS1Name,...
         ts1cellarray,TS2Name,ts2cellarray,...)
```

Specify data in the first cell array element and `Quality` in the second cell array element.

Note If a time-series member already has `Quality` values but you only provide data samples, `0`s are added to the existing `Quality` array at the times given by `Time`.

See Also

`delsamplefromcollection` | `timeseries` | `tscollection`

Introduced before R2006a

addtodate

Modify date number by field

Syntax

```
R = addtodate(D, Q, F)
```

Description

`R = addtodate(D, Q, F)` adds quantity `Q` to the indicated date field `F` of a scalar serial date number `D`, returning the updated date number `R`.

The quantity `Q` to be added can be a positive or negative integer. The absolute value of `Q` must be less than or equal to 1e16. The date field `F` must be a 1-by-`N` character array equal to one of the following: `'year'`, `'month'`, `'day'`, `'hour'`, `'minute'`, `'second'`, or `'millisecond'`.

If the addition to the date field causes the field to roll over, the MATLAB software adjusts the next more significant fields accordingly. Adding a negative quantity to the indicated date field rolls back the calendar on the indicated field. If the addition causes the field to roll back, MATLAB adjusts the next less significant fields accordingly.

Examples

Modify the hours, days, and minutes of a given date:

```
t = datenum('07-Apr-2008 23:00:00');
datestr(t)
ans =
   07-Apr-2008 23:00:00

t= addtodate(t, 2, 'hour');
datestr(t)
ans =
   08-Apr-2008 01:00:00
```

```
t= addtodate(t, -7, 'day');
datestr(t)
ans =
   01-Apr-2008 01:00:00

t= addtodate(t, 59, 'minute');
datestr(t)
ans =
   01-Apr-2008 01:59:00
```

Adding 20 days to the given date in late December causes the calendar to roll over to January of the next year:

```
R = addtodate(datenum('12/24/2007 12:45'), 20, 'day');

datestr(R)
ans =
   13-Jan-2008 12:45:00
```

See Also

`date` | `datenum` | `datestr` | `datevec`

Introduced before R2006a

addts

Add `timeseries` object to `tscollection` object

Syntax

```
tsc = addts(tsc,ts)
tsc = addts(tsc,ts)
tsc = addts(tsc,ts,Name)
tsc = addts(tsc,Data,Name)
```

Description

`tsc = addts(tsc,ts)` adds the `timeseries` object ts to `tscollection` object tsc.

`tsc = addts(tsc,ts)` adds a cell array of `timeseries` objects ts to the `tscollection` tsc.

`tsc = addts(tsc,ts,Name)` adds a cell array of `timeseries` objects ts to `tscollection` tsc. Name is a cell array of strings that gives the names of the `timeseries` objects in `ts`.

`tsc = addts(tsc,Data,Name)` creates a new `timeseries` object from Data with the name Name and adds it to the `tscollection` object tsc. Data is a numerical array and Name is a string.

Examples

The following example shows how to add a time series to a time-series collection:

1 Create two `timeseries` objects, `ts1` and `ts2`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                 'name','acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
```

```
                    'name','speed');
```

2 Create a time-series collection `tsc`, which includes `ts1`.

```
tsc = tscollection(ts1);
```

3 Add `ts2` to the `tsc` collection.

```
tsc = addts(tsc, ts2);
```

4 To view the members of `tsc`, type

```
tsc
```

at the MATLAB prompt. the response is

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:

      acceleration
      speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of the `timeseries` objects `ts1` and `ts2`, respectively.

More About

Tips

The `timeseries` objects you add to the collection must have the same time vector as the collection. That is, the time vectors must have the same time values and units.

Suppose that the time vector of a `timeseries` object is associated with calendar dates. When you add this `timeseries` to a collection with a time vector without calendar dates, the time vectors are compared based on the units and the values relative to the `StartDate` property. For more information about properties, see the `timeseries` reference page.

See Also

`removets` | `tscollection`

Introduced before R2006a

airy

Airy Functions

Syntax

```
W = airy(Z)
W = airy(k,Z)
W = airy(k,Z,scale)
```

Description

`W = airy(Z)` returns the Airy function, *Ai*(*Z*), for each element of *Z*.

`W = airy(k,Z)` returns any of four different Airy functions, depending on the value of `k`, such as the Airy function of the second kind or the first derivative of an Airy function.

`W = airy(k,Z,scale)` scales the resulting Airy function. `airy` applies a specific scaling function to `W` depending on your choice of `k` and `scale`.

Examples

Airy Function of Real-Valued x

Define `x`.

```
x = -10:0.01:1;
```

Calculate *Ai(x)*

```
ai = airy(x);
```

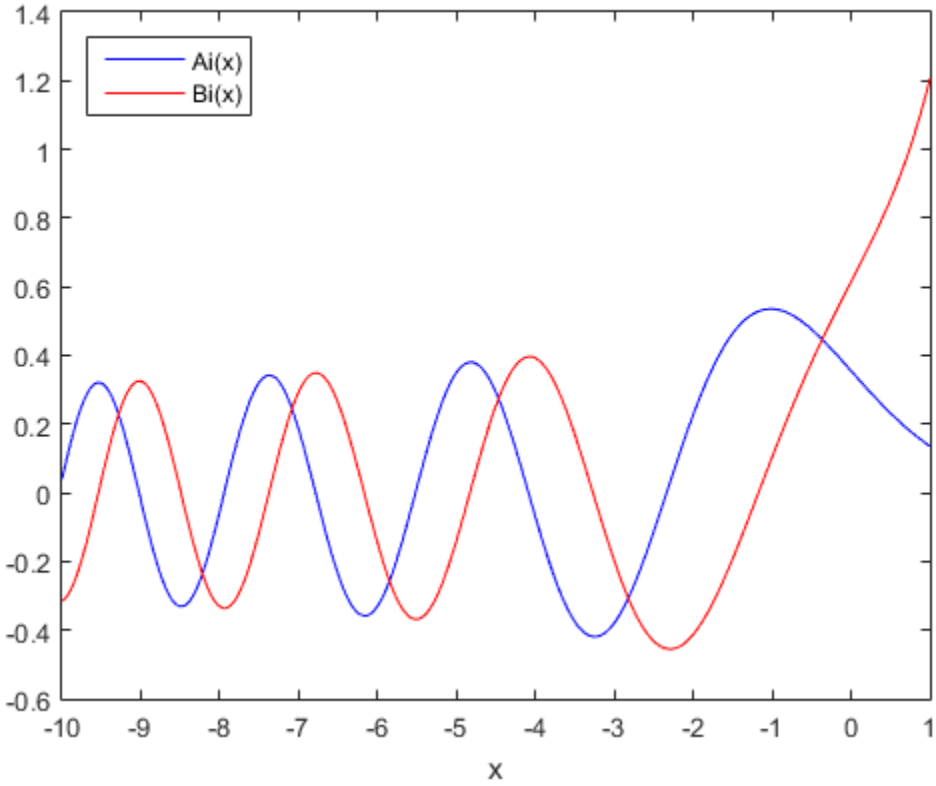
Calculate *Bi(x)* using $k = 2$.

```
bi = airy(2,x);
```

Plot both results together on the same axes.

```
figure
plot(x,ai,'-b',x,bi,'-r')
axis([-10 1 -0.6 1.4])
```

```
xlabel('x')
legend('Ai(x)','Bi(x)','Location','NorthWest')
```



Airy Function of Complex-Valued x

Compute the Airy function at a slice through the complex plane at $x + i$.

Take a slice through the complex plane.

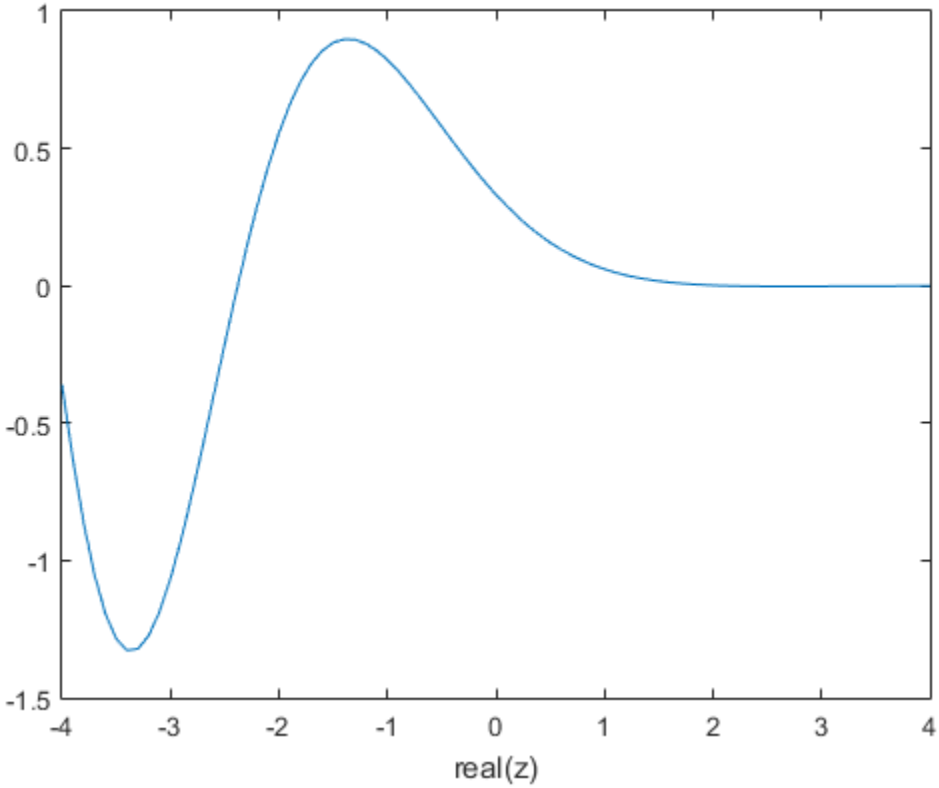
```
x = -4:0.1:4;
z = x+1i;
```

Calculate *Ai(z)*.

```
w = airy(z);
```

Plot the real part of the result.

```
figure
plot(x, real(w))
axis([-4 4 -1.5 1])
xlabel('real(z)')
```



Scaled Airy Function

Define `x`.

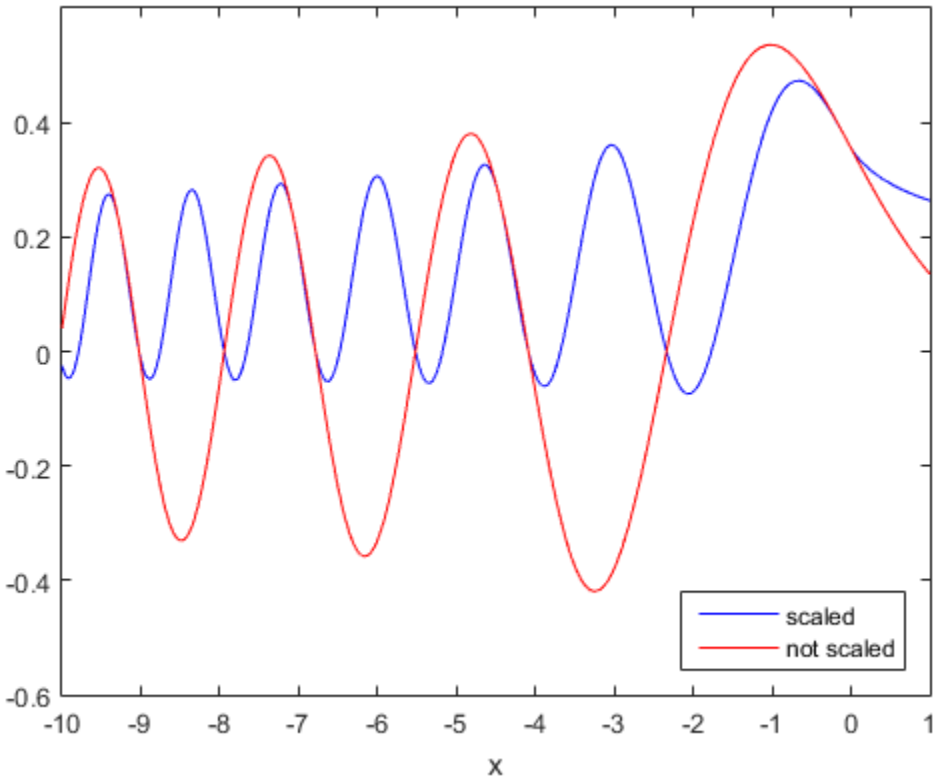
```
x = -10:0.01:1;
```

Calculate the scaled and unscaled Airy function.

```
scaledAi = airy(0,x,1);
noscaleAi =  airy(0,x,0);
```

Plot the real part of each result.

```
rscaled = real(scaledAi);
rnoscale = real(noscaleAi);
figure
plot(x,rscaled,'-b',x,rnoscale,'-r')
axis([-10 1 -0.60 0.60])
xlabel('x')
legend('scaled','not scaled','Location','SouthEast')
```



Input Arguments

Z — System variable

vector | matrix | N-D Array

System variable, specified as a real or complex vector, matrix, or N-D array.

Data Types: `single` | `double`
Complex Number Support: Yes

k — Type of Airy function

0 (default) | 1 | 2 | 3

Type of Airy function, specified as one of four values.

k	Returns
0	Airy function, $Ai(Z)$, which is the same as `airy(Z)`.
1	First derivative of Airy function, $Ai'(Z)$.
2	Airy function of the second kind, $Bi(Z)$
3	First derivative of Airy function of the second kind, $Bi'(Z)$

Data Types: `single` | `double`

`scale` — Scaling option

0 (default) | 1

Scaling option, specified as `0` or `1`. Use `scale = 1` to enable the scaling of `Z`. The values you specify for `k` and `scale` determine the scaling function `airy` applies to `Z`.

scale	k	Scaling applied to output
0	Any	None
1	0 or 1	$e^{\frac{2}{3}Z^{(3/2)}}$
1	2 or 3	$e^{-\left\lvert\frac{2}{3}Re\left(Z^{(3/2)}\right)\right\rvert}$

Data Types: `single` | `double`

Output Arguments

`W` — Airy function of `Z`

vector | matrix | N-D Array

Airy function of `Z`, returned as an array the same size as `Z`.

More About

Airy Functions

The Airy functions form a pair of linearly independent solutions to

$$\frac{d^2W}{dZ^2} - ZW = 0.$$

The relationship between the Airy and modified Bessel functions is

$$Ai(Z) = \left[\frac{1}{\pi}\sqrt{\frac{Z}{3}}\right] K_{1/3}(\zeta)$$

$$Bi(Z) = \sqrt{\frac{Z}{3}}\left[I_{-1/3}(\zeta) + I_{1/3}(\zeta)\right],$$

where

$$\zeta = \frac{2}{3}Z^{3/2}.$$

See Also

`besselh` | `besseli` | `besselj` | `besselk` | `bessely`

Introduced before R2006a

align

Align user interface controls (`uicontrols`) and axes

Syntax

```
align(HandleList,'HorizontalAlignment','VerticalAlignment')
Positions = align(HandleList, 'HorizontalAlignment',
'VerticalAlignment')
Positions = align(CurPositions, 'HorizontalAlignment',
'VerticalAlignment')
```

Description

`align(HandleList,'HorizontalAlignment','VerticalAlignment')` aligns the `uicontrol` and `axes` objects in `HandleList`, a vector of handles, according to the options `HorizontalAlignment` and `VerticalAlignment`. The following tables show the possible values for `HorizontalAlignment` and `VerticalAlignment`.

`HorizontalAlignment`	Definition
`None`	No horizontal alignment
`Left`	Shifts the objects' left edges to that of the first object selected
`Center`	Shifts objects to center their positions to the average of the extreme *x*-values of the group
`Right`	Shifts the objects' right edges to that of the first object selected
`Distribute`	Equalizes *x*-distances between all objects within the span of the extreme *x*-values
`Fixed`	Spaces objects to have a specified number of points between them in the *y*-direction

`VerticalAlignment`	Definition
`None`	No vertical alignment

VerticalAlignment	Definition
Top	Shifts the objects' top edges to that of the first object selected
Middle	Shifts objects to center their positions to the average of the extreme *y*-values of the group
Bottom	Shifts the objects' bottom edges to that of the first object selected
Distribute	Equalizes *y*-distances between all objects within the span of the extreme *y*-values
Fixed	Spaces objects to have a specified number of points between them in the *x*-direction

Aligning objects does not change their absolute sizes. All alignment options align the objects within the bounding box that encloses the objects. `Distribute` and `Fixed` align objects to the bottom left of the bounding box. `Distribute` evenly distributes the objects while `Fixed` distributes the objects with a fixed distance (in points) between them. When you specify both horizontal and vertical distance together, the keywords `'HorizontalAlignment'` and `'VerticalAlignment'` are not necessary.

If you use `Fixed` for `HorizontalAlignment` or `VerticalAlignment`, you must also specify the distance, in points, where 72 points equals 1 inch. For example:

```
align(HandleList,'Fixed',Distance,'VerticalAlignment')
```

distributes the specified components `Distance` points horizontally and aligns them vertically as specified.

```
align(HandleList,'HorizontalAlignment','Fixed',Distance)
```

aligns the specified components horizontally as specified and distributes them `Distance` points vertically.

```
align(HandleList,'Fixed',HorizontalDistance,...
      'Fixed',VerticalDistance)
```

distributes the specified components `HorizontalDistance` points horizontally and distributes them `VerticalDistance` points vertically.

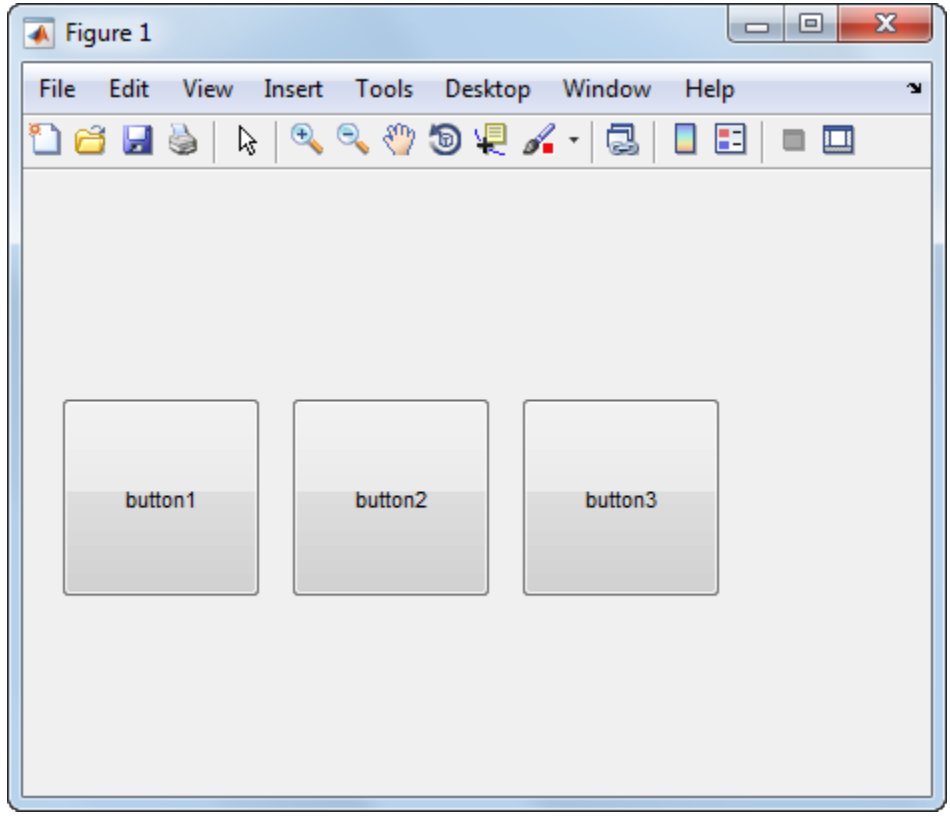
`Positions = align(HandleList, 'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the specified objects as a vector of `Position` vectors. The position of the objects on the figure does not change.

`Positions = align(CurPositions, 'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the objects whose positions are contained in `CurPositions`, where `CurPositions` is a vector of `Position` vectors. The position of the objects on the figure does not change.

Examples

Create a UI containing three buttons, and use the `align` function to line up the buttons.

```
% Create a figure window and one button object:
f=figure;
u1 = uicontrol('Style','push', 'parent', f,'pos',...
[20 100 100 100],'string','button1');
% Create two more button objects, not aligned with
% each other or any part of the figure window:
u2 = uicontrol('Style','push', 'parent', f,'pos',...
[150 250 100 100],'string','button2');
u3 = uicontrol('Style','push', 'parent', f,'pos',...
[250 100 100 100],'string','button3');
% Align the button objects with the bottom of the first
% button object, equalizing the distance between the
% objects within the span of the extreme x-values:
align([u1 u2 u3],'distribute','bottom');
```



More About

- "Align Objects in Graph Using Alignment Tools"

See Also

`uicontrol` | `uistack`

alim

Set or query axes alpha limits

Syntax

```
alpha_limits = alim
alim([amin amax])
alim_mode = alim('mode')
alim('alim_mode')
alim(axes_handle,...)
```

Description

`alpha_limits = alim` returns the alpha limits (`ALim` property) of the current axes.

`alim([amin amax])` sets the alpha limits to the specified values. `amin` is the value of the data mapped to the first alpha value in the alphamap, and `amax` is the value of the data mapped to the last alpha value in the alphamap. Data values in between are linearly interpolated across the alphamap, while data values outside are clamped to either the first or last alphamap value, whichever is closest.

`alim_mode = alim('mode')` returns the alpha limits mode (`ALimMode` property) of the current axes.

`alim('alim_mode')` sets the alpha limits mode on the current axes. `alim_mode` can be

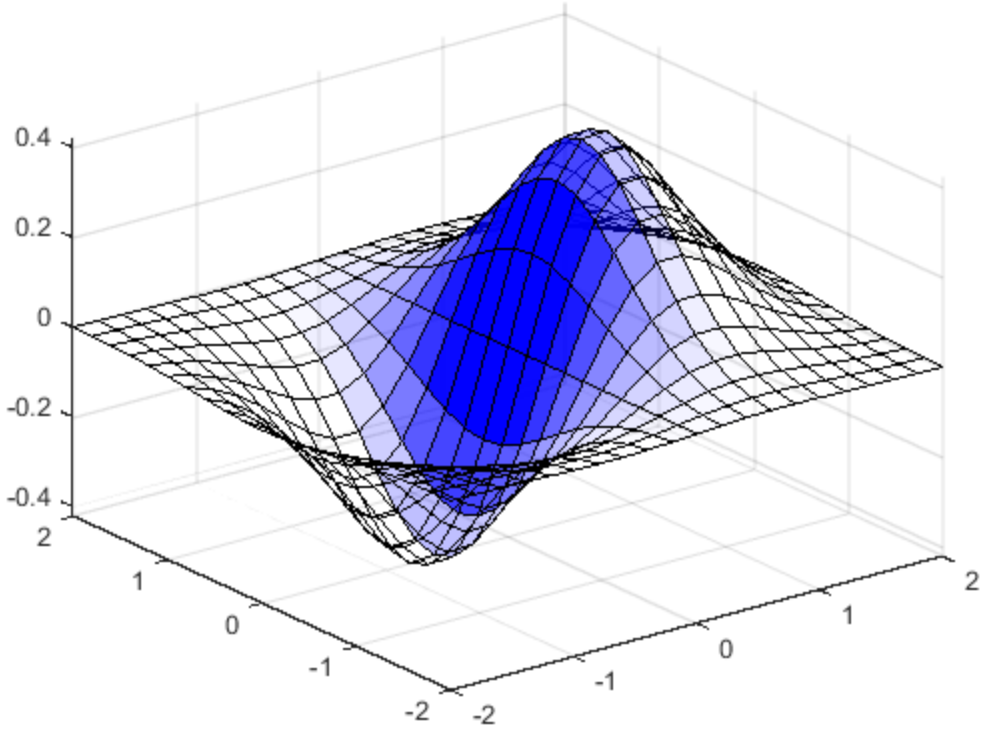
- `auto` — MATLAB automatically sets the alpha limits based on the alpha data of the objects in the axes.
- `manual` — MATLAB does not change the alpha limits.

`alim(axes_handle,...)` operates on the specified axes.

Examples

Map transparency to a surface plot of `z`-data and change the `alim` property to make all values below zero transparent:

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
% Plot the data, using the gradient of z as
% the alphamap:
surf(x,y,z+.001,'FaceAlpha','flat',...
'AlphaDataMapping','scaled',...
'AlphaData',gradient(z),...
'FaceColor','blue');
axis tight
% Adjust the alim property to see only where
% the gradient is between 0 and 0.15:
alim([0 .15])
```



See Also

Functions
`alpha` | `alphamap` | `caxis`

Properties
Axes Properties

Introduced before R2006a

all

Determine if all array elements are nonzero or `true`

Syntax

```
B = all(A)
B = all(A,dim)
```

Description

`B = all(A)` tests along the first array dimension of `A` whose size does not equal 1, and determines if the elements are all nonzero or logical `1` (`true`). In practice, `all` is a natural extension of the logical AND operator.

- If `A` is a vector, then `all(A)` returns logical `1` (`true`) if all the elements are nonzero and returns logical `0` (`false`) if one or more elements are zero.
- If `A` is a nonempty, nonvector matrix, then `all(A)` treats the columns of `A` as vectors and returns a row vector of logical `1`s and `0`s.
- If `A` is an empty 0-by-0 matrix, then `all(A)` returns logical `1` (`true`).
- If `A` is a multidimensional array, then `all(A)` acts along the first array dimension whose size does not equal 1 and returns an array of logical values. The size of this dimension becomes `1`, while the sizes of all other dimensions remain the same.

`B = all(A,dim)` tests elements along dimension `dim`. The `dim` input is a positive integer scalar.

Examples

Test Matrix Columns

Create a 3-by-3 matrix.

```
A = [0 0 3;0 0 3;0 0 3]
```

```
A =
```

```
     0     0     3
     0     0     3
     0     0     3
```

Test each column for all nonzero elements.

```
B = all(A)

B =

     0     0     1
```

Reduce a Logical Vector to a Single Condition

Create a vector of decimal values and test which values are less than 0.5.

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69];
B = (A < 0.5)

B =

0   0   1   1   1   1   0
```

The output is a vector of logical values. The `all` function reduces such a vector of logical values to a single condition. In this case, `B = all(A < 0.5)` yields logical `0`.

This makes `all` particularly useful in `if` statements.

```
if all(A < 0.5)
    %do something
else
    %do something else
end
```

The code is executed depending on a single condition, rather than a vector of possibly conflicting conditions.

Test Arrays of Any Dimension

Create a 3-by-7-by-5 multidimensional array and test to see if all of its elements are less than 3.

```
A = rand(3,7,5) * 5;
```

```
B = all(A(:) < 3)

B =

     0
```

You can also test the array for elements that are greater than zero.

```
B = all(A(:) > 0)

B =

     1
```

The syntax `A(:)` turns the elements of `A` into a single column vector, so you can use this type of statement on an array of any size.

Test Matrix Rows

Create a 3-by-3 matrix.

```
A = [0 0 3;0 0 3;0 0 3]

A =

     0     0     3
     0     0     3
     0     0     3
```

Test the rows of `A` for all nonzero elements by specifying `dim = 2`.

```
B = all(A,2)

B =

     0
     0
     0
```

Input Arguments

A — Input Array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

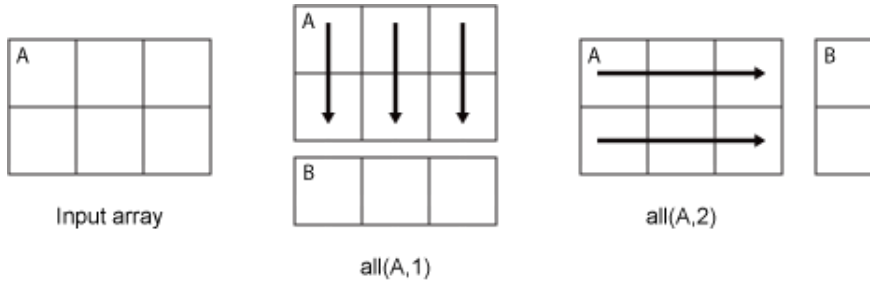
`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `all(A,1)` works on successive elements in the columns of `A` and returns a row vector of logical values.
- `all(A,2)` works on successive elements in the rows of `A` and returns a column vector of logical values.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`B` — Logical array

scalar | vector | matrix | multidimensional array

Logical array, returned as a scalar, vector, matrix, or multidimensional array. The dimension of `A` acted on by `all` has size `1` in `B`.

More About

- "Reduce Logical Arrays to Single Value"

See Also

`and` | `any` | `colon (:)` | `prod` | `sum`

Introduced before R2006a

allchild

Find all children of specified objects

Syntax

```
child_handles = allchild(handle_list)
```

Description

`child_handles = allchild(handle_list)` returns the list of all children (including ones with hidden handles) for each handle. If `handle_list` is a single element, `allchild` returns the output in a vector. If `handle_list` is a vector of handles, the output is a cell array.

Examples

Compare the results these two statements return:

```
axes
get(gca,'Children')
allchild(gca)
```

See Also

`findall` | `findobj`

Introduced before R2006a

alpha

Set transparency properties for objects in current axes

Syntax

```
alpha
alpha(object_handle,value)
alpha(face_alpha)
alpha(alpha_data)
alpha(alpha_data)
alpha(alpha_data_mapping)
```

Description

`alpha` sets one of three transparency properties, depending on what arguments you specify with the call to this function. For available arguments, see Inputs.

`alpha(object_handle,value)` sets the transparency property only on the object identified by `object_handle`.

Input Arguments

Face Alpha

`alpha(face_alpha)` sets the `FaceAlpha` property of all image, patch, and surface objects in the current axes. You can set `face_alpha` to

scalar	Set the `FaceAlpha` property to the specified value (for images, set the `AlphaData` property to the specified value).
`'flat'`	Set the `FaceAlpha` property to `flat`.
`'interp'`	Set the `FaceAlpha` property to `interp`.
`'texture'`	Set the `FaceAlpha` property to `texture`.
`'opaque'`	Set the `FaceAlpha` property to `1`.

`'clear'`	Set the `FaceAlpha` property to `0`.

See “Specifying Transparency” for more information.

AlphaData (Surface Objects)

`alpha(alpha_data)` sets the `AlphaData` property of all surface objects in the current axes. You can set `alpha_data` to

matrix the same size as `CData`	Set the `AlphaData` property to the specified values.
`'x'`	Set the `AlphaData` property to be the same as `XData`.
`'y'`	Set the `AlphaData` property to be the same as `YData`.
`'z'`	Set the `AlphaData` property to be the same as `ZData`.
`'color'`	Set the `AlphaData` property to be the same as `CData`.
`'rand'`	Set the `AlphaData` property to a matrix of random values equal in size to `CData`.

AlphaData (Image Objects)

`alpha(alpha_data)` sets the `AlphaData` property of all image objects in the current axes. You can set `alpha_data` to

matrix the same size as `CData`	Set the `AlphaData` property to the specified value.
`'x'`	Ignored.
`'y'`	Ignored.
`'z'`	Ignored.
`'color'`	Set the `AlphaData` property to be the same as `CData`.
`'rand'`	Set the `AlphaData` property to a matrix of random values equal in size to `CData`.

AlphaDataMapping

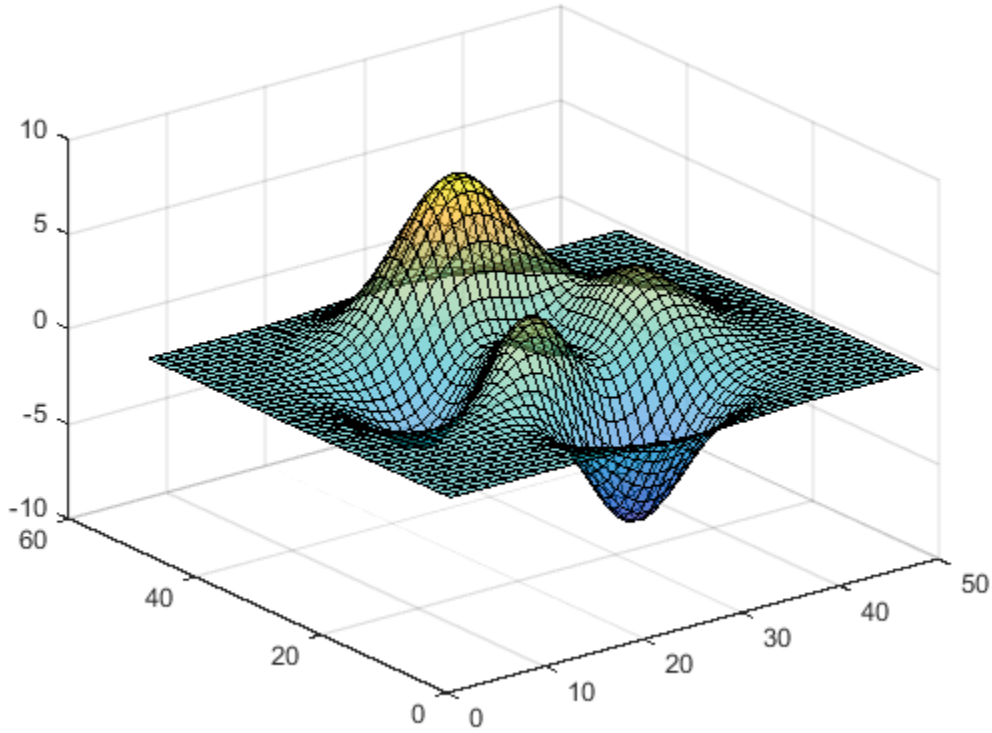
`alpha(alpha_data_mapping)` sets the `AlphaDataMapping` property of all image, patch, and surface objects in the current axes. You can set `alpha_data_mapping` to

`'scaled'`	Set the `AlphaDataMapping` property to `scaled`.
`'direct'`	Set the `AlphaDataMapping` property to `direct`.
`'none'`	Set the `AlphaDataMapping` property to `none`.

Examples

Create a surface plot and change its transparency using `alpha`:

```
surf(peaks)
alpha(0.5)
```



See Also

`alim` | `alphamap`

Introduced before R2006a

alphamap

Specify figure alphamap (transparency)

Syntax

```
alphamap(alpha_map)
alphamap('parameter')
alphamap('parameter',length)
alphamap('parameter',delta)
alphamap(figure_handle,...)
alphamap(axes_handle,...)
alpha_map = alphamap
alpha_map = alphamap(figure_handle)
alpha_map = alphamap(axes_handle)
alpha_map = alphamap('parameter')
```

Description

`alphamap(alpha_map)` sets the `AlphaMap` of the current figure to the specified `m`-by-1 array of alpha values, `alpha_map`.

`alphamap('parameter')` creates a new alphamap or modifies the current alphamap. You can specify the following parameters:

- `'default'` — Set the `AlphaMap` property to the figure's default alphamap.
- `'rampup'` — Create a linear alphamap with increasing opacity (default `length` equals the current alphamap length).
- `'rampdown'` — Create a linear alphamap with decreasing opacity (default `length` equals the current alphamap length).
- `'vup'` — Create an alphamap that is opaque in the center and becomes more transparent linearly towards the beginning and end (default `length` equals the current alphamap length).
- `'vdown'` — Create an alphamap that is transparent in the center and becomes more opaque linearly towards the beginning and end (default `length` equals the current alphamap length).

- `'increase'` — Modify the alphamap making it more opaque (default `delta` is `.1`, added to the current values).
- `'decrease'` — Modify the alphamap making it more transparent (default `delta` is `.1`, subtracted from the current values).
- `'spin'` — Rotate the current alphamap (default `delta` is 1; `delta` must be an integer).

`alphamap('parameter',length)` creates a new alphamap with the length specified by the integer `length` (used with parameters `'rampup'`, `'rampdown'`, `'vup'`, `'vdown'`).

`alphamap('parameter',delta)` modifies the existing alphamap using the value specified by the integer `delta` (used with parameters `'increase'`, `'decrease'`, `'spin'`).

`alphamap(figure_handle,...)` performs the operation on the alphamap of the figure identified by `figure_handle`.

`alphamap(axes_handle,...)` performs the operation on the alphamap of the axes identified by `axes_handle`.

`alpha_map = alphamap` returns the current alphamap.

`alpha_map = alphamap(figure_handle)` returns the current alphamap from the figure identified by `figure_handle`.

`alpha_map = alphamap(axes_handle)` returns the current alphamap from the axes identified by `axes_handle`.

`alpha_map = alphamap('parameter')` returns the alphamap modified by the `parameter`, but does not set the `AlphaMap` property.

Examples

Change Alphamap for Surface Plot

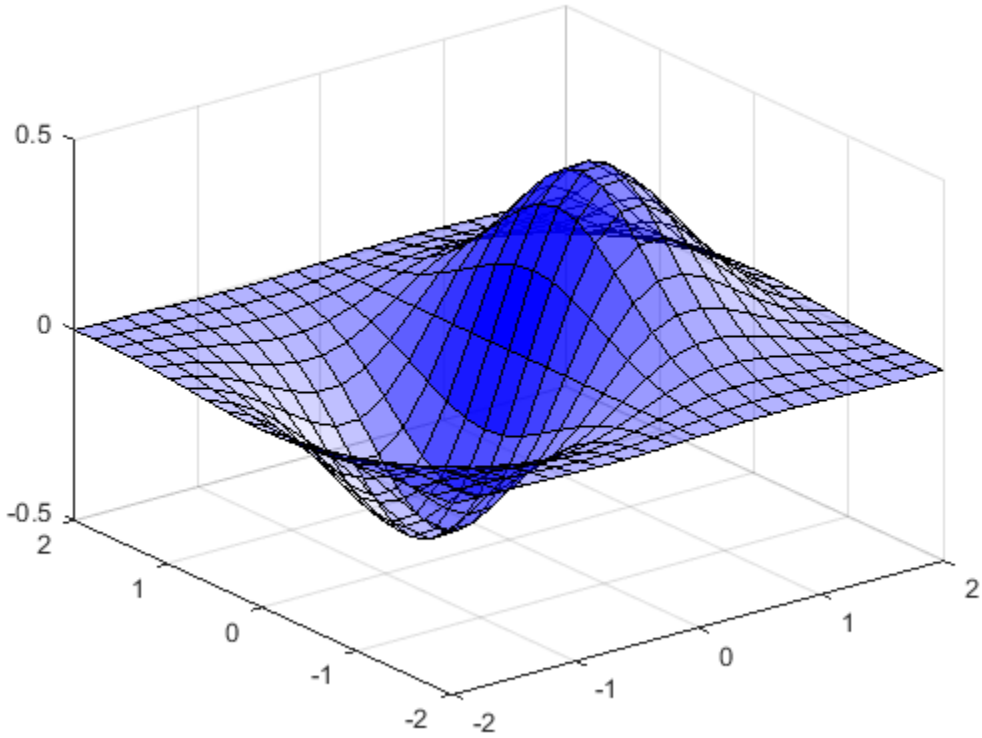
Create a surface plot and change the alphamap.

```
[x,y] = meshgrid([-2:.2:2]);
```

```
z = x.*exp(-x.^2-y.^2);
```

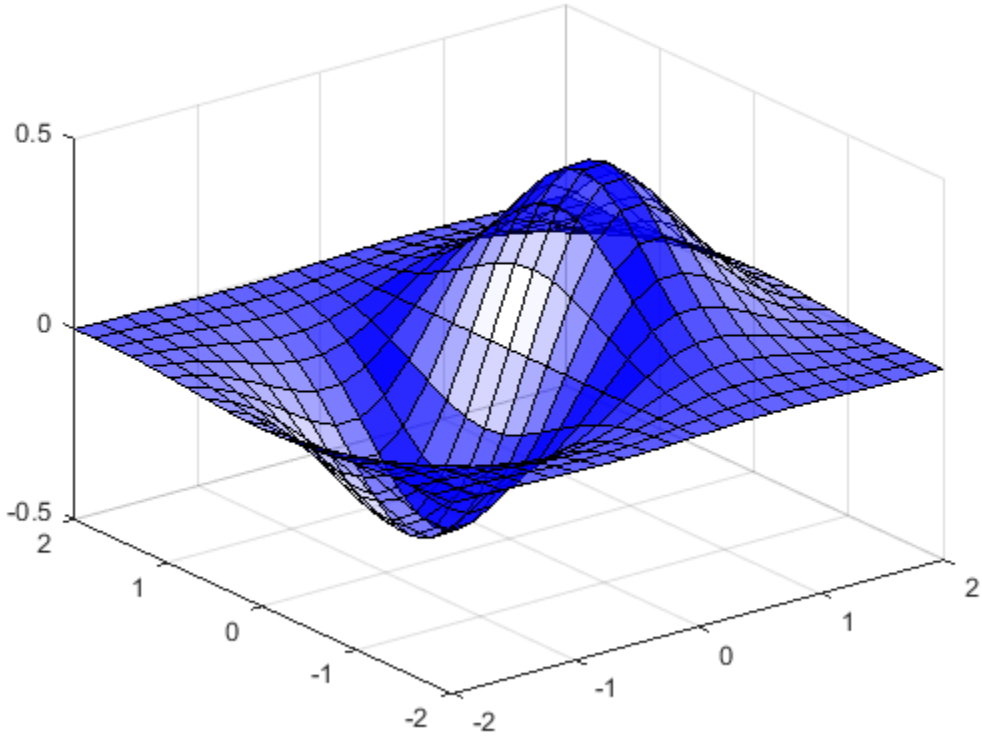
Plot the data, using the gradient of `z` as the alphamap.

```
figure
surf(x,y,z+.001,'FaceAlpha','flat',...
    'AlphaDataMapping','scaled',...
    'AlphaData',gradient(z),...
    'FaceColor','blue')
```



Change the alphamap to be opaque at the middle and transparent towards the ends.

```
alphamap('vup')
```



- “Making Objects Transparent”

See Also

`alim` | `alpha`

Introduced before R2006a

alphaShape

Polygons and polyhedra from points in 2-D and 3-D

Syntax

```
shp = alphaShape(x,y)
shp = alphaShape(x,y,z)
shp = alphaShape(P)
shp = alphaShape(___,a)
shp = alphaShape(___,Name,Value)
```

Description

`shp = alphaShape(x,y)` creates a 2-D alpha shape of the points `(x,y)` using the default alpha radius. The default alpha radius produces the tightest fitting alpha shape, which encloses all of the points.

`shp = alphaShape(x,y,z)` creates a 3-D alpha shape of the points `(x,y,z)` using the default alpha radius.

`shp = alphaShape(P)` specifies points `(x,y)` or `(x,y,z)` in the columns of matrix `P`.

`shp = alphaShape(___,a)` creates an alpha shape with alpha radius `a` using any of the arguments in the previous syntaxes.

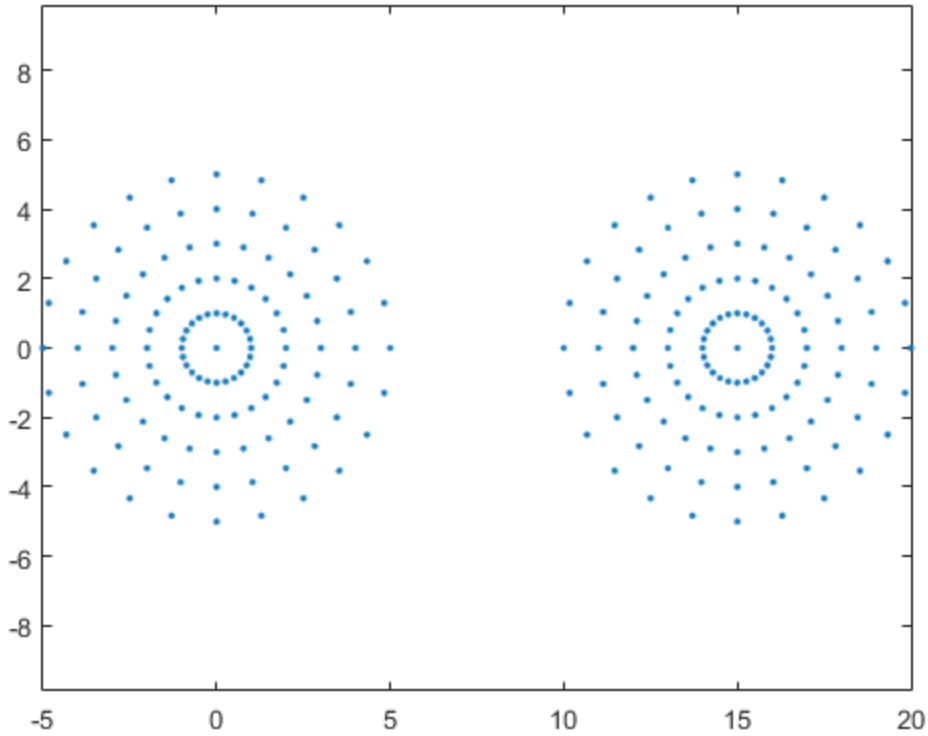
`shp = alphaShape(___,Name,Value)` uses additional options specified by one or more Name,Value pair arguments. For example, you can suppress interior holes or voids using `'HoleThreshold'`.

Examples

Alpha Shape from 2-D Point Cloud

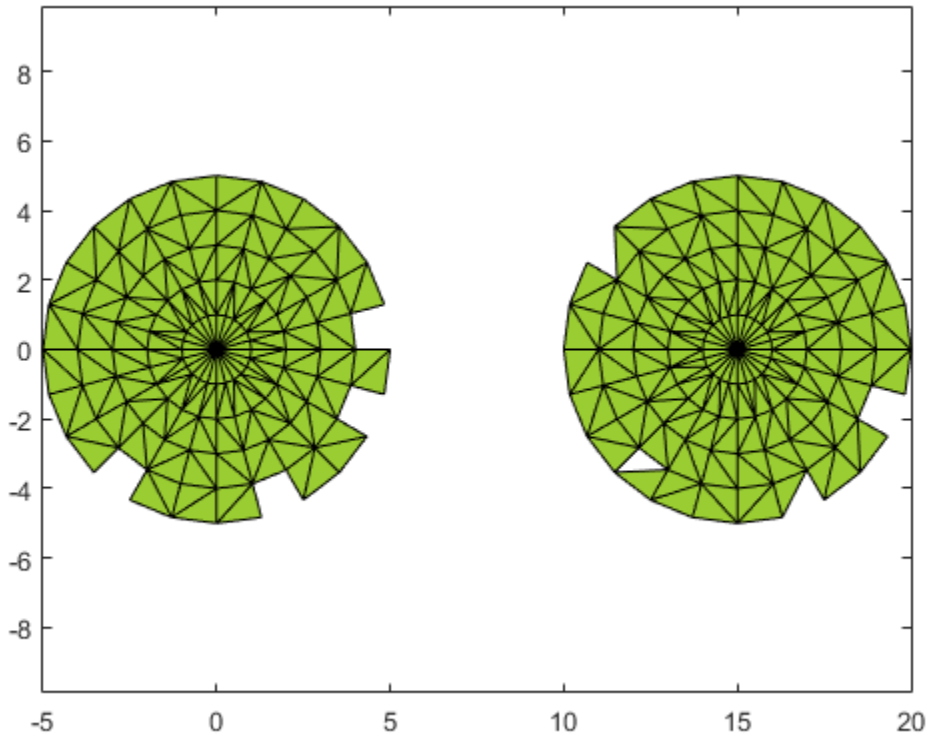
Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



Compute an alpha shape for the point set using the default alpha radius.

```
shp = alphaShape(x,y);
plot(shp)
```



Check the value of the default alpha radius.

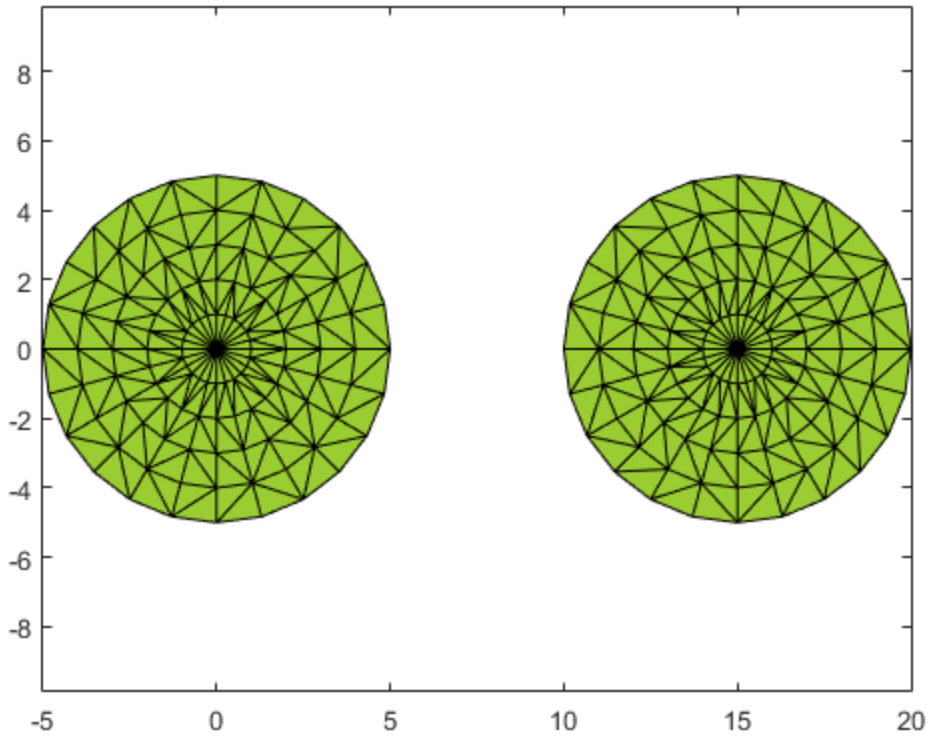
```
shp.Alpha
```

```
ans =

    0.7752
```

The default alpha radius results in an alpha shape with a jagged boundary. To better capture the boundary of the point set, try a larger alpha radius.

Compute an alpha shape using an alpha value of 2.5.

```
shp.Alpha = 2.5;
plot(shp)
```

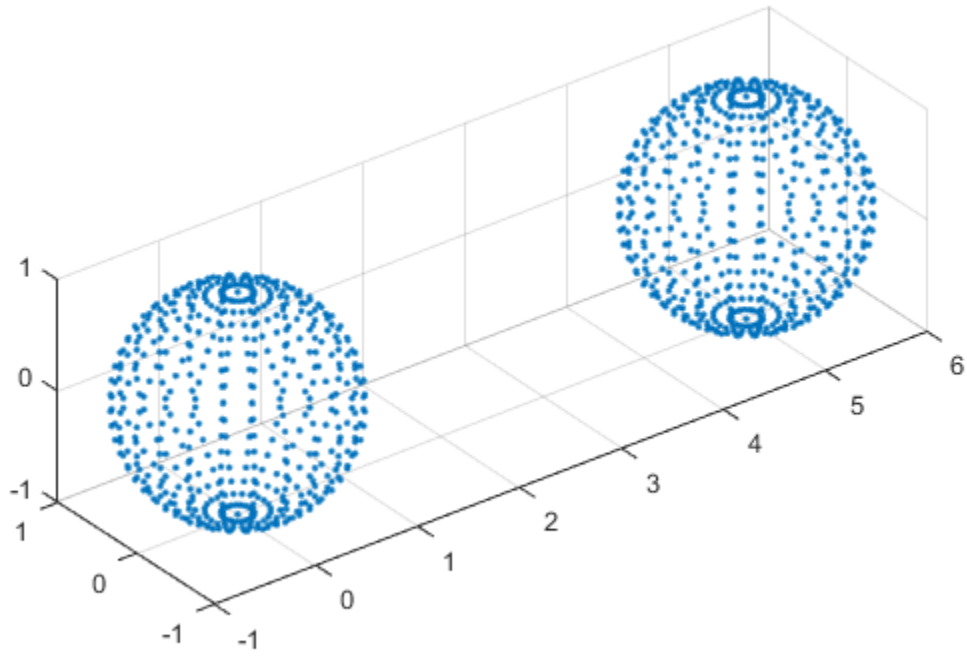


Alpha Shape from 3-D Point Cloud

Create and plot a set of 3-D points.

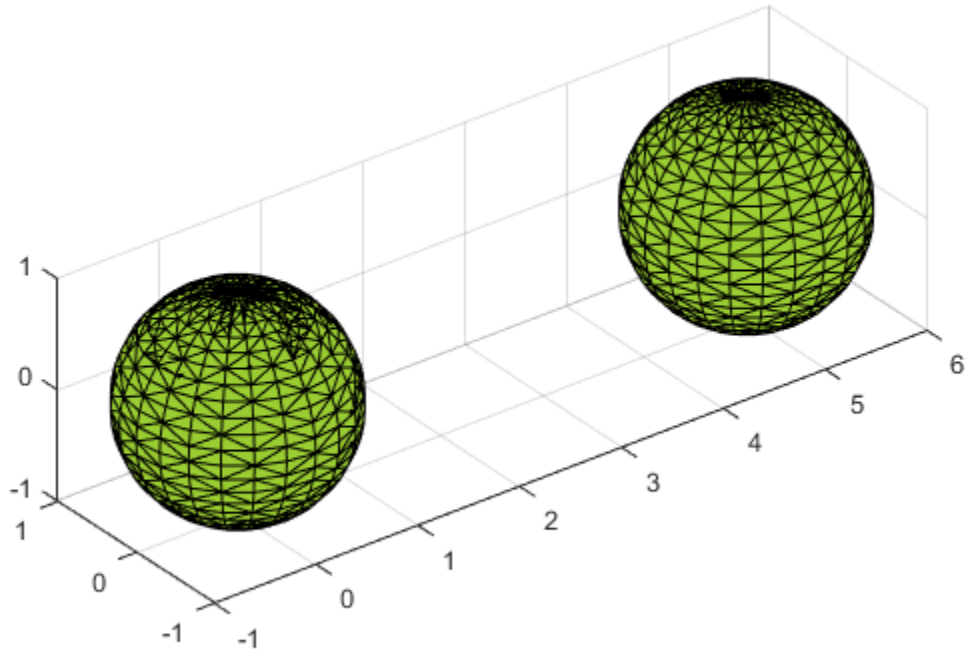
```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
```

```
P = unique(P,'rows');
plot3(P(:,1),P(:,2),P(:,3),'.')
axis equal
grid on
```



Compute a 3-D alpha shape using an alpha radius of 1.

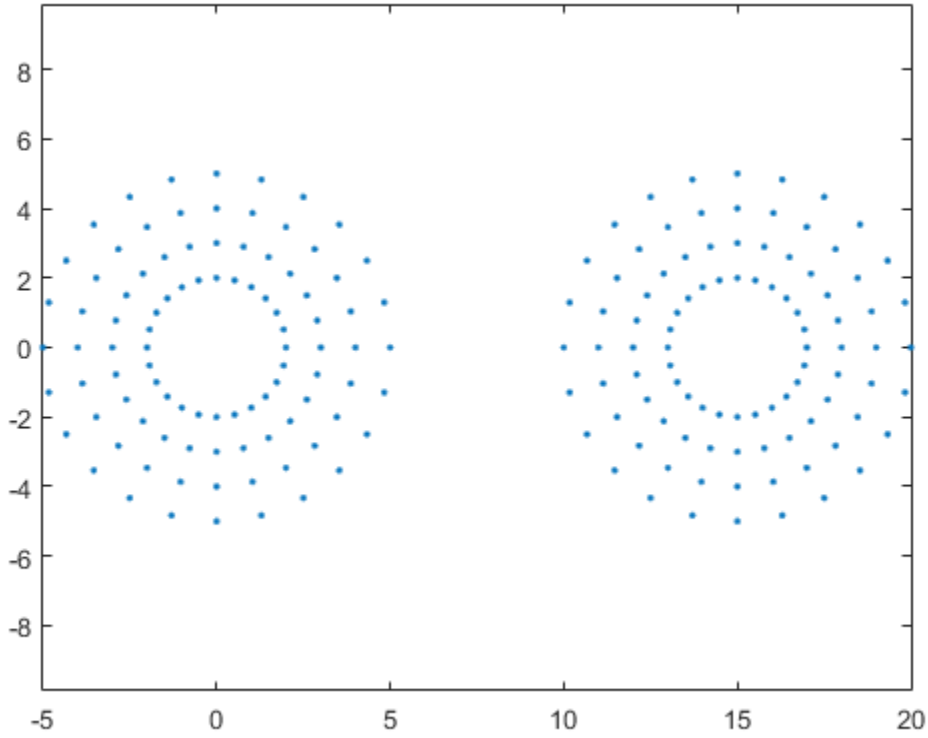
```
shp = alphaShape(P(:,1),P(:,2),P(:,3),1);
plot(shp)
axis equal
```



Fill Holes in 2-D Alpha Shape

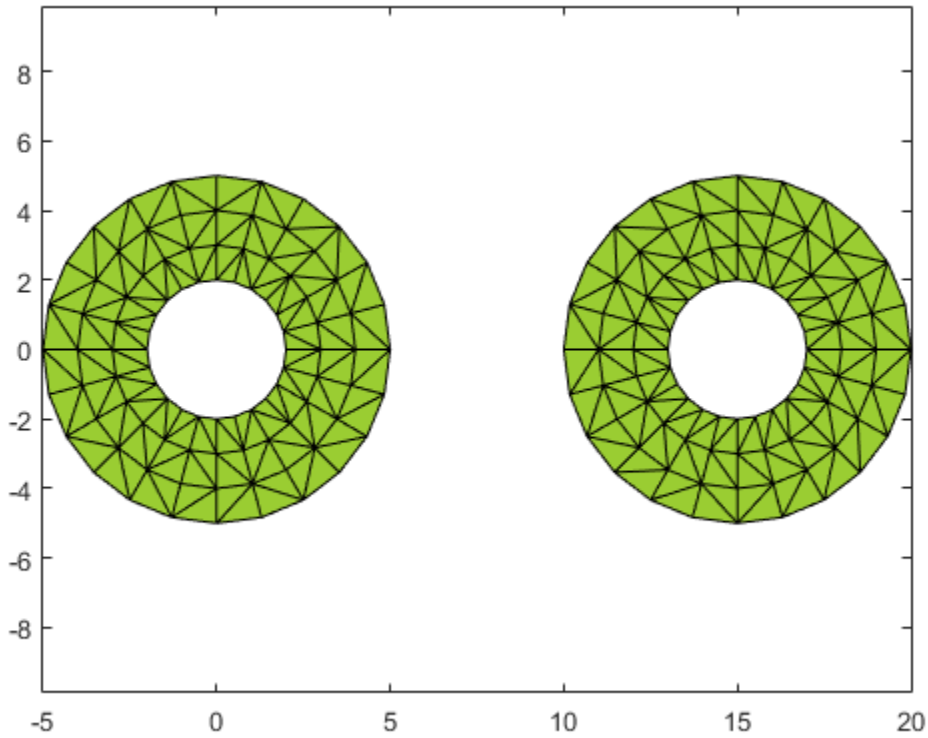
Create and plot a 2-D set of points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(2:5), numel(cos(th)*(2:5)),1);];
y1 = [reshape(sin(th)*(2:5), numel(sin(th)*(2:5)),1);];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



Compute an alpha shape for the point set using an alpha radius of 1.

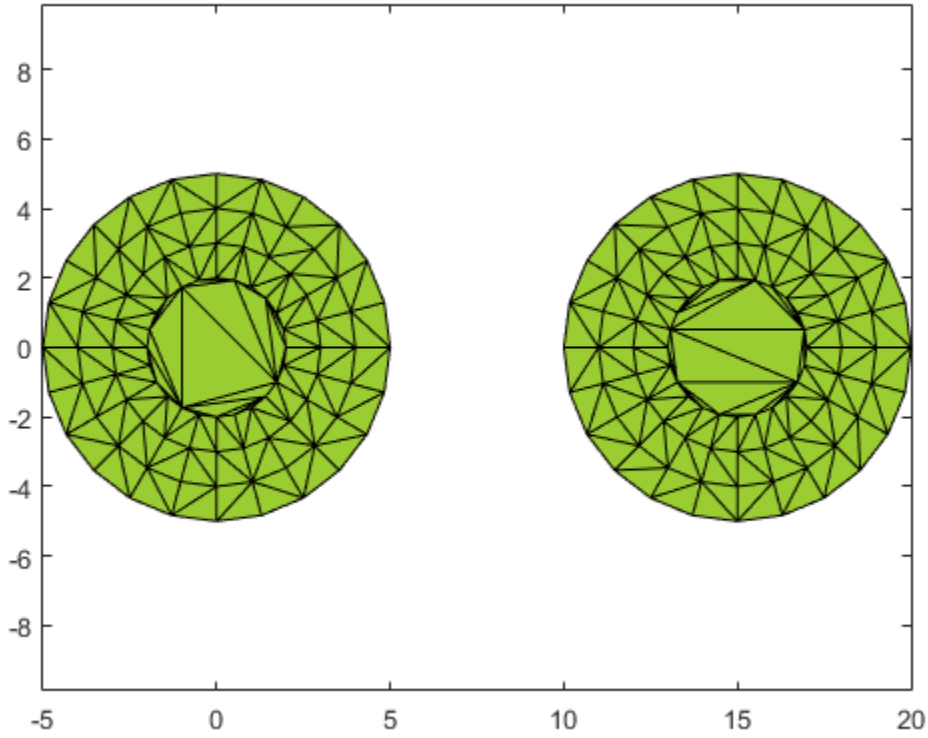
```
shp = alphaShape(x,y,1);
plot(shp)
```



An alpha radius of 1 results in an alpha shape with two regions containing holes. To suppress the small holes in the alpha shape, you can specify a `HoleThreshold` by estimating the area of the largest hole to fill. To fill all holes in the shape, you can assign an arbitrarily large value to `HoleThreshold`.

Create a new alpha shape that supresses the holes by specifying a `HoleThreshold` of 15.

```
shp = alphaShape(x,y,1,'HoleThreshold',15);
plot(shp)
```

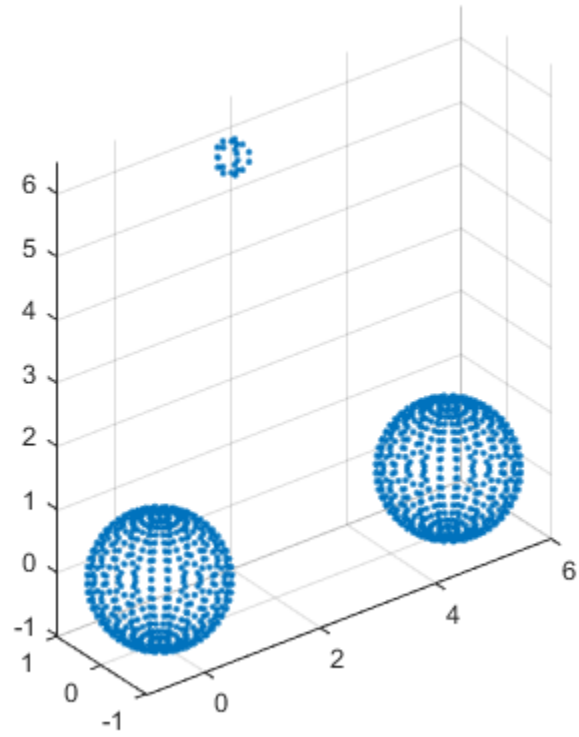


Discard Small Regions of 3-D Alpha Shape

Create and plot a set of 3-D points.

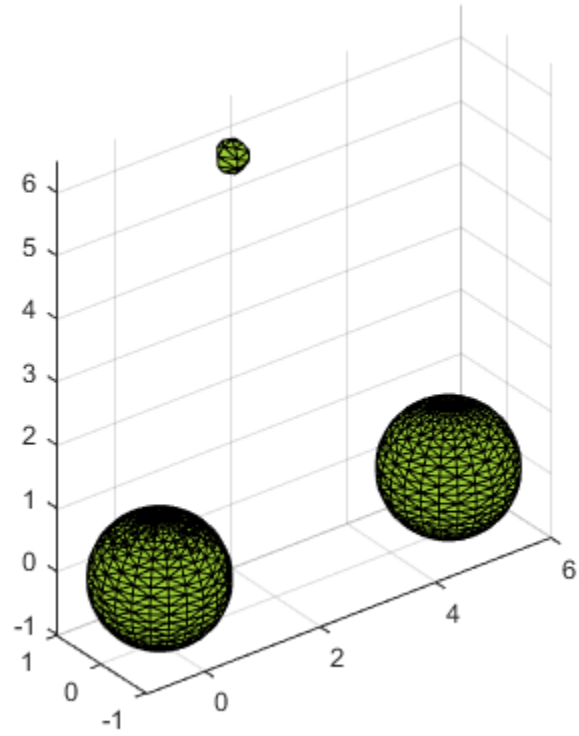
```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
[x3, y3, z3] = sphere(5);
x3 = x3(:)+5;
y3 = y3(:);
z3 = z3(:)+25;
P = [x1 y1 z1; x2 y1 z1; 0.25*x3 0.25*y3 0.25*z3];
```

```
P = unique(P,'rows');
plot3(P(:,1),P(:,2),P(:,3),'.')
axis equal
grid on
```



Compute an alpha shape for the point set using an alpha radius of 1.

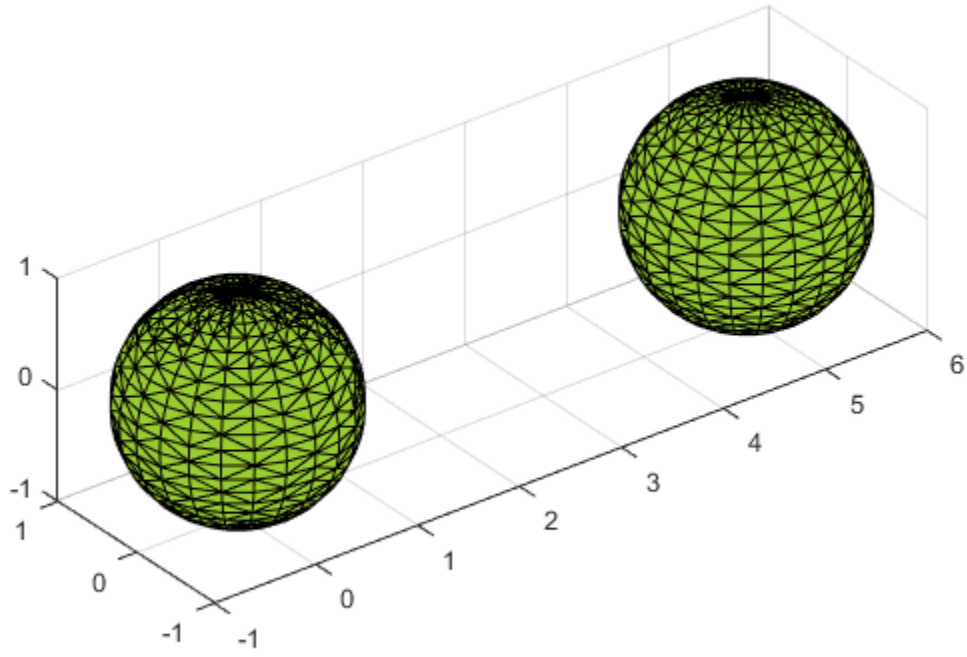
```
shp = alphaShape(P,1);
plot(shp)
axis equal
```



In this case, the alpha shape produces a small region above the two equal-sized spheres. To suppress this region, you can specify a `RegionThreshold` by estimating its volume.

Specify a `RegionThreshold` of 2. The resulting shape contains only the two larger regions.

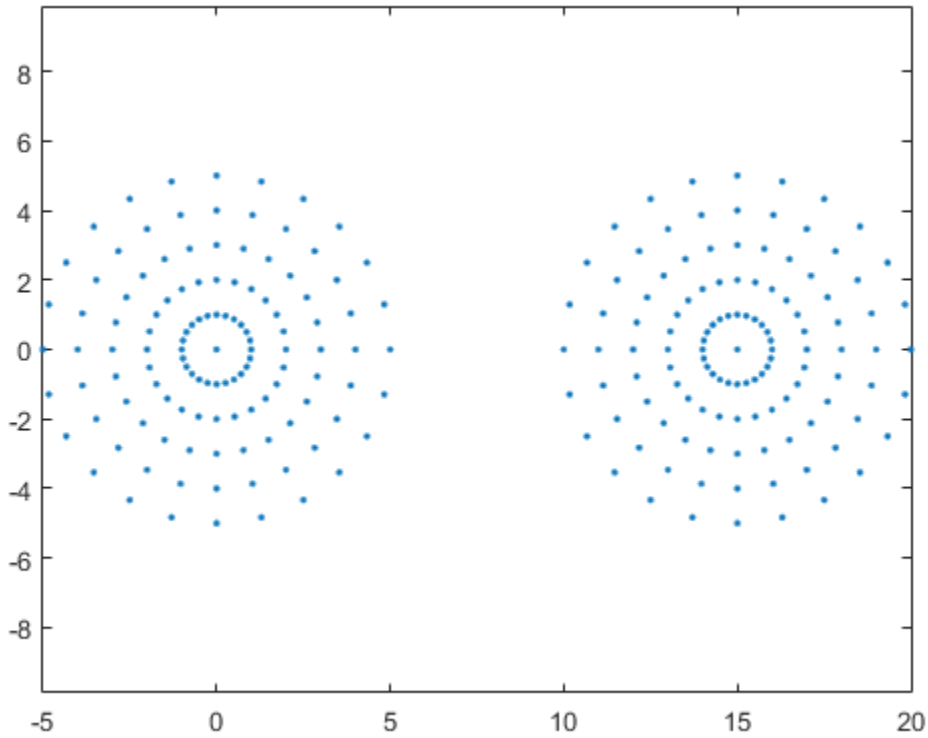
```
shp.RegionThreshold = 2;
plot(shp)
axis equal
```



Modify Points of 2-D Alpha Shape

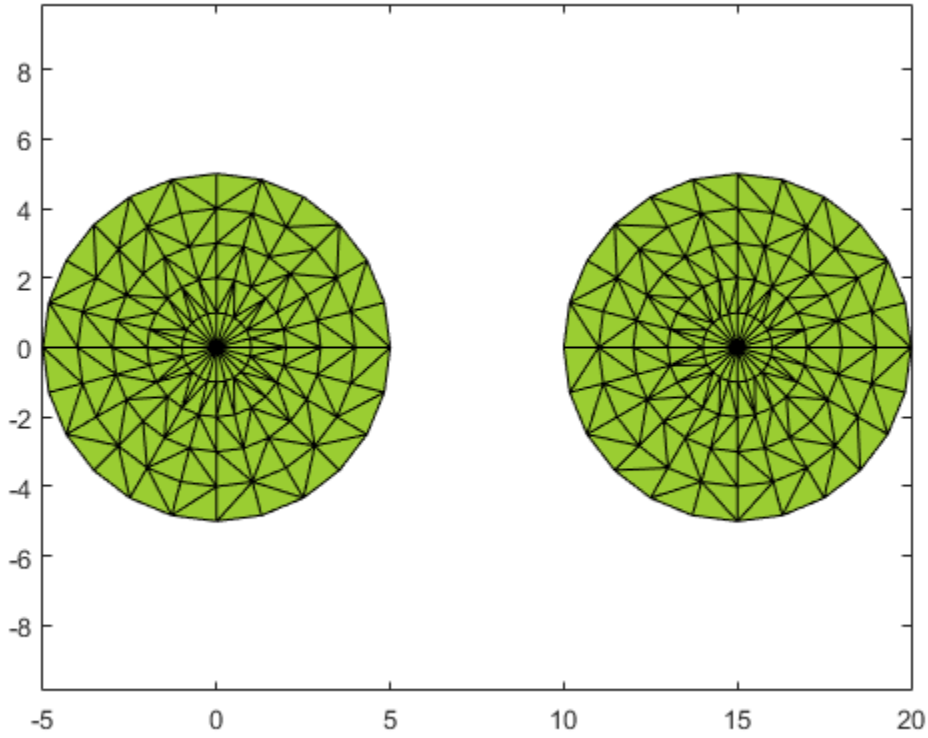
Create and plot a 2-D set of points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



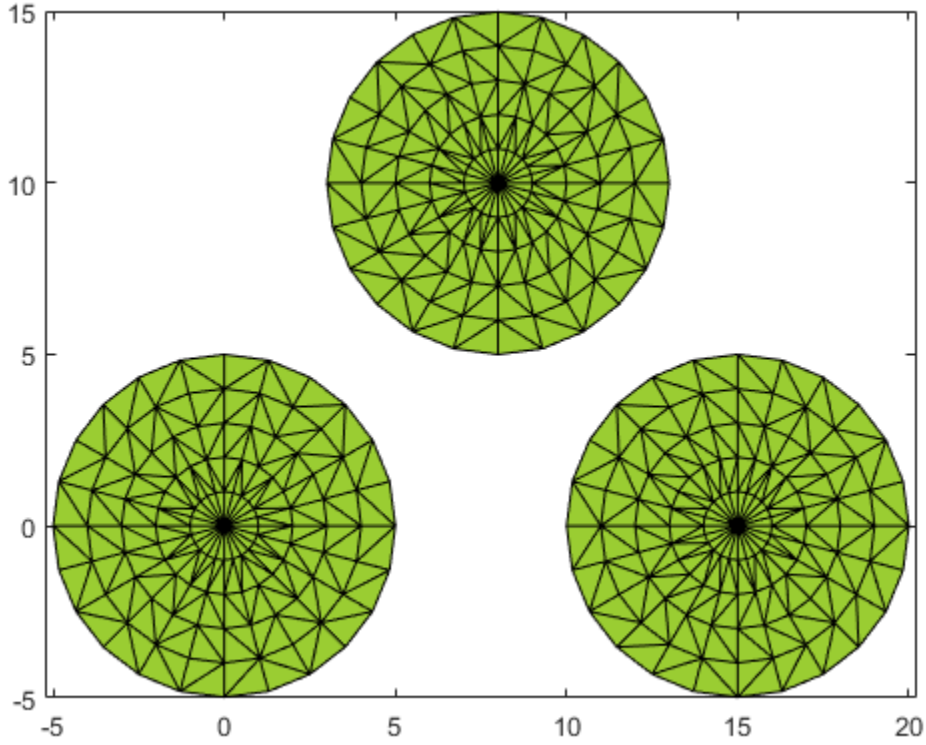
Compute an alpha shape for the point set using an alpha radius of 1. The resulting alpha shape has two regions.

```
shp = alphaShape(x,y,1);
plot(shp)
```



Now add a third region to the alpha shape by adding new points directly to the `shp.Points` matrix.

```
x3 = x1+8;
y3 = y1+10;
shp.Points(end+1,:) = [x3 y3];
plot(shp)
```



Input Arguments

x — x-coordinates of points

column vector

x-coordinates of points, specified as a column vector.

Data Types: `double`

y — y-coordinates of points

column vector

y-coordinates of points, specified as a column vector.

Data Types: `double`

z — z-coordinates of points

column vector

z-coordinates of points, specified as a column vector.

Data Types: `double`

P — Point coordinates

matrix with two columns | matrix with three columns

Point coordinates, specified as a matrix with two columns (for a 2-D alpha shape) or a matrix with three columns (for a 3-D alpha shape).

- For 2-D, the columns of `P` represent `x` and `y` coordinates, respectively.
- For 3-D, the columns of `P` represent `x`, `y`, and `z` coordinates, respectively.

Data Types: `double`

a — Alpha radius

nonnegative scalar

Alpha radius, specified as a nonnegative scalar. The default alpha radius is `a = criticalAlpha(shp,'all-points')`, which is the smallest alpha radius that produces an alpha shape that encloses all points.

Specify `a = criticalAlpha(shp,'one-region')` to use the smallest alpha radius that produces an alpha shape with only one region.

The extreme values of `a` are

- `Inf`, where `alphaShape` produces the convex hull.
- `0`, where `alphaShape` produces an empty alpha shape.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `shp = alphaShape(...,'HoleThreshold',10)`

`'HoleThreshold'` — Maximum area or volume of interior holes or voids to fill in

0 (default) | finite, nonnegative scalar

Maximum area or volume of interior holes or voids to fill in, specified as a finite, nonnegative scalar.

- For 2-D, `HoleThreshold` specifies the maximum area of interior holes to fill in.
- For 3-D, `HoleThreshold` specifies the maximum volume of interior voids to fill in. Holes extending completely through the alpha shape cannot be filled in.

When you specify both a `'HoleThreshold'` and a `'RegionThreshold'`, the application of the thresholds is order dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

`'RegionThreshold'` — Maximum area or volume of regions to suppress

0 (default) | finite, nonnegative scalar

Maximum area (2-D) or volume (3-D) of regions to suppress, specified as a finite, nonnegative scalar.

When you specify a `'HoleThreshold'` and a `'RegionThreshold'`, the application of the thresholds is order dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

Output Arguments

`shp` — 2-D or 3-D alpha shape

alphaShape object

2-D or 3-D alpha shape. For more information, see Using alphaShape Objects.

The `alphaShape` function returns regularized alpha shapes:

- In 2-D, `shp` represents a polygon. The polygon has no isolated points or edges, nor does it have dangling edges.
- In 3-D, `shp` represents a polyhedron. The polyhedron has the previously stated polygon traits, but it additionally does not have isolated faces or dangling faces.

See Also

`boundary` | `convhull` | `criticalAlpha` | `delaunayTriangulation` | `triangulation` | `trisurf`

Introduced in R2014b

alphaSpectrum

Alpha values giving distinct alpha shapes

Syntax

```
a = alphaSpectrum(shp)
```

Description

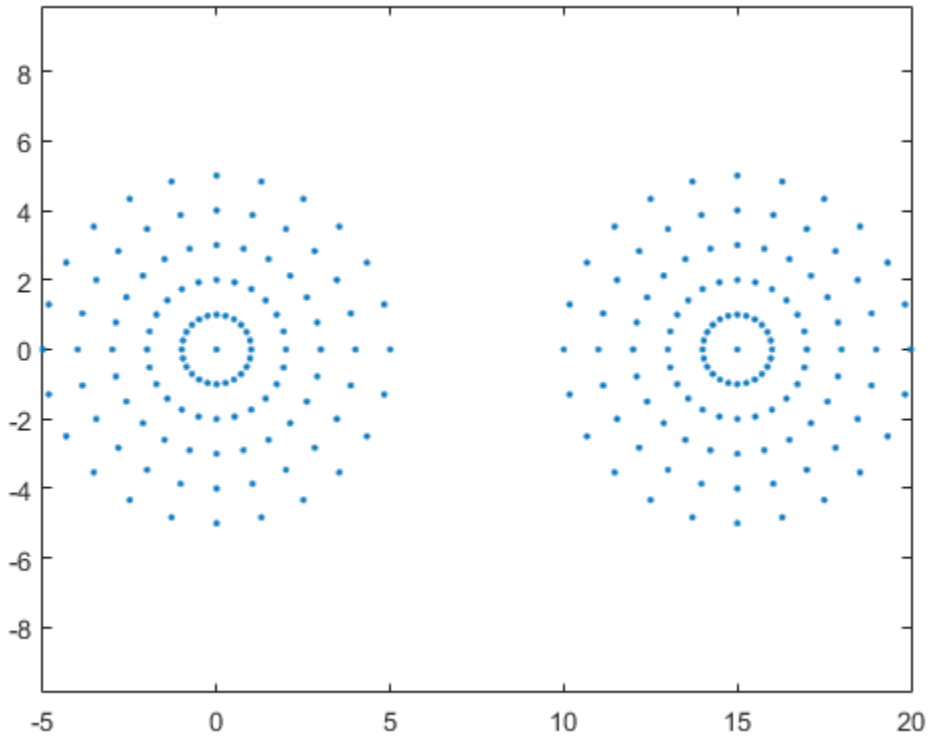
`a = alphaSpectrum(shp)` returns the values of the alpha radius that produce distinct alpha shapes. `a` is in descending sorted order. Each element in `a` represents a value of the alpha radius that results in a distinct shape. The length of `a` is equal to the number of unique shapes. Values of alpha that lie between the values in `a` do not produce unique alpha shapes.

Examples

Find Alpha Spectrum for 2-D Point Cloud

Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



Create an alpha shape for the point cloud using the default alpha radius.

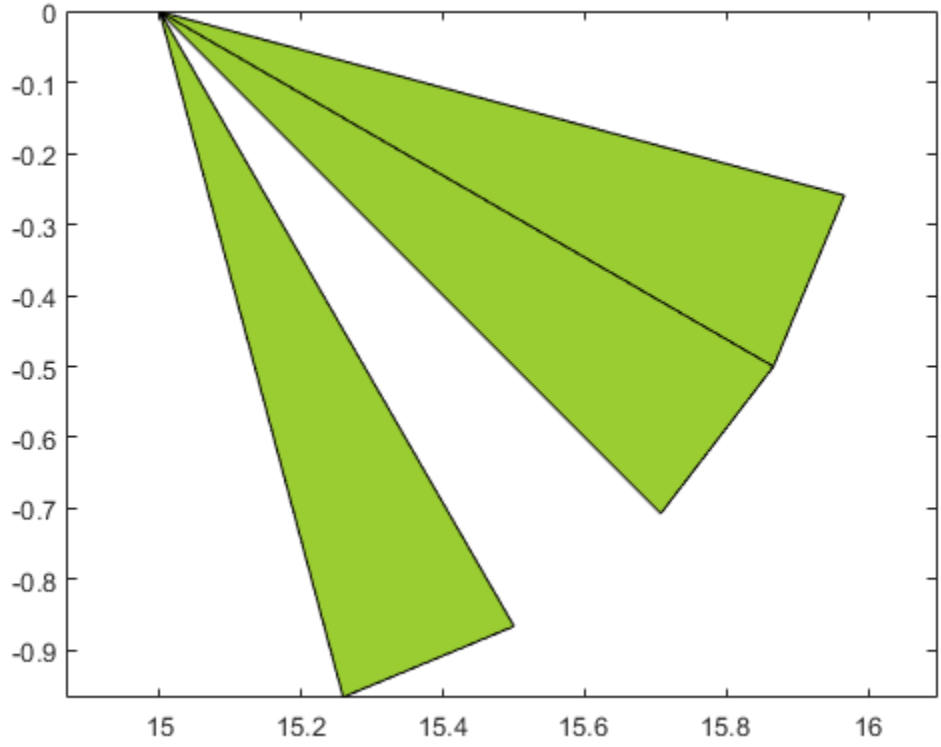
```
shp = alphaShape(x,y);
```

Find the spectrum of critical alpha values that produces unique alpha shapes for the point cloud.

```
alphaspec = alphaSpectrum(shp);
```

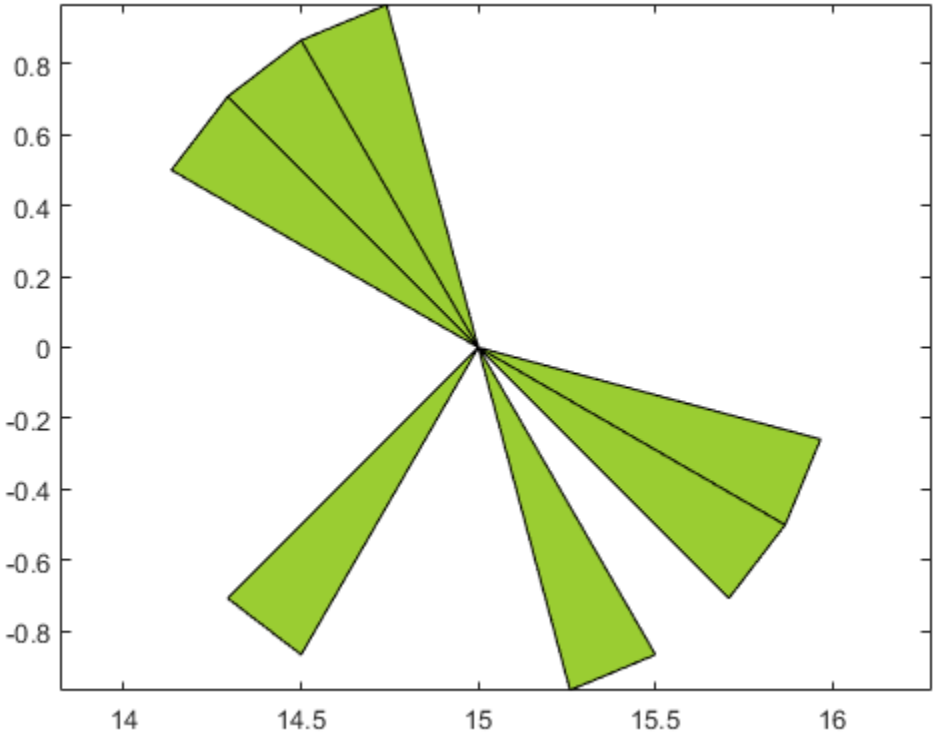
Plot the alpha shape corresonding to the smallest alpha value in the spectrum.

```
shp.Alpha = alphaspec(length(alphaspec));
plot(shp)
```



Compare this alpha shape to the one produced by the next smallest critical alpha value. The alpha shapes are unique.

```
shp.Alpha = alphaspec(length(alphaspec)-1);
plot(shp)
```



Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

Output Arguments

a — Alpha values for distinct alpha shapes

column vector

Alpha values for distinct alpha shapes, returned as a column vector in descending sorted order.

More About

- Using alphaShape Objects

See Also

`alphaShape` | `plot`

Introduced in R2014b

alphaTriangulation

Triangulation that fills alpha shape

Syntax

```
tri = alphaTriangulation(shp)
tri = alphaTriangulation(shp,RegionID)

[tri,P] = alphaTriangulation( ___ )
```

Description

`tri = alphaTriangulation(shp)` returns a triangulation that defines the domain of the alpha shape. Each row in `tri` specifies a triangle or tetrahedron defined by vertex IDs (the row numbers of the `shp.Points` matrix).

`tri = alphaTriangulation(shp,RegionID)` returns a triangulation for a region of the alpha shape. `RegionID` is the ID for the region and $1 \leq$ `RegionID` $\leq$ `numRegions(shp)`.

`[tri,P] = alphaTriangulation( ___ )` also returns a matrix of vertex coordinates, `P`, using any of the previous syntaxes.

Examples

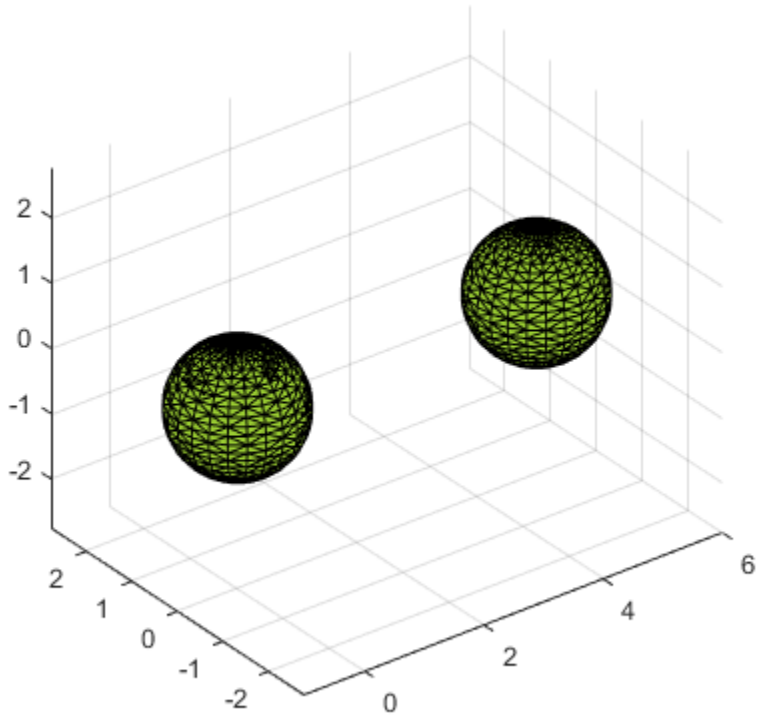
Compute Triangulation for 3-D Point Cloud

Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

Create and plot an alpha shape for the point cloud using an alpha radius of 1.

```
shp = alphaShape(P,1);
plot(shp)
```



Use `alphaTriangulation` to recover the triangulation that defines the domain of the alpha shape.

```
tri = alphaTriangulation(shp);
```

Find the total number of tetrahedra that make up the alpha shape.

```
numtetrahedra = size(tri,1)
```

```
numtetrahedra =

        3729
```

Input Arguments

`shp` — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

`RegionID` — ID number for a region in the alpha shape

positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

Output Arguments

`tri` — Triangulation

matrix

Triangulation, returned as a matrix. `tri` is of size `mtri`-by-`nv`, where `mtri` is the number of triangles or tetrahedra in the alpha shape and `nv` is the number of vertices. The value of `nv` is `3` for 2-D alpha shapes and `4` for 3-D alpha shapes.

`P` — Vertex coordinates

matrix

Vertex coordinates, returned as a matrix. `P` is of size `N`-by-`dim`, where `N` is the number of points in the alpha shape and `dim` is either `2` or `3` (for either a 2-D or 3-D alpha shape).

More About

- Using alphaShape Objects

See Also

`alphaShape` | `plot` | `triangulation` | `triplot`

Introduced in R2014b

area

Area of 2-D alpha shape

Syntax

```
A = area(shp)
A = area(shp,RegionID)
```

Description

`A = area(shp)` returns the area of 2-D alpha shape `shp`.

`A = area(shp,RegionID)` returns the area of a region of the alpha shape. `RegionID` is the ID for the region and $1 \leq$ `RegionID` $\leq$ `numRegions(shp)`.

Examples

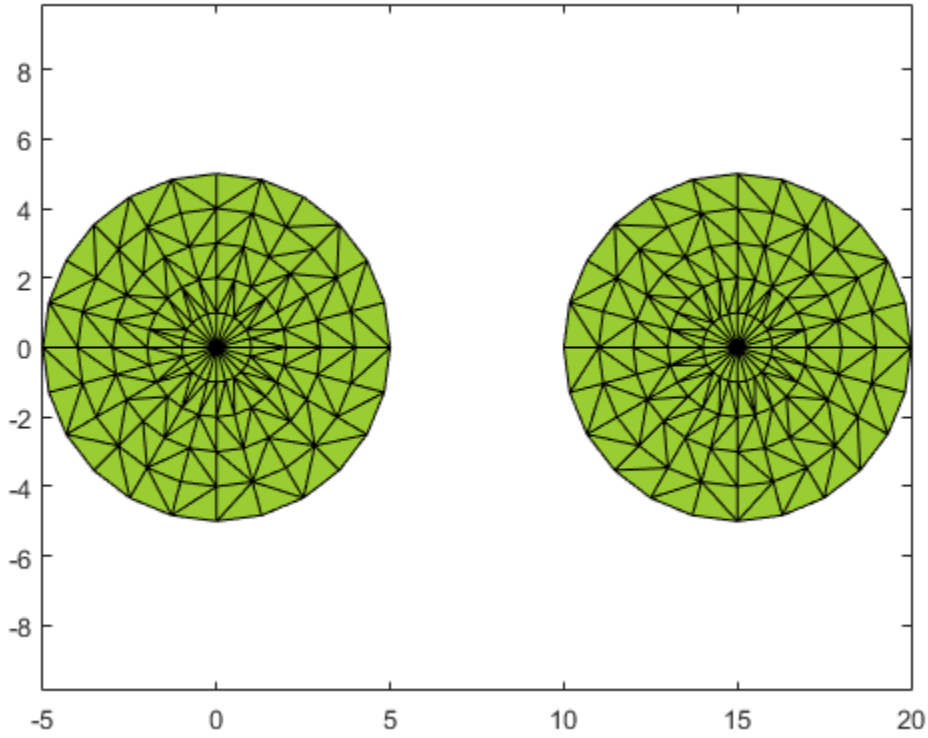
Find Area of 2-D Alpha Shape

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
```

Create and plot an alpha shape using an alpha radius of 2.5.

```
shp = alphaShape(x,y,2.5);
plot(shp)
```



Compute the area of the alpha shape.

```
totalarea = area(shp)
```

```
totalarea =

  155.2914
```

Compute the areas of each of the two regions separately.

```
regionareas = area(shp, 1:numRegions(shp))
```

```
regionareas =

   77.6457   77.6457
```

Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

RegionID — ID number for a region in the alpha shape

positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

More About

- Using alphaShape Objects

See Also

`alphaShape` | `criticalAlpha` | `perimeter` | `volume`

Introduced in R2014b

boundaryFacets

Boundary facets of alpha shape

Syntax

```
bf = boundaryFacets(shp)
bf = boundaryFacets(shp,RegionID)

[bf,P] = boundaryFacets( ___ )
```

Description

`bf = boundaryFacets(shp)` returns a matrix representing the facets that make up the boundary of the alpha shape. The facets represent edge segments in 2-D and triangles in 3-D. The vertices of the facets index into the `shp.Points` matrix.

`bf = boundaryFacets(shp,RegionID)` returns the boundary facets for a region of the alpha shape. `RegionID` is the ID for the region and $1 \le$ `RegionID` $\le$ `numRegions(shp)`.

`[bf,P] = boundaryFacets( ___ )` also returns a matrix of vertex coordinates, `P`, using any of the previous syntaxes.

Examples

Find Boundary of 3-D Alpha Shape

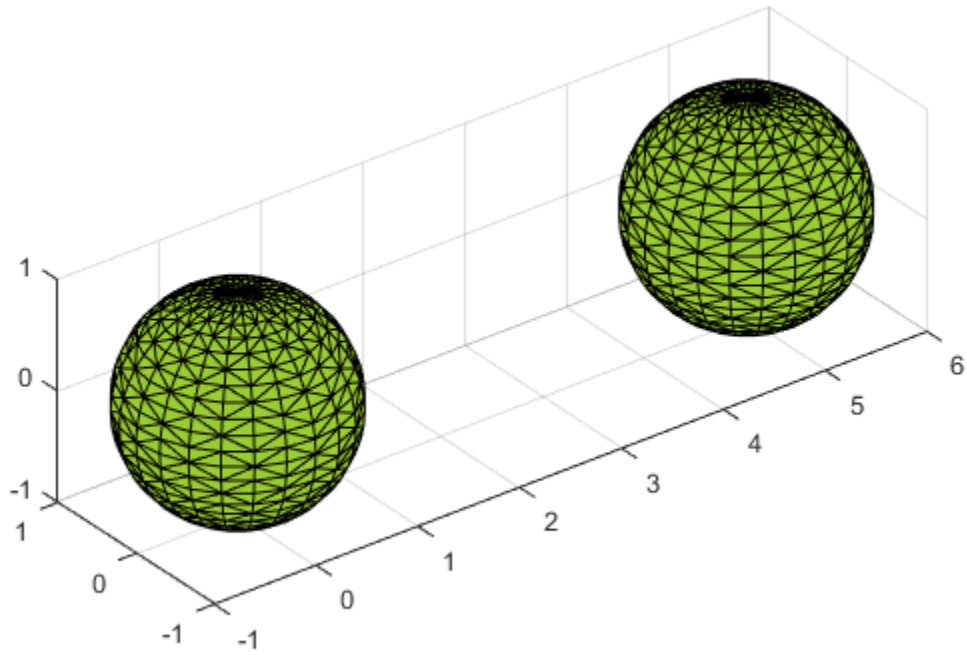
Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
```

```
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

Create and plot an alpha shape using an alpha radius of 1.5.

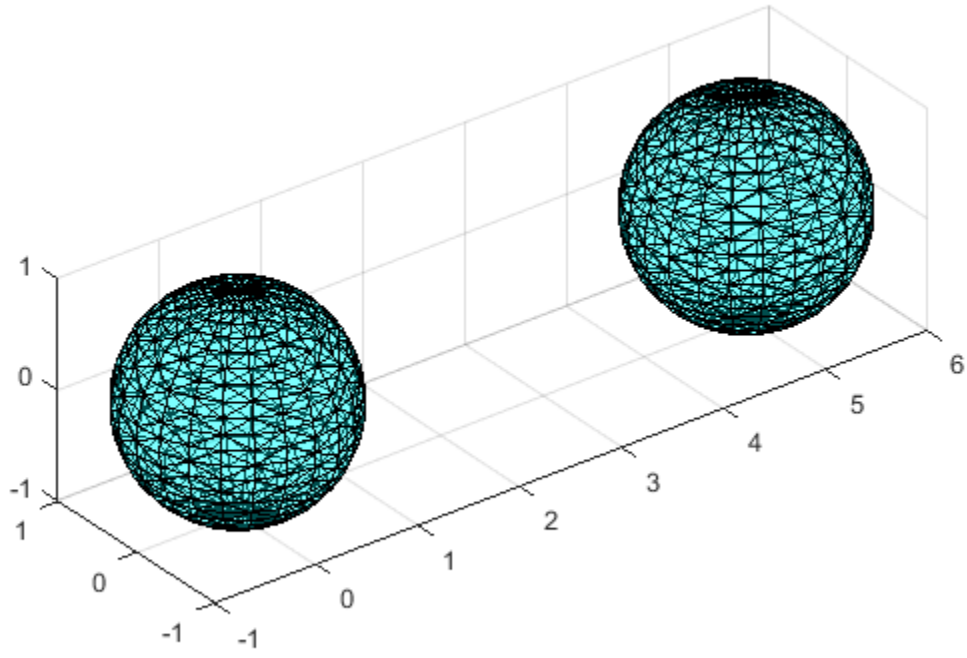
```
shp = alphaShape(P,1.5);
plot(shp)
axis equal
```



Compute and plot only the boundary of the alpha shape.

```
[tri, xyz] = boundaryFacets(shp);
trisurf(tri,xyz(:,1),xyz(:,2),xyz(:,3),...
    'FaceColor','cyan','FaceAlpha',0.3)
```

```
axis equal
```



Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

`RegionID` — ID number for a region in the alpha shape

positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

Output Arguments

`bf` — Boundary facets

matrix

Boundary facets, returned as a matrix. `bf` is of size `m`-by-`n`, where `m` is the number of boundary facets and `n` is the number of vertices per facet.

`P` — Vertex coordinates

matrix

Vertex coordinates, returned as a matrix. `P` is of size `N`-by-`dim`, where `N` is the number of points on the boundary of the the alpha shape and `dim` is either `2` or `3` (for either a 2-D or 3-D alpha shape).

More About

- Using alphaShape Objects

See Also

`alphaShape` | `plot` | `triangulation`

Introduced in R2014b

criticalAlpha

Alpha radius defining a critical transition in the shape

Syntax

```
a = criticalAlpha(shp,type)
```

Description

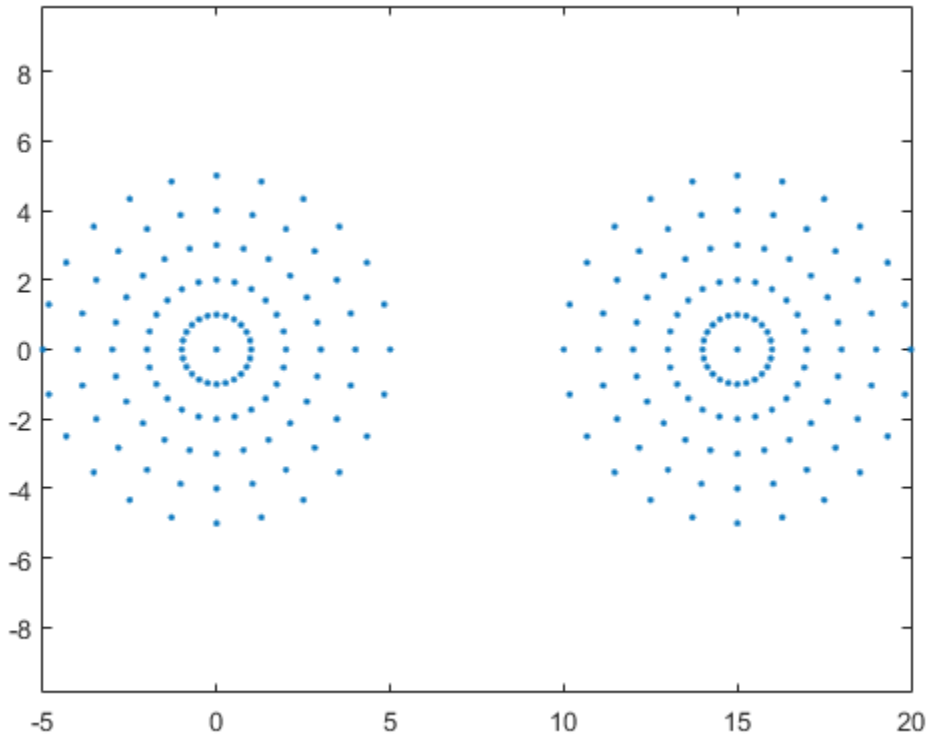
`a = criticalAlpha(shp,type)` returns the critical alpha radius that produces a notable transition in the alpha shape. Specifying `type` as `'all-points'` returns the smallest alpha radius producing an alpha shape that encloses all points. Specifying `type` as `'one-region'` returns the smallest alpha radius producing an alpha shape that encloses all points *and* has only one region.

Examples

Find Critical Alpha Values of 2-D Point Cloud

Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```

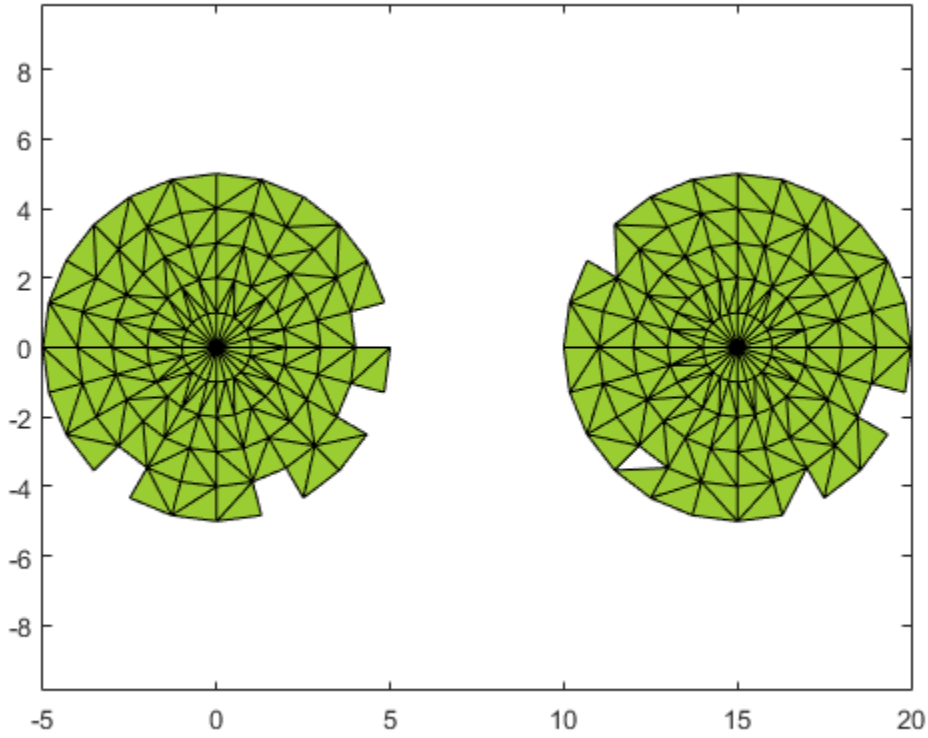


Create an alpha shape using the default alpha radius.

```
shp = alphaShape(x,y);
```

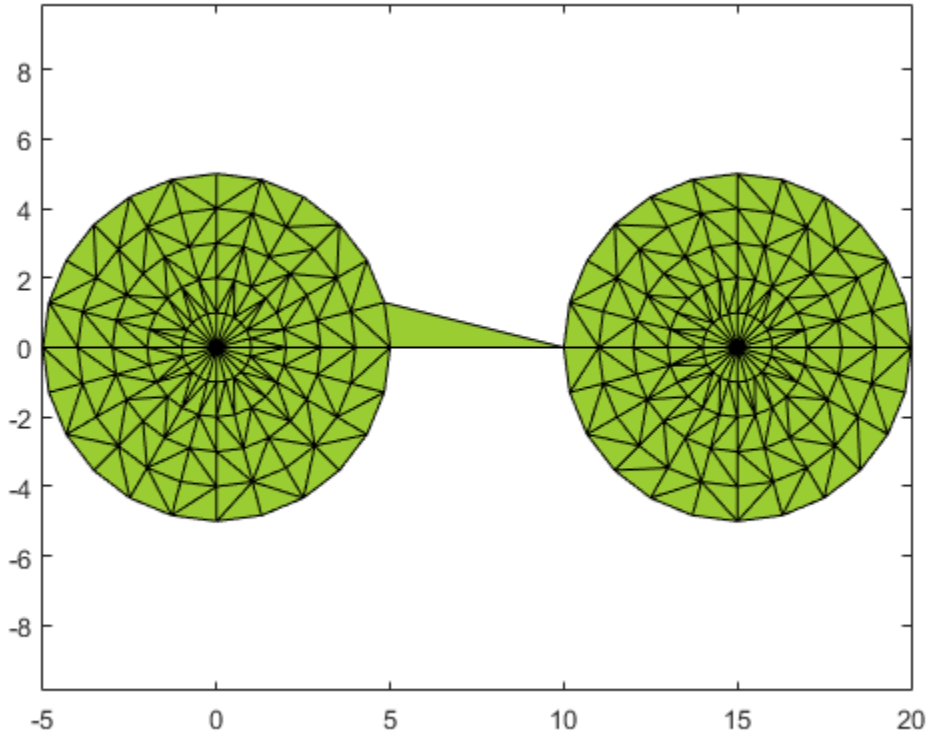
Compute the smallest alpha radius that produces an alpha shape enclosing all of the points and plot the corresponding alpha shape.

```
pc = criticalAlpha(shp,'all-points');
shp.Alpha = pc;
plot(shp)
```



Compute the smallest alpha radius that produces an alpha shape enclosing all of the points and having only one region.

```
shp = alphaShape(x,y);
pc = criticalAlpha(shp,'one-region');
shp.Alpha = pc;
plot(shp)
```



Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

`type` — Type of critical transition

`'all-points' | 'one-region'`

Type of critical transition, specified as either `'all-points'` or `'one-region'`.

- `'all-points'` corresponds to the smallest alpha radius producing an alpha shape that encloses all points.
- `'one-region'` corresponds to the smallest alpha radius producing an alpha shape that encloses all points *and* has only one region.

Data Types: `char`

Output Arguments

`a` — Critical alpha radius

scalar

Critical alpha radius, returned as a scalar. `a` is the value of the alpha radius that produces an alpha shape, which either encloses all points (if `type` is `'all-points'`), or encloses all points within a single region (if `type` is `'one-region'`).

After using `criticalAlpha` to find `a`, you can make the alpha radius of `shp` equal to `a` by typing `shp.Alpha = a`.

More About

- Using alphaShape Objects

See Also

`alphaShape` | `alphaSpectrum`

Introduced in R2014b

inShape

Determine if point is inside alpha shape

Syntax

```
tf = inShape(shp,qx,qy)
tf = inShape(shp,qx,qy,qz)
tf = inShape(shp,QP)
tf = inShape( ___ ,RegionID)
[tf,ID] = inShape( ___ )
```

Description

`tf = inShape(shp,qx,qy)` returns logical `1` (`true`) values for the 2-D query points `(qx,qy)` that are within 2-D alpha shape `shp`. Otherwise, `inShape` returns values of logical `0` (`false`). The `qx` and `qy` arguments are numeric arrays whose corresponding elements specify the `(x,y)` query point coordinates.

`tf = inShape(shp,qx,qy,qz)` tests whether the 3-D query points `(qx,qy,qz)` are within 3-D alpha shape `shp`.

`tf = inShape(shp,QP)` specifies the 2-D or 3-D query point coordinates in a matrix with 2 or 3 columns.

`tf = inShape( ___ ,RegionID)` tests whether the query points are within a specific region of the alpha shape, using any of the previous syntaxes. `RegionID` is the ID for the region and $1 \le$ `RegionID` $\le$ `numRegions(shp)`.

`[tf,ID] = inShape( ___ )` also returns the IDs for the regions in the alpha shape that contain the query points. `ID` is `NaN` for query points that are not in the alpha shape.

Examples

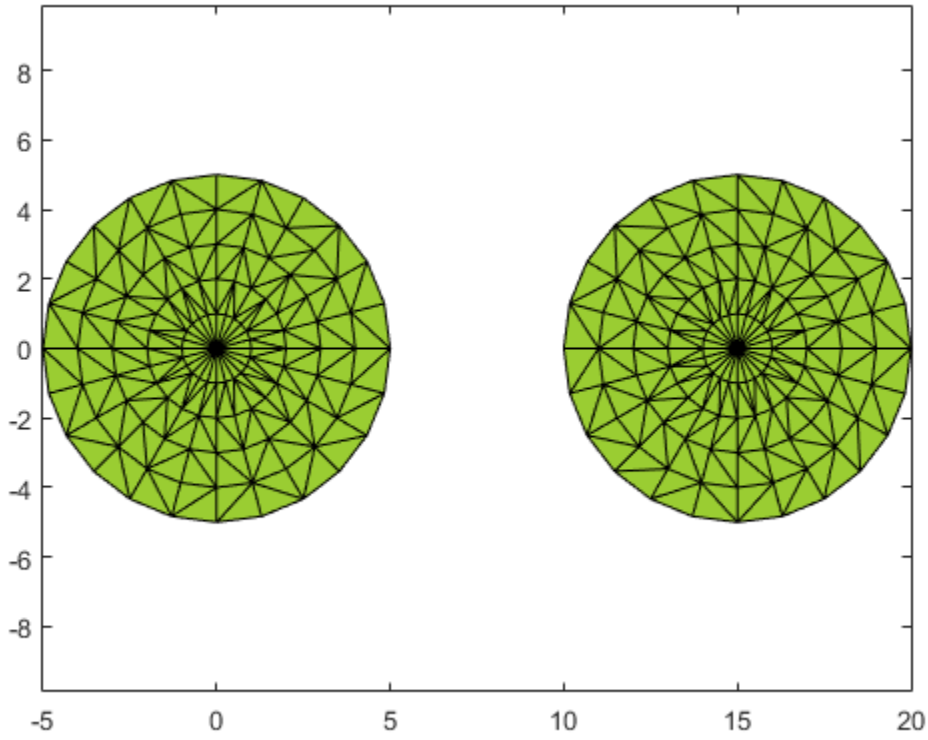
Query Points Inside and Outside of 2-D Alpha Shape

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
```

Create and plot an alpha shape using an alpha radius of 2.5.

```
shp = alphaShape(x,y,2.5);
plot(shp)
```

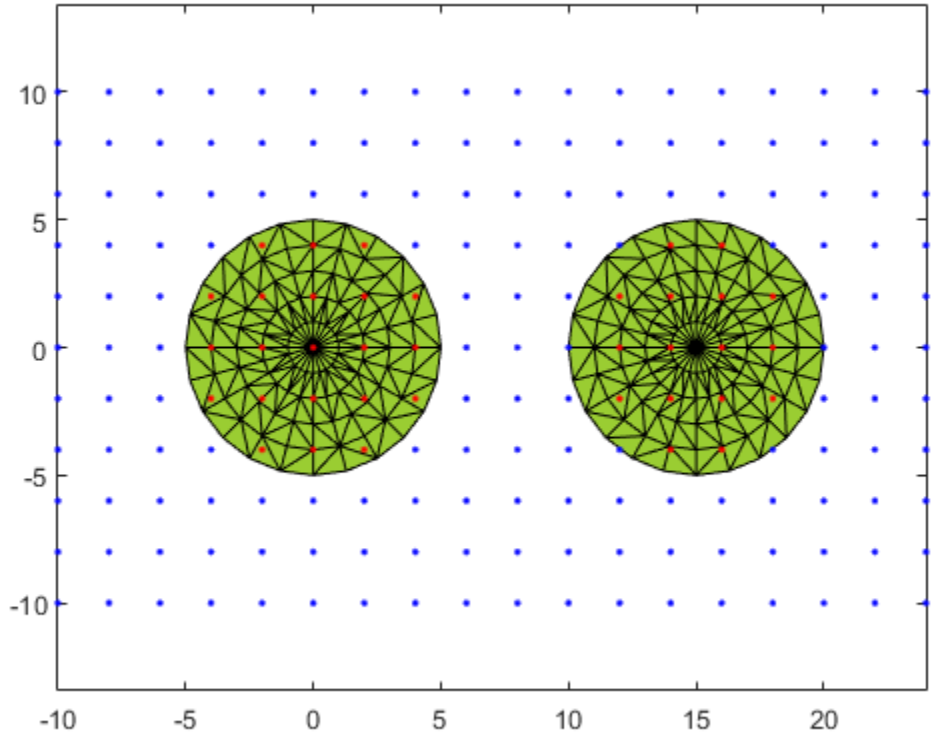


Create a Cartesian grid of query points near the alpha shape.

```
[qx, qy] = meshgrid(-10:2:25, -10:2:10);
```

Check if the query points are inside of the alpha shape, and if so, plot them red. Plot the query points that lie outside of the alpha shape in blue.

```
in = inShape(shp,qx,qy);
plot(shp)
hold on
plot(qx(in),qy(in),'r.')
plot(qx(~in),qy(~in),'b.')
```



Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

`qx` — Query point x-coordinates

numeric array

Query point x-coordinates, specified as a numeric array.

Data Types: `double`

`qy` — Query point y-coordinates

numeric array

Query point y-coordinates, specified as a numeric array.

Data Types: `double`

`qz` — Query point z-coordinates

numeric array

Query point z-coordinates, specified as a numeric array.

Data Types: `double`

`QP` — Query point coordinates

matrix with two columns | matrix with three columns

Query point coordinates, specified as a matrix with two columns (2-D) or a matrix with three columns (3-D).

- For 2-D, the columns of `QP` represent `x` and `y` coordinates, respectively.
- For 3-D, the columns of `QP` represent `x`, `y`, and `z` coordinates, respectively.

Data Types: `double`

`RegionID` — ID number for a region in the alpha shape

positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

Output Arguments

`tf` — Containment status of query points

logical array

Status of the query points, returned as a logical array. The size of `tf` is equal to the size of the inputs that specify the query points (`qx`, `qy`, `qz`, or `QP`).

`inShape` returns logical `1` (`true`) values for points that are within the alpha shape or exactly on the boundary.

`ID` — IDs of the regions containing the query points

numeric array

IDs of regions containing query points, returned as a numeric array. `ID` is the same size as `tf`.

More About

- Using alphaShape Objects

See Also

`alphaShape` | `plot`

Introduced in R2014b

nearestNeighbor

Determine nearest alpha shape boundary point

Syntax

```
I = nearestNeighbor(shp,qx,qy)
I = nearestNeighbor(shp,qx,qy,qz)
I = nearestNeighbor(shp,QP)

I = nearestNeighbor( ___ ,RegionID)

[I,D] = nearestNeighbor( ___ )
```

Description

`I = nearestNeighbor(shp,qx,qy)`, for a 2-D alpha shape `shp`, returns the indices of points on the boundary of `shp` closest to the query points. `I` is the array of nearest neighbor indices where each index corresponds to the row index in `shp.Points`. The `qx` and `qy` query coordinates must be the same size.

`I = nearestNeighbor(shp,qx,qy,qz)`, for a 3-D alpha shape, returns the indices of the boundary points of `shp` closest to `(qx,qy,qz)` and corresponds to the row indices in `shp.Points`. The `qx`, `qy`, and `qz` query coordinates must be the same size.

`I = nearestNeighbor(shp,QP)` specifies the query points as a matrix `QP`. For a 2-D alpha shape, `QP` is a matrix with two columns representing the `qx` and `qy` coordinates. For a 3-D alpha shape, `QP` has three columns representing the `qx`, `qy`, and `qz` coordinates.

`I = nearestNeighbor( ___ ,RegionID)` returns the index of the nearest point that lies on the boundary of the region specified by `RegionID`, where $1 \le$ `RegionID` $\le$ `numRegions(shp)`. You can include any of the input arguments in the previous syntaxes.

`[I,D] = nearestNeighbor( ___ )` additionally returns the Euclidean distance between the query point and its nearest neighbor. `D` has the same size as I.

Examples

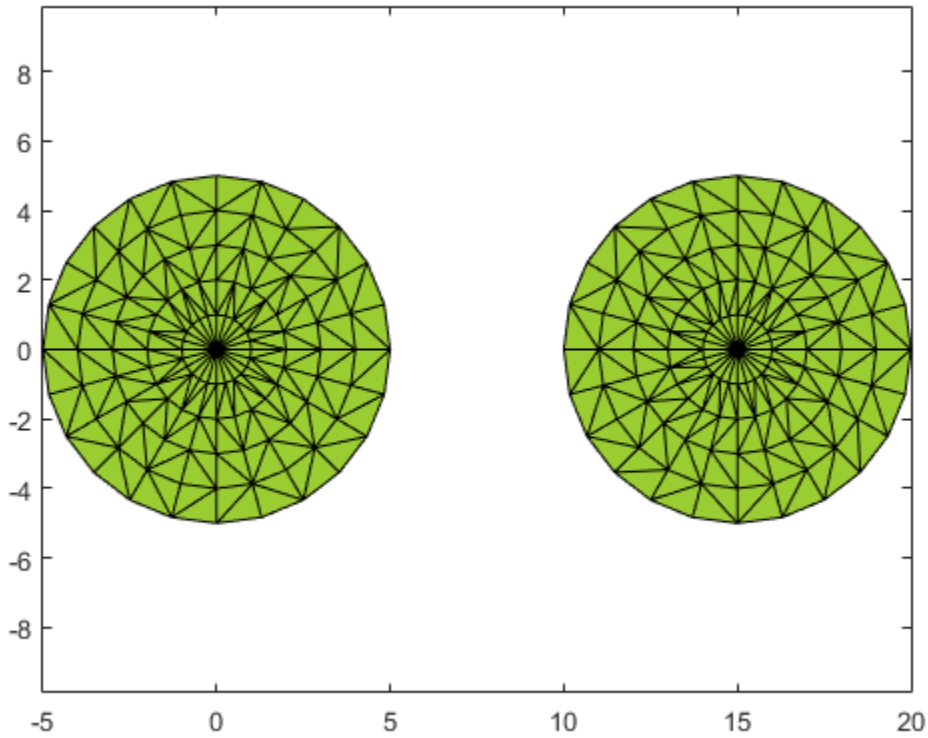
Nearest Alpha Shape Boundary Point

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15];
y = [y1; y1];
```

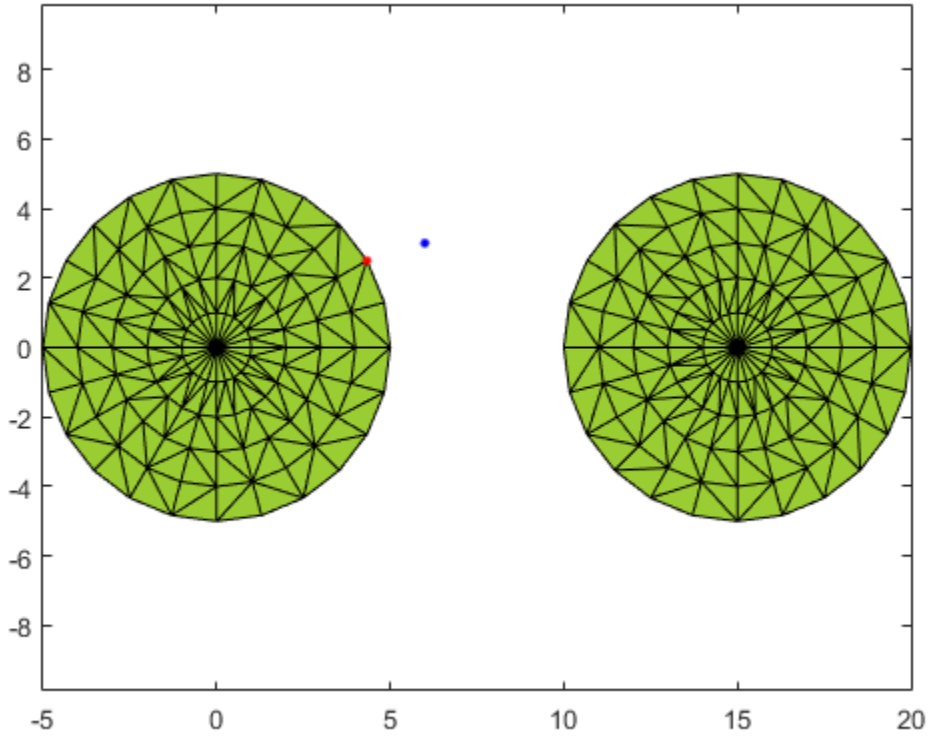
Create and plot an alpha shape with alpha radius equal to 1.

```
shp = alphaShape(x,y,1);
plot(shp)
hold on
```



Compute the nearest `shp` boundary point to the query point `QP`. Plot the query point in blue and the nearest boundary neighbor in red.

```
QP = [6 3];
plot(QP(1),QP(2),'b.','MarkerSize',10)
hold on
I = nearestNeighbor(shp, QP);
plot(shp.Points(I,1),shp.Points(I,2),'r.','MarkerSize',10)
```



Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

qx — Query point x-coordinates
numeric array

Query point x-coordinates, specified as a numeric array.

Data Types: `double`

qy — Query point y-coordinates
numeric array

Query point y-coordinates, specified as a numeric array.

Data Types: `double`

qz — Query point z-coordinates
numeric array

Query point z-coordinates, specified as a numeric array.

Data Types: `double`

QP — Query point coordinates
two-column matrix | three-column matrix

Query point coordinates, specified as a two-column matrix or a three-column matrix.

- For 2-D, the columns of `P` represent `qx` and `qy` coordinates, respectively.
- For 3-D, the columns of `P` represent `qx`, `qy`, and `qz` coordinates, respectively.

Data Types: `double`

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

Output Arguments

`I` — Nearest neighbor indices

integer-valued array

Nearest neighbor indices, returned as an integer-valued array. The indices correspond to the row index of `shp.Points` and indicate the points on the boundary of `shp` that are closest to the given query points.

`D` — Distance from query points to nearest neighbors

numeric array

Distance from query points to nearest neighbors, returned as a numeric array. `D` is the 2-D or 3-D Euclidean distance and is the same size as `I`.

More About

- Using alphaShape Objects

See Also

`alphaShape`

Introduced in R2015a

numRegions

Number of regions in alpha shape

Syntax

```
N = numRegions(shp)
```

Description

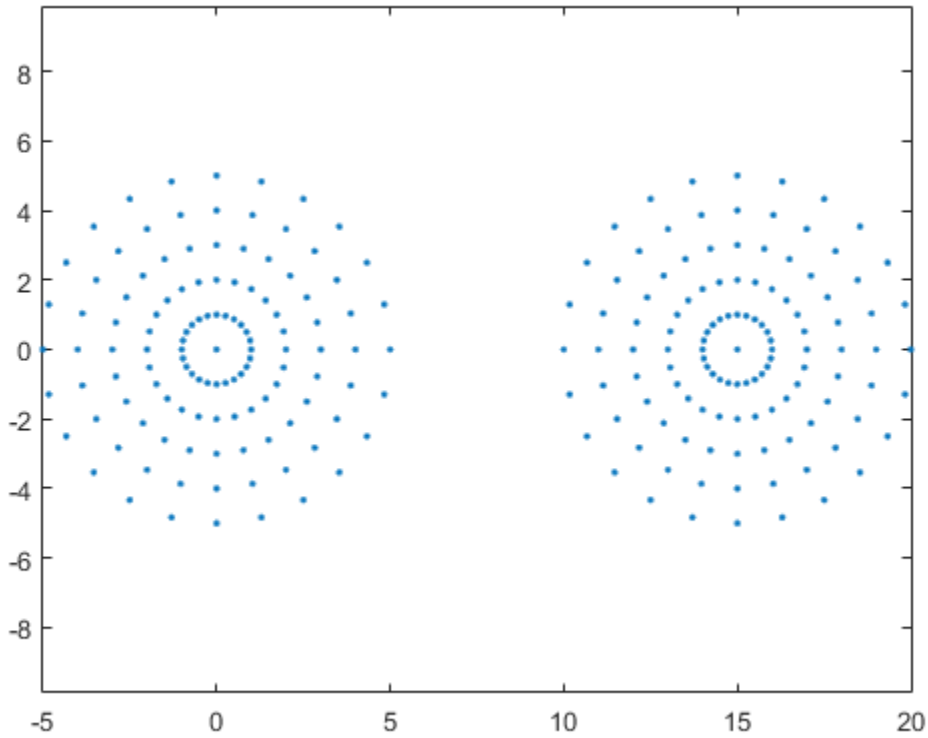
`N = numRegions(shp)` returns the number of distinct regions that make up the alpha shape. For an alpha radius of `Inf`, the alpha shape is the convex hull, and the number of regions is one. As the value of the alpha radius decreases the shape can break into separate regions, depending on the point set.

Examples

Find Number of Regions in 2-D Alpha Shape

Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



Create an alpha shape using an alpha radius of 7 and query the number of distinct regions in the shape.

```
shp = alphaShape(x,y,7);
nregions = numRegions(shp)
```

```
nregions =

     1
```

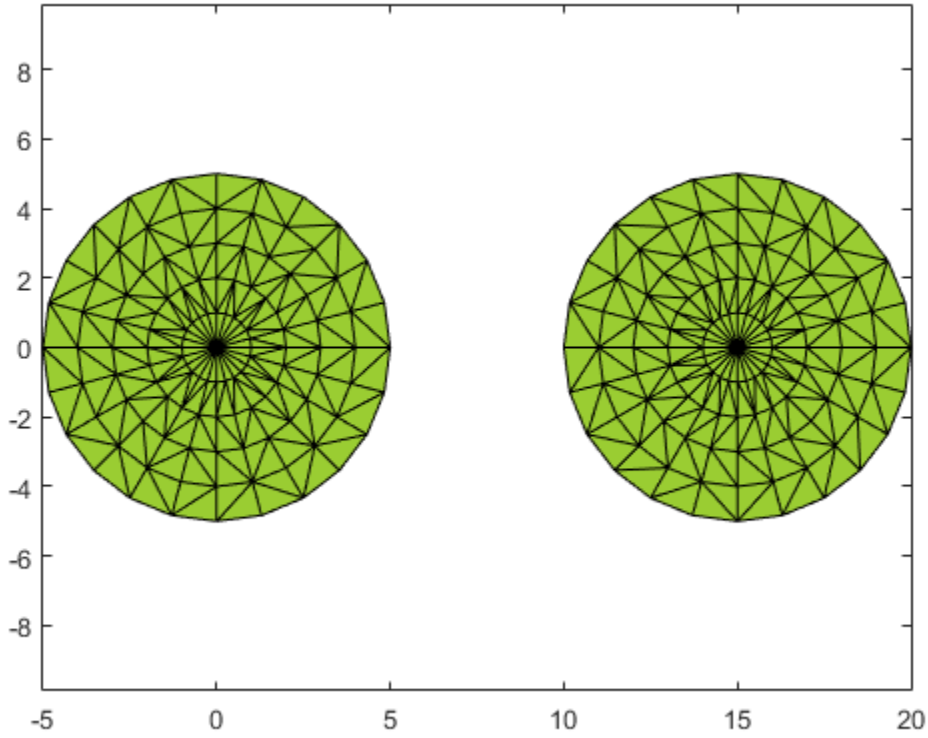
Use a smaller alpha radius of 2.5 to better capture the boundary and then retrieve the new number of distinct regions.

```
shp.Alpha = 2.5;
nregions = numRegions(shp)
```

```
nregions =

     2
```

Plot the alpha shape to check the boundary quality.

```
plot(shp)
```



Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

More About

- Using alphaShape Objects

See Also

`alphaShape`

Introduced in R2014b

perimeter

Perimeter of 2-D alpha shape

Syntax

```
L = perimeter(shp)
L = perimeter(shp,RegionID)
```

Description

`L = perimeter(shp)` returns the total perimeter of 2-D alpha shape `shp`, including the perimeter of any interior holes in the alpha shape.

`L = perimeter(shp,RegionID)` returns the perimeter of a region within the alpha shape. `RegionID` is the ID for the region and $1 \leq$ `RegionID` $\leq$ `numRegions(shp)`.

Examples

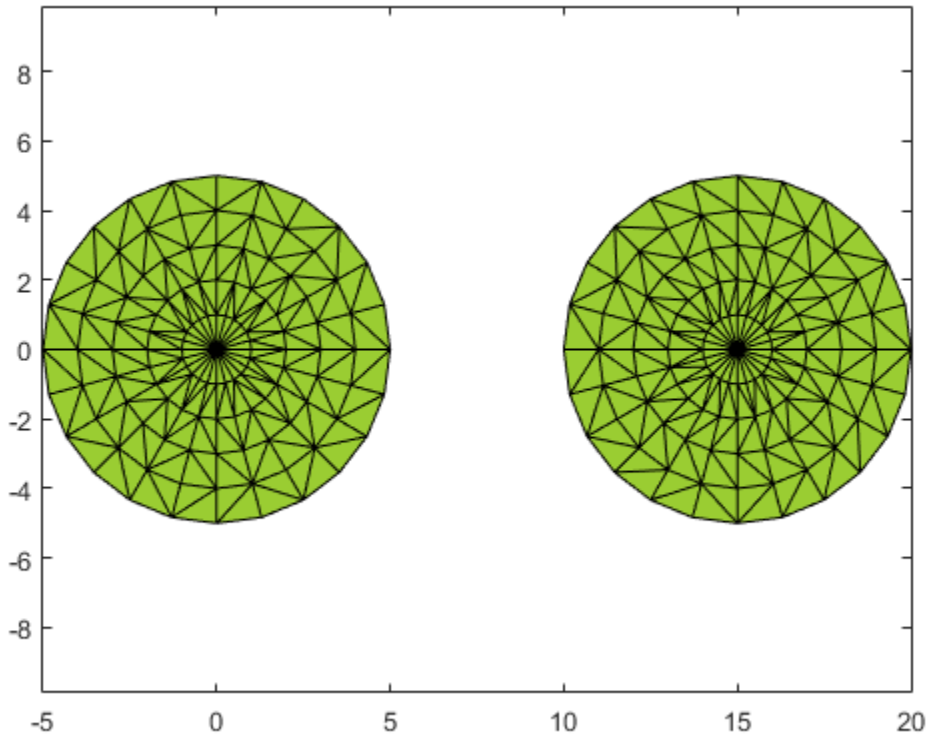
Find Perimeter of 2-D Alpha Shape

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
```

Create and plot an alpha shape using an alpha radius of 2.5.

```
shp = alphaShape(x,y,2.5);
plot(shp)
```



Compute the perimeter of the alpha shape.

```
totalperim = perimeter(shp)
```

```
totalperim =

   62.6526
```

Compute the perimeters of each of the two regions separately.

```
regionperims = perimeter(shp, 1:numRegions(shp))
```

```
regionperims =

   31.3263   31.3263
```

Input Arguments

shp — 2-D alpha shape

`alphaShape` object

2-D alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

RegionID — ID number for a region in the alpha shape

positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

More About

- Using alphaShape Objects

See Also

`alphaShape` | `area` | `criticalAlpha`

Introduced in R2014b

plot

Plot alpha shape

Syntax

```
plot(shp)
plot(shp,Name,Value)
h = plot( ___ )
```

Description

`plot(shp)` plots alpha shape `shp` in a figure window.

`plot(shp,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For a complete list of allowed `Name,Value` pairs, see Patch Properties.

`h = plot( ___ )` returns a handle to a `Patch` object using any of the previous syntaxes.

Examples

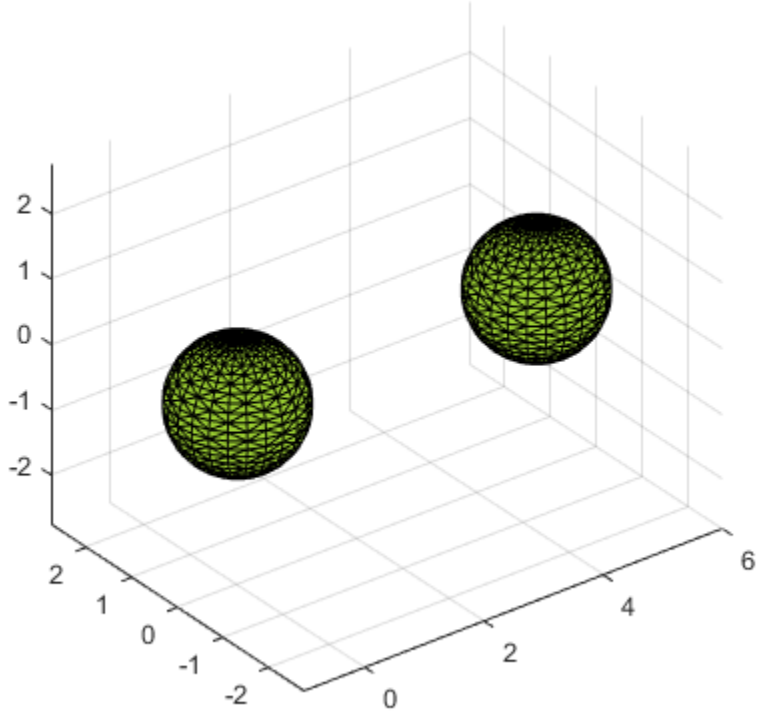
Plot 3-D Alpha Shape and Modify Patch Properties

Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

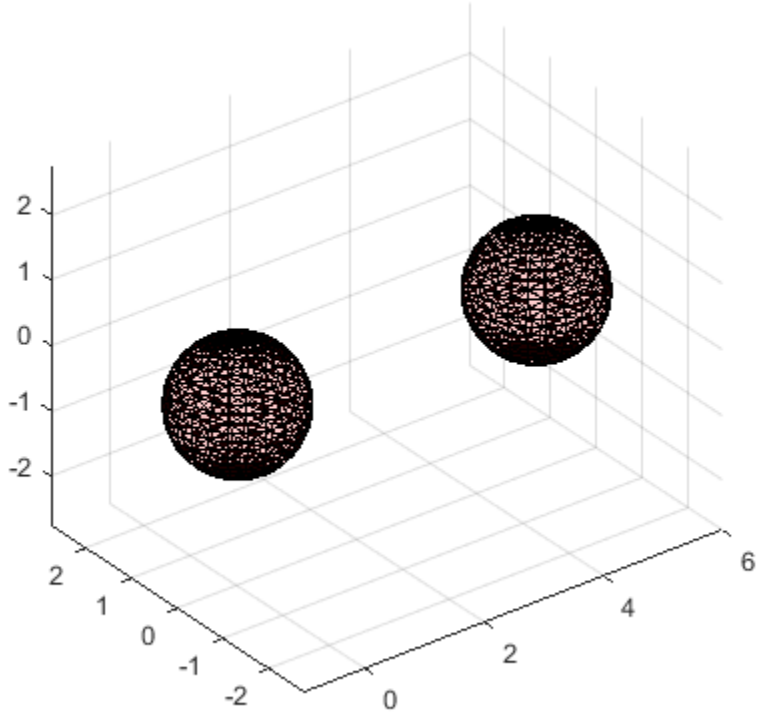
Create and plot an alpha shape using an alpha radius of 1.5.

```
shp = alphaShape(P,1.5);
plot(shp)
```



Plot the alpha shape with a specified color and transparency factor.

```
plot(shp,'FaceColor','red','FaceAlpha',0.1)
```



Input Arguments

shp — Alpha shape

`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

Output Arguments

h — Handle to `Patch` object

handle

Handle to `Patch` object, returned as a handle. For more information, see "Introduction to Patch Objects".

More About

- Using alphaShape Objects

See Also

`alphaShape` | `patch`

Introduced in R2014b

surfaceArea

Surface area of 3-D alpha shape

Syntax

```
A = surfaceArea(shp)
A = surfaceArea(shp,RegionID)
```

Description

`A = surfaceArea(shp)` returns the total surface area of 3-D alpha shape `shp`, including the surface area of any interior voids in the alpha shape.

`A = surfaceArea(shp,RegionID)` returns the surface area of a region of the alpha shape. `RegionID` is the ID for the region and $1 \leq$ `RegionID` $\leq$ `numRegions(shp)`.

Examples

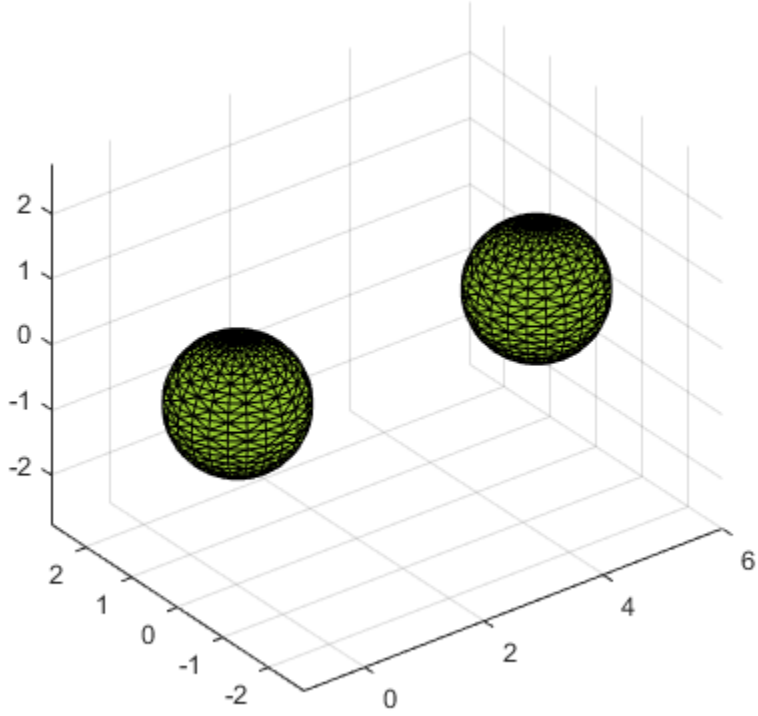
Find Surface Area of 3-D Alpha Shape

Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

Create and plot an alpha shape using an alpha radius of 1.5.

```
shp = alphaShape(P,1.5);
plot(shp)
```



Compute the surface area of the alpha shape.

```
totalsurfarea = surfaceArea(shp)
```

```
totalsurfarea =

   24.9361
```

Compute the surface area of each region separately.

```
regionsurfareas = surfaceArea(shp, 1:numRegions(shp))
```

```
regionsurfareas =

   12.4680   12.4680
```

Input Arguments

shp — 3-D alpha shape

`alphaShape` object

3-D alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`

Example: `shp = alphaShape(x,y,z)` creates a 3-D `alphaShape` object from the `(x,y,z)` point coordinates.

RegionID — ID number for a region in the alpha shape

positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

More About

- Using alphaShape Objects

See Also

`alphaShape` | `area` | `volume`

Introduced in R2014b

volume

Volume of 3-D alpha shape

Syntax

```
V = volume(shp)
V = volume(shp,RegionID)
```

Description

`V = volume(shp)` returns the volume of 3-D alpha shape `shp`.

`V = volume(shp,RegionID)` returns the volume of a region of the alpha shape. `RegionID` is the ID for the region and $1 \le$ `RegionID` $\le$ `numRegions(shp)`.

Examples

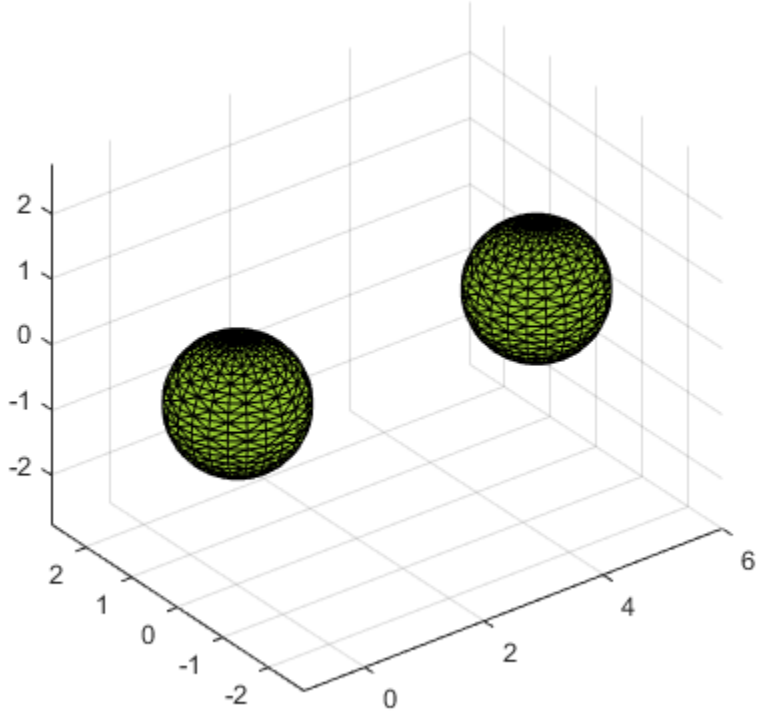
Find Volume of 3-D Alpha Shape

Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

Create and plot an alpha shape using an alpha radius of 1.5.

```
shp = alphaShape(P,1.5);
plot(shp)
```



Compute the volume of the alpha shape.

```
totalvol = volume(shp)
```

```
totalvol =

    8.2468
```

Compute the volumes of each of the two regions separately.

```
regionvols = volume(shp, 1:numRegions(shp))
```

```
regionvols =

    4.1234    4.1234
```

Input Arguments

shp — 3-D alpha shape

`alphaShape` object

3-D alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`

Example: `shp = alphaShape(x,y,z)` creates a 3-D `alphaShape` object from the `(x,y,z)` point coordinates.

RegionID — ID number for a region in the alpha shape

positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

More About

- Using alphaShape Objects

See Also

`alphaShape` | `area` | `surfaceArea`

Introduced in R2014b

amd

Approximate minimum degree permutation

Syntax

```
P = amd(A)
P = amd(A,opts)
```

Description

`P = amd(A)` returns the approximate minimum degree permutation vector for the sparse matrix `C = A + A'`. The Cholesky factorization of `C(P,P)` or `A(P,P)` tends to be sparser than that of `C` or `A`. The `amd` function tends to be faster than `symamd`, and also tends to return better orderings than `symamd`. Matrix `A` must be square. If `A` is a full matrix, then `amd(A)` is equivalent to `amd(sparse(A))`.

`P = amd(A,opts)` allows additional options for the reordering. The `opts` input is a structure with the two fields shown below. You only need to set the fields of interest:

- **dense** — A nonnegative scalar value that indicates what is considered to be dense. If A is n-by-n, then rows and columns with more than `max(16,(dense*sqrt(n)))` entries in `A + A'` are considered to be "dense" and are ignored during the ordering. MATLAB software places these rows and columns last in the output permutation. The default value for this field is 10.0 if this option is not present.
- **aggressive** — A scalar value controlling aggressive absorption. If this field is set to a nonzero value, then aggressive absorption is performed. This is the default if this option is not present.

MATLAB software performs an assembly tree post-ordering, which is typically the same as an elimination tree post-ordering. It is not always identical because of the approximate degree update used, and because “dense” rows and columns do not take part in the post-order. It well-suited for a subsequent `chol` operation, however, If you require a precise elimination tree post-ordering, you can use the following code:

```
P = amd(S);
C = spones(S)+spones(S');
[ignore, Q] = etree(C(P,P));
```

```
P = P(Q);
```

If `S` is already symmetric, omit the second line, `C = spones(S)+spones(S')`.

Examples

This example constructs a sparse matrix and computes a two Cholesky factors: one of the original matrix and one of the original matrix preordered by `amd`. Note how much sparser the Cholesky factor of the preordered matrix is compared to the factor of the matrix in its natural ordering:

```
A = gallery('wathen',50,50);
p = amd(A);
L = chol(A,'lower');
Lp = chol(A(p,p),'lower');

figure;
subplot(2,2,1);    spy(A);
title('Sparsity structure of A');

subplot(2,2,2); spy(A(p,p));
title('Sparsity structure of AMD ordered A');

subplot(2,2,3); spy(L);
title('Sparsity structure of Cholesky factor of A');

subplot(2,2,4); spy(Lp);
title('Sparsity structure of Cholesky factor of AMD ordered A');

set(gcf,'Position',[100 100 800 700]);
```

See Also

`colamd` | `colperm` | `symamd` | `symrcm`

ancestor

Ancestor of graphics object

Syntax

```
p = ancestor(h,type)
p = ancestor(h,type,'toplevel')
```

Description

`p = ancestor(h,type)` returns the handle of the closest ancestor of `h`, if the ancestor is one of the types of graphics objects specified by `type`. `type` can be:

- a string that is the name of a single type of object. For example, `'figure'`
- a cell array containing the names of multiple objects. For example, `{'hgtransform','hggroup','axes'}`

If MATLAB cannot find an ancestor of `h` that is one of the specified types, then `ancestor` returns `p` as empty. When `ancestor` searches the hierarchy, it includes the object itself in the search. Therefore, if the object with handle `h` is of one of the types listed in `type`, `ancestor` will return object `h`.

`ancestor` returns `p` as empty but does not issue an error if `h` is not the handle of a Handle Graphics object.

`p = ancestor(h,type,'toplevel')` returns the highest-level ancestor of `h`, if this type appears in the `type` argument.

Examples

Find the ancestors of a line object:

```
% Create some line objects and parent them
% to an hggroup object.
hgg = hggroup;
hgl = line(randn(5),randn(5),'Parent',hgg);
```

```
% Now get the ancestor of the lines:
p = ancestor(hgg,{'figure','axes','hggroup'});
get(p,'Type')
% Now get the top-level ancestor:
ptop=ancestor(hgg,{'figure','axes','hggroup'},'toplevel');
get(ptop,'type')
```

See Also

`findobj`

Introduced before R2006a

and, &

Find logical AND

Syntax

```
A & B & ...
and(A, B)
```

Description

`A & B & ...` performs a logical AND of all input arrays `A`, `B`, etc., and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if all input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input `A` is a 3-by-5 matrix and input `B` is the number 1, then `B` is treated as if it were a 3-by-5 matrix of ones.

`and(A, B)` is called for the syntax `A & B` when either `A` or `B` is an object.

Note The symbols & and && perform different operations in the MATLAB software. The element-wise AND operator described here is &. The short-circuit AND operator is &&.

Examples

If matrix `A` is

```
    0.4235    0.5798         0    0.7942         0
    0.5155         0    0.7833    0.0592    0.8744
```

```
    0.3340         0         0         0    0.0150
    0.4329    0.6405    0.6808    0.0503         0
```

and matrix B is

```
     0     1     0     1     0
     1     1     1     0     1
     0     1     1     1     0
     0     1     0     0     1
```

then

```
A & B
ans =
     0     1     0     1     0
     1     0     1     0     1
     0     0     0     0     0
     0     1     0     0     0
```

More About

- "Truth Table for Logical Operations"

See Also

`all` | `any` | `bitand` | `Logical Operators: Short Circuit` | `not` | `or` | `xor`

Introduced before R2006a

angle

Phase angle

Syntax

```
P = angle(Z)
```

Description

`P = angle(Z)` returns the phase angles, in radians, for each element of complex array `Z`. The angles lie between $\pm\pi$.

For complex `Z`, the magnitude `R` and phase angle `theta` are given by

```
R = abs(Z)
theta = angle(Z)
```

and the statement

```
Z = R.*exp(i*theta)
```

converts back to the original complex `Z`.

Examples

```
Z = [ 1 - 1i   2 + 1i   3 - 1i   4 + 1i
      1 + 2i   2 - 2i   3 + 2i   4 - 2i
      1 - 3i   2 + 3i   3 - 3i   4 + 3i
      1 + 4i   2 - 4i   3 + 4i   4 - 4i ]

P = angle(Z)

P =
   -0.7854    0.4636   -0.3218    0.2450
    1.1071   -0.7854    0.5880   -0.4636
   -1.2490    0.9828   -0.7854    0.6435
    1.3258   -1.1071    0.9273   -0.7854
```

More About

Algorithms

The `angle` function can be expressed as `angle(z) = imag(log(z)) = atan2(imag(z),real(z))`.

See Also

`abs` | `atan2` | `unwrap`

Introduced before R2006a

Using Animated Line Objects

Line animations

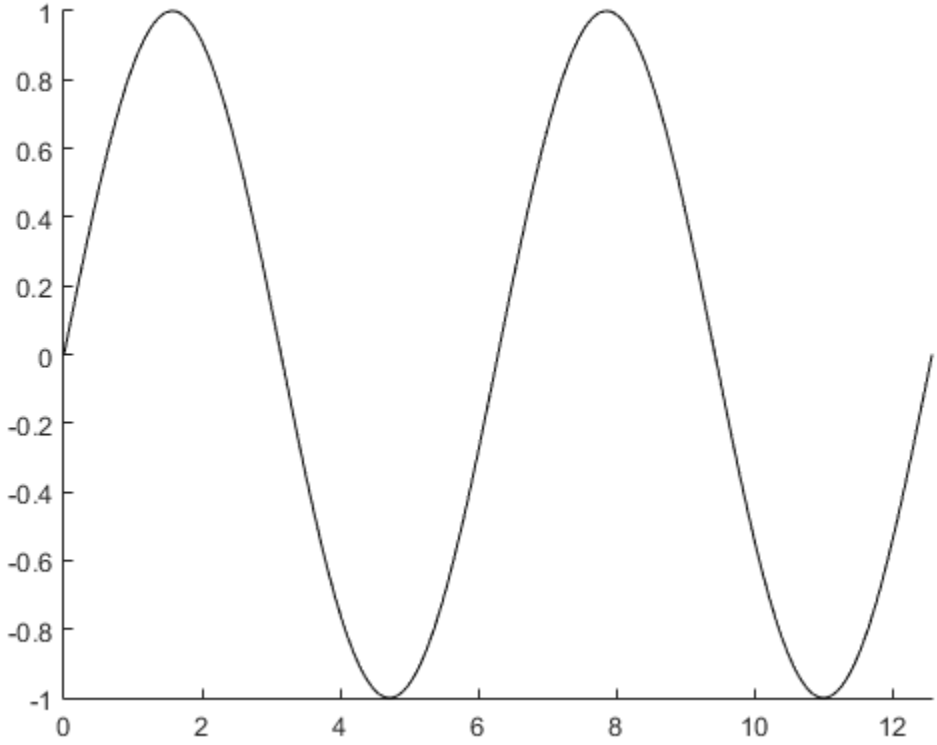
Animated line objects optimize line animations by accumulating data from a streaming data source. After you create the initial animated line using the `animatedline` function, you can add new points to the line without having to redefine the existing points. Modify the appearance of the animated line by setting its properties.

Examples

Display Line Animation

Create the initial animated line object. Then, use a loop to add 1,000 points to the line. After adding each new point, use `drawnow` to display the new point on the screen.

```
h = animatedline;
axis([0,4*pi,-1,1])

x = linspace(0,4*pi,1000);
y = sin(x);
for k = 1:length(x)
    addpoints(h,x(k),y(k));
    drawnow
end
```



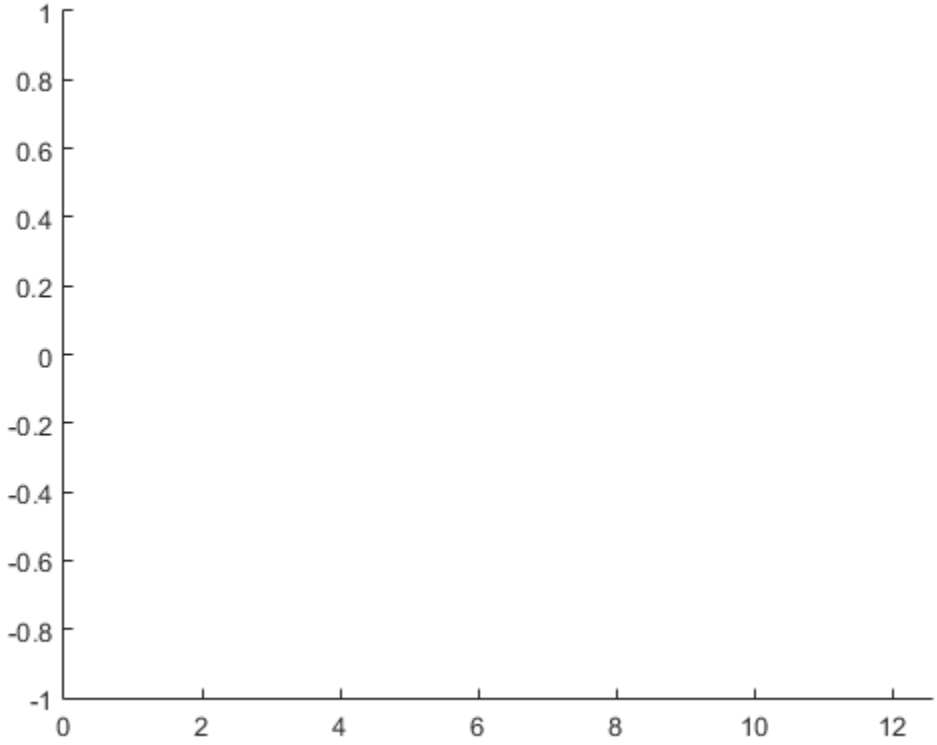
For faster rendering, add more than one point to the line each time through the loop or use `drawnow limitrate`.

Query the points of the line.

```
[xdata,ydata] = getpoints(h);
```

Clear the points from the line.

```
clearpoints(h)
drawnow
```

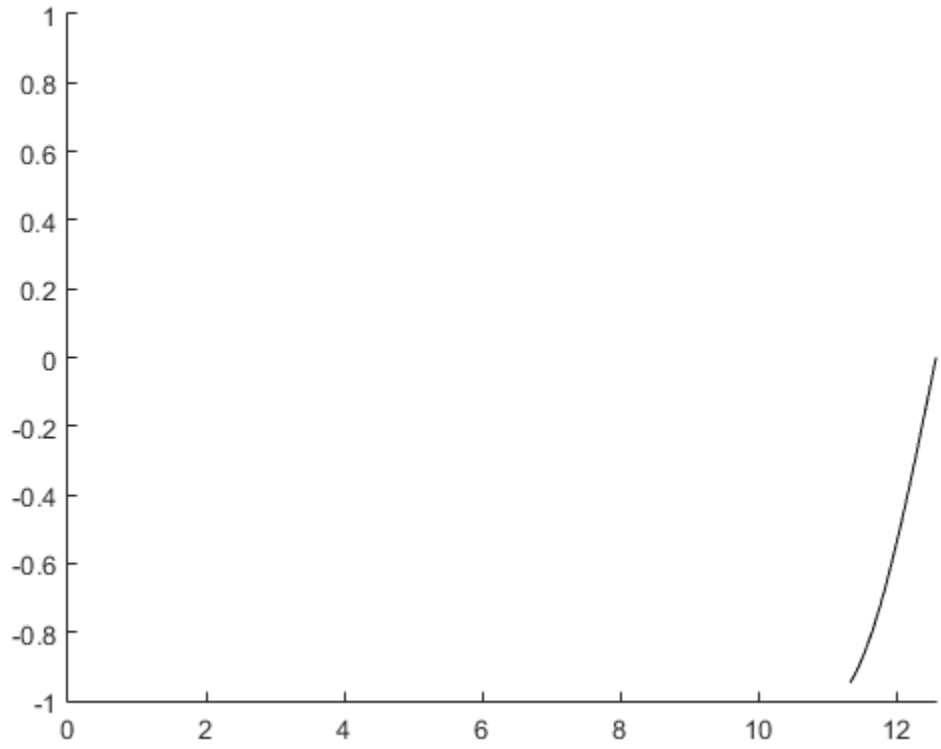


Set Maximum Number of Points

Limit the number of points in the animated line to 100. Use a loop to add one point to the line at a time. When the line contains 100 points, adding a new point to the line deletes the oldest point.

```
h = animatedline('MaximumNumPoints',100);
axis([0,4*pi,-1,1])

x = linspace(0,4*pi,1000);
y = sin(x);
for k = 1:length(x)
    addpoints(h,x(k),y(k));
    drawnow
```

```
end
```

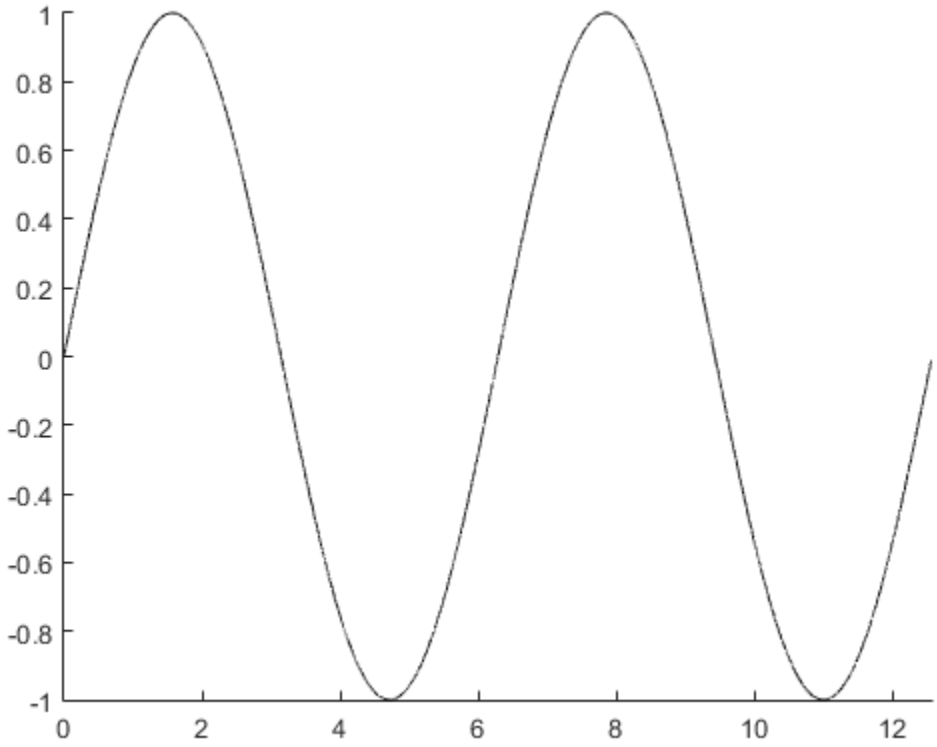


Add Points in Sets for Fast Animation

Use a loop to add 100,000 points to an animated line. Since the number of points is large, adding one point to the line each time through the loop might be slow. Instead, add 100 points to the line each time through the loop for a faster animation.

```
h = animatedline;
axis([0,4*pi,-1,1])

numpoints = 100000;
x = linspace(0,4*pi,numpoints);
y = sin(x);
for k = 1:100:numpoints-100
```

```
    xvec = x(k:k+99);
    yvec = y(k:k+99);
    addpoints(h,xvec,yvec)
    drawnow
end
```

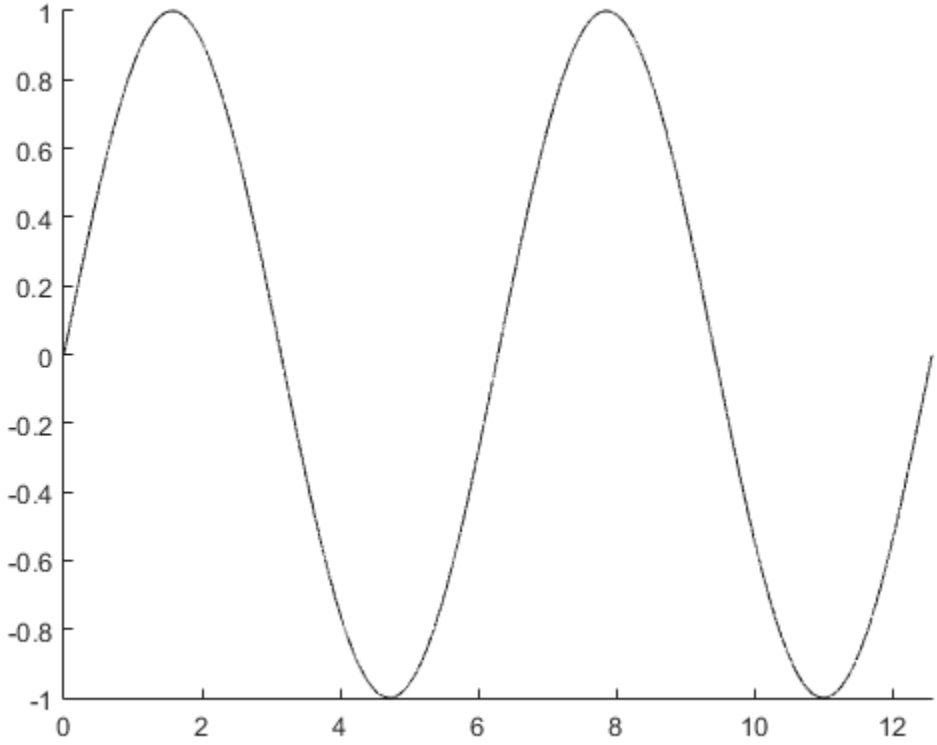


Another technique for creating faster animations is to use `drawnow limitrate` instead of `drawnow`.

Use drawnow limitrate for Fast Animation

Use a loop to add 100,000 points to an animated line. Since the number of points is large, using `drawnow` to display the changes might be slow. Instead, use `drawnow limitrate` for a faster animation.

```
h = animatedline;
axis([0,4*pi,-1,1])

numpoints = 100000;
x = linspace(0,4*pi,numpoints);
y = sin(x);
for k = 1:numpoints
    addpoints(h,x(k),y(k))
    drawnow limitrate
end
```



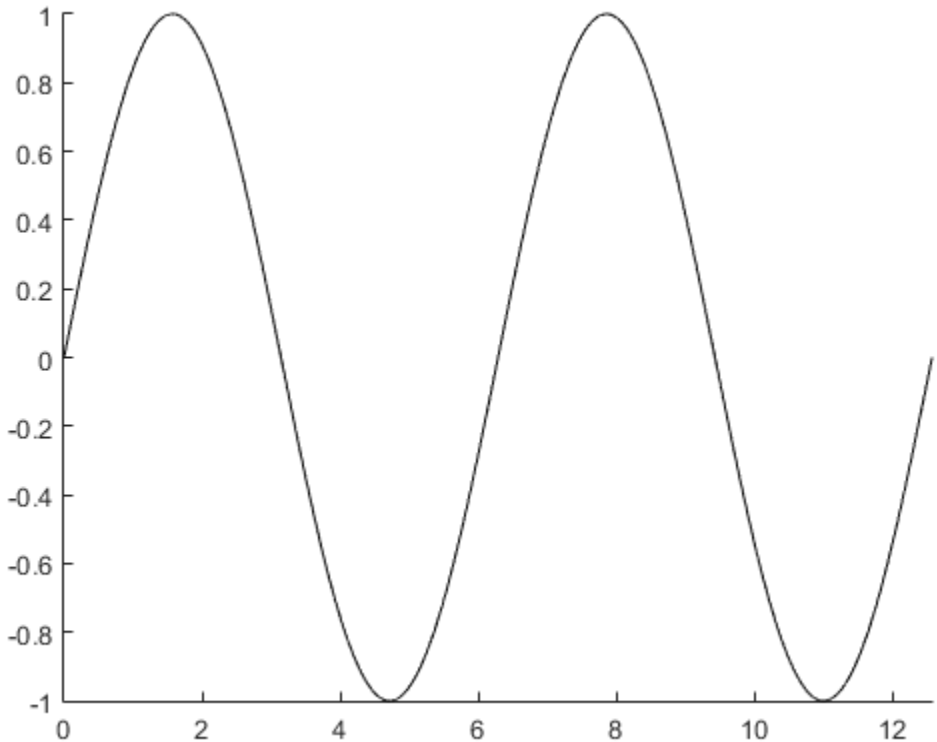
Control Animation Speed

Control the animation speed by running through several iterations of the animation loop before drawing the updates on the screen. Use this technique when `drawnow` is too slow and `drawnow limitrate` is too fast.

For example, update the screen every 1/30 seconds. Use the `tic` and `toc` commands to keep track of how much time passes between screen updates.

```
h = animatedline;
axis([0,4*pi,-1,1])
numpoints = 10000;
x = linspace(0,4*pi,numpoints);
y = sin(x);
```

```
a = tic; % start timer
for k = 1:numpoints
    addpoints(h,x(k),y(k))
    b = toc(a); % check timer
    if b > (1/30)
        drawnow % update screen every 1/30 seconds
        a = tic; % reset timer after updating
    end
end
drawnow % draw final frame
```



A smaller interval updates the screen more often and results in a slower animation. For example, use `b > (1/1000)` to slow down the animation.

Properties

Animated Line Properties

Object Functions

`addpoints` `getpoints` `clearpoints`

Create Object

Create an animated line object using the `animatedline` function.

See Also

`drawnow` | `movie`

animatedline

Create animated line

Syntax

```
h = animatedline
h = animatedline(x,y)
h = animatedline(x,y,z)

h = animatedline(___,Name,Value)
```

Description

`h = animatedline` creates an animated line that has no data and adds it to the current axes. Add points to the line in a loop to create a line animation. For more information, see Using Animated Line Objects.

`h = animatedline(x,y)` creates an animated line with initial data points defined by `x` and `y`.

`h = animatedline(x,y,z)` creates an animated line with initial data points defined by `x` , `y`, and `z`.

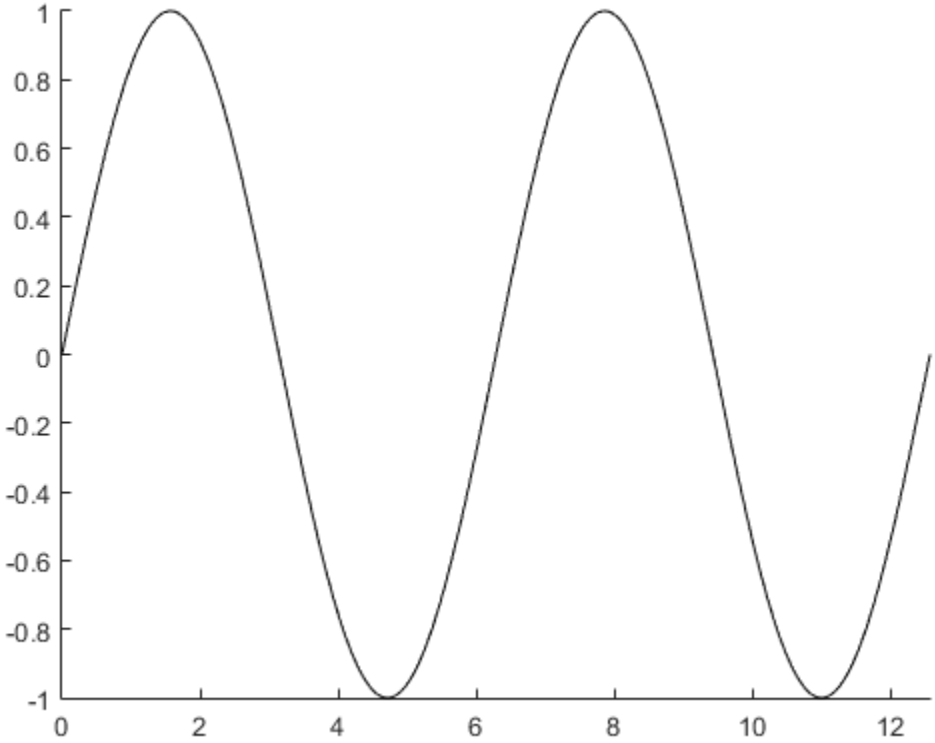
`h = animatedline(___,Name,Value)` specifies animated line properties using one or more `Name,Value` pair arguments. For example, `'Color','r'` sets the line color to red. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Display Line Animation

Create the initial animated line object. Then, use a loop to add 1,000 points to the line. After adding each new point, use `drawnow` to display the new point on the screen.

```
h = animatedline;
axis([0,4*pi,-1,1])

x = linspace(0,4*pi,1000);
y = sin(x);
for k = 1:length(x)
    addpoints(h,x(k),y(k));
    drawnow
end
```



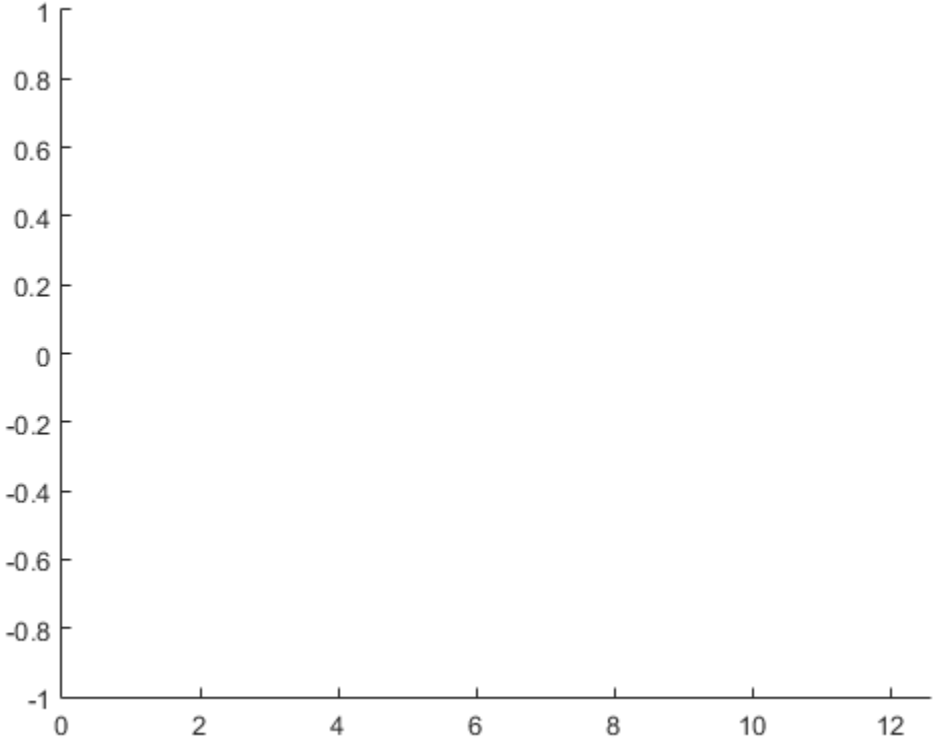
For faster rendering, add more than one point to the line each time through the loop or use `drawnow limitrate`.

Query the points of the line.

```
[xdata,ydata] = getpoints(h);
```

Clear the points from the line.

```
clearpoints(h)
drawnow
```

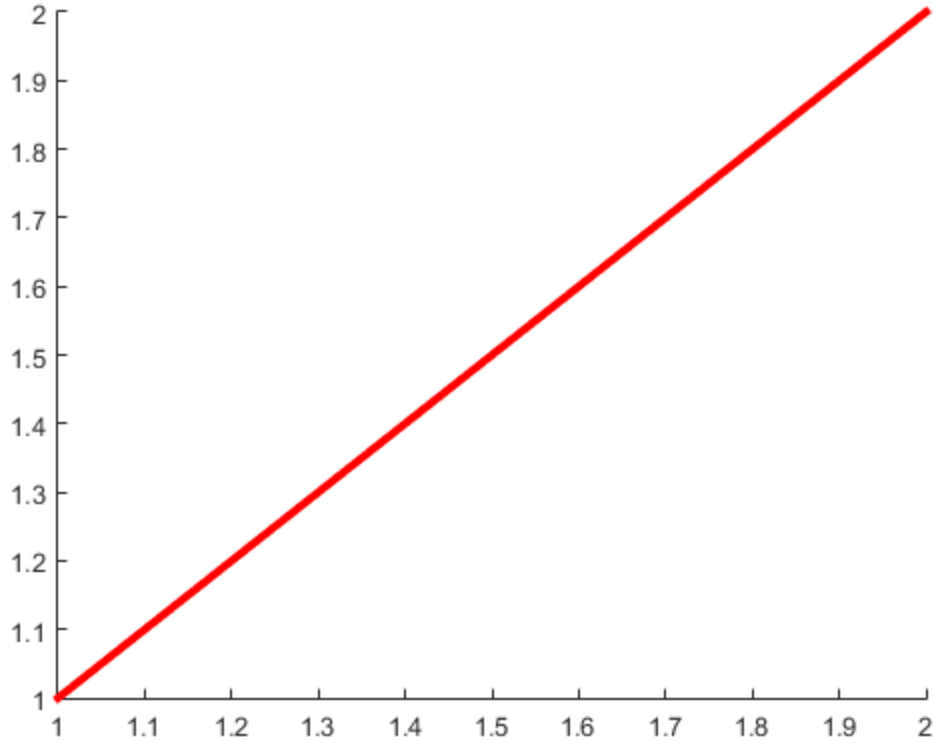


Specify Animated Line Color

Set the color of the animated line to red and set its line width to 3 points.

```
x = [1 2];
y = [1 2];
```

```
h = animatedline(x,y,'Color','r','LineWidth',3);
```



Input Arguments

x — Initial x values

[ ] (default) | scalar or vector

Initial *x* values for the animated line, specified as a scalar or vector. The input `x` must be equal in size to `y`.

Example: `1:10`

Data Types: `double`

y — Initial y values

[ ] (default) | scalar or vector

Initial *y* values for the animated line, specified as a scalar or vector. The input `y` must be equal in size to `x`.

Example: `1:10`

Data Types: `double`

z — Initial z values

[ ] (default) | scalar or vector

Initial *z* values for the animated line, specified as a scalar or vector. The input `z` must be equal in size to `x` and `y`.

Example: `1:10`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The animated line properties listed here are only a subset. For a complete list, see Animated Line Properties.

Example: `'Color','red','Marker','o'`

'Color' — Line color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the `Color` as `'none'`, then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

`[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`'LineStyle'` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- - - - -
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	_._._._.-
`'none'`	No line	No line

`'LineWidth'` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`'Marker'` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the marker symbol strings listed in this table. By default, the animated line object does not have markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`'MarkerSize'` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`'MarkerEdgeColor'` — Color of marker edge

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`'MarkerFaceColor'` — Color of marker interior

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`'MaximumNumPoints'` — Maximum number of points stored and displayed

`1000000` (default) | positive value | `Inf`

Maximum number of points stored and displayed as part of the line, specified as a positive value or `Inf`. By default, the value is one million points. If the number of points exceeds the maximum value permitted, then the animated line keeps the most recently added points and drops points from the beginning of the line. These dropped points no longer display on the screen and are not returned when using `getpoints`.

Use this property to limit the number of points appearing on the screen at any given time or to limit the amount of memory used. If you specify the value as `Inf`, then the animated line does not drop any points, but the number of points stored is limited by the amount of memory available.

Example: `10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`h` — Animated line object

animated line object

Animated line object. Use `h` to modify the animated line object after its been created, such as changing property values or adding points to the line. For more information, see Using Animated Line Objects.

More About

Tips

- Animated lines do not support data tips.

See Also

Functions
`addpoints` | `clearpoints` | `getpoints`

Using Objects
Using Animated Line Objects

Properties
Animated Line Properties

Introduced in R2014b

Animated Line Properties

Control animated line appearance and behavior

Animated line properties control the appearance and behavior of an animated line object. By changing property values, you can modify certain aspects of the line. Use dot notation to refer to a particular object and property:

```
h = animatedline;
c = h.Color;
h.Color = 'red';
```

Line

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the `Color` as `'none'`, then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

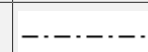
Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`MaximumNumPoints` — Maximum number of points stored and displayed

`1000000` (default) | positive value | `Inf`

Maximum number of points stored and displayed as part of the line, specified as a positive value or `Inf`. By default, the value is one million points. If the number of points exceeds the maximum value permitted, then the animated line keeps the most recently added points and drops points from the beginning of the line. These dropped points no longer display on the screen and are not returned when using `getpoints`.

Use this property to limit the number of points appearing on the screen at any given time or to limit the amount of memory used. If you specify the value as `Inf`, then the animated line does not drop any points, but the number of points stored is limited by the amount of memory available.

Example: 10

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

AlignVertexCenters — Sharp vertical and horizontal lines

'off' (default) | 'on'

Sharp vertical and horizontal lines, specified as 'off' or 'on'.

If the associated figure has a GraphicsSmoothing property set to 'on' and a Renderer property set to 'opengl', then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the AlignVertexCenters property to eliminate the uneven appearance.

- 'off' — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- 'on' — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type opengl info. If it is supported, then the returned fields contain the line SupportsAlignVertexCenters: 1.

Markers

Marker — Marker symbol

'none' (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the animated line object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
'o'	Circle
'+'	Plus sign
'*'	Asterisk

String	Marker Symbol
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

`6` (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]

Long Name	Short Name	RGB Triplet
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.3 0.2 0.1]

Example: 'green'

Visibility

Visible — Visibility of animated line

'on' (default) | 'off'

Visibility of animated line, specified as one of these values:

- 'on' — Display the animated line.
- 'off' — Hide the animated line without deleting it. You still can access the properties of an invisible animated line object.

Clipping — Clipping of animated line to axes limits

'on' (default) | 'off'

Clipping of animated line to the axes limits, specified as one of these values:

- 'on' — Do not display parts of the animated line that are outside the axes limits.
- 'off' — Display the entire animated line, even if parts of it appear outside the axes limits. Parts of the animated line might appear outside the axes limits if you create a plot, set hold on, freeze the axis scaling, and then create the animated line that is larger than the original plot.

Identifiers

Type — Type of graphics object

'animatedline'

Type of graphics object, returned as 'animatedline'. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using findobj.

`Tag` — User-specified tag

`''` (default) | string

Tag to associate with the animated line, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with animated line

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the animated line object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the animated line.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the animated line object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the animated line from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the animated line object in the legend as one entry (default).
   - `'off'` — Do not include the animated line object in the legend.
   - `'children'` — Include only children of the animated line object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of animated line

axes object | group object | transform object

Parent of animated line, specified as an axes, group, or transform object.

`Children` — Children of animated line

empty `GraphicsPlaceholder` array

The animated line has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of animated line object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The animated line object handle is always visible.
- `'off'` — The animated line object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The animated line object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the animated line at the command-line, but allows callback functions to access it.

If the animated line object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the animated line. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The animated line object — You can access properties of the animated line object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the animated line. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the animated line when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the animated line.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the animated line that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the animated line responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the animated line passes the click to the object below it in the current view of the figure window. The HitTest property of the animated line has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the animated line. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the animated line that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the animated line object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the animated line is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the

---

`BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the animated line tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the animated line. Setting the `CreateFcn` property on an existing animated line has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during animated line creation. MATLAB executes the callback after creating the animated line and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The animated line object — You can access properties of the animated line object from within the callback function. You also can access the animated line object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the animated line. MATLAB executes the callback before destroying the animated line so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The animated line object — You can access properties of the animated line object from within the callback function. You also can access the animated line object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of animated line

`'off'` (default) | `'on'`

Deletion status of animated line, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the animated line begins

execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the animated line no longer exists.

Check the value of the BeingDeleted property to verify that the animated line is not about to be deleted before querying or modifying it.

See Also

Functions

`animatedline`

Using Objects

Using Animated Line Objects

More About

- “Access Property Values”
- “Graphics Object Properties”

annotation

Create annotation objects

Syntax

```
annotation(annotation_type)
annotation('line',x,y)
annotation('arrow',x,y)
annotation('doublearrow',x,y)
annotation('textarrow',x,y)
annotation('textbox',[x y w h])
annotation('ellipse',[x y w h])
annotation('rectangle',[x y w h])
annotation(handle,...)
annotation(...,'PropertyName',PropertyValue,...)
anno_obj_handle = annotation(...)
```

Description

`annotation(annotation_type)` creates the specified annotation type using default values for all properties. *`annotation_type`* can be one of the following strings:

- `'line'`
- `'arrow'`
- `'doublearrow'` (two-headed arrow),
- `'textarrow'` (arrow with attached text box),
- `'textbox'`
- `'ellipse'`
- `'rectangle'`

`annotation('line',x,y)` creates a line annotation object that extends from the point defined by `x(1)`,`y(1)` to the point defined by `x(2)`,`y(2)`, specified in normalized figure units.

`annotation('arrow',x,y)` creates an arrow annotation object that extends from the point defined by `x(1)`,`y(1)` to the point defined by `x(2)`,`y(2)`, specified in normalized figure units.

`annotation('doublearrow',x,y)` creates a two-headed annotation object that extends from the point defined by `x(1)`,`y(1)` to the point defined by `x(2)`,`y(2)`, specified in normalized figure units.

`annotation('textarrow',x,y)` creates a textarrow annotation object that extends from the point defined by `x(1)`,`y(1)` to the point defined by `x(2)`,`y(2)`, specified in normalized figure units. The tail end of the arrow is attached to an editable text box.

`annotation('textbox',[x y w h])` creates an editable text box annotation with its lower left corner at the point `x`,`y`, a width `w`, and a height `h`, specified in normalized figure units. Specify `x`, `y`, `w`, and `h` in a single vector.

To type in the text box, enable plot edit mode (`plotedit`) and double-click within the box.

`annotation('ellipse',[x y w h])` creates an ellipse annotation with the lower left corner of the bounding rectangle at the point `x`,`y`, a width `w`, and a height `h`, specified in normalized figure units. Specify `x`, `y`, `w`, and `h` in a single vector.

`annotation('rectangle',[x y w h])` creates a rectangle annotation with the lower left corner of the rectangle at the point `x`,`y`, a width `w`, and a height `h`, specified in normalized figure units. Specify `x`, `y`, `w`, and `h` in a single vector.

`annotation(handle,...)` creates the annotation in the figure, uipanel, or uitab specified by `handle`.

`annotation(...,'PropertyName',PropertyValue,...)` creates the annotation and sets the specified properties to the specified values.

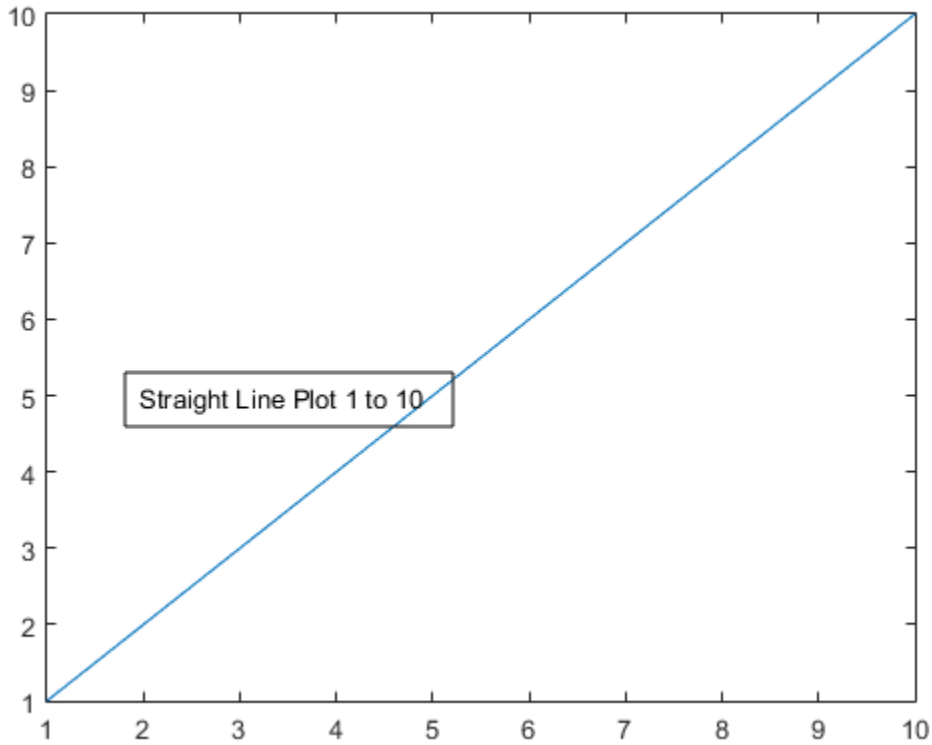
`anno_obj_handle = annotation(...)` returns the handle to the annotation objects.

Examples

Add Text Box to Graph

Annotate the graph with a text box. Specify the text box location in normalized figure coordinates.

```
figure; % new figure window
plot(1:10);
annotation('textbox', [0.2,0.4,0.1,0.1],...
           'String', 'Straight Line Plot 1 to 10');
```

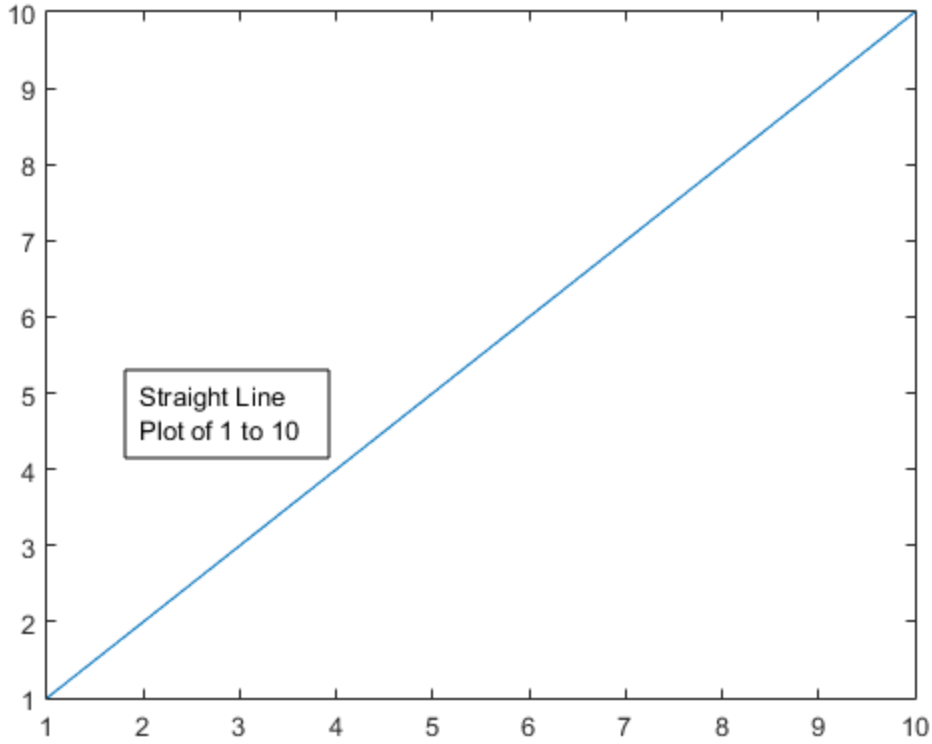


Add Text Box with Multiple Lines to Graph

Insert multiple lines in a text box by creating a cell array of strings. Each element is used as a separate line.

```
figure
plot(1:10)
str = {'Straight Line', 'Plot of 1 to 10'};
```

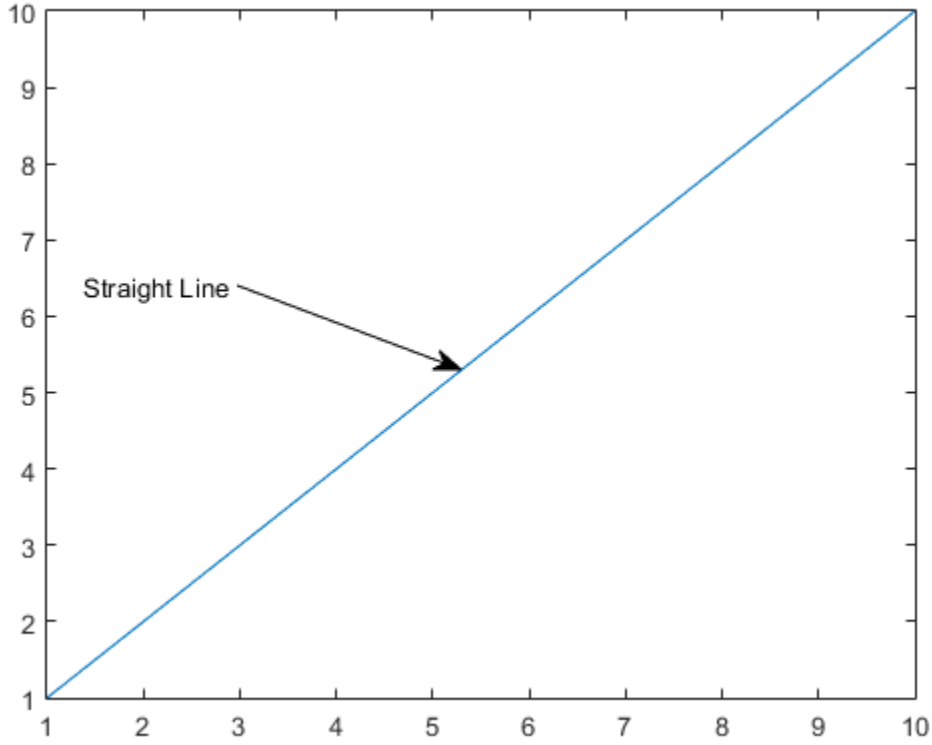
```
annotation('textbox', [0.2,0.4,0.1,0.1],...
           'String', str)
```



Add Text Arrow to Graph

Add a text arrow to the graph. Define the text arrow to start from `(0.3,0.6)` and extend to `(0.5,0.5)` in normalized figure coordinates.

```
figure
plot(1:10)
annotation('textarrow',[0.3,0.5],[0.6,0.5],...
           'String','Straight Line ')
```



Adding Annotations Interactively

It is often convenient to place annotations interactively. For details, see “Customize Graph Using Plot Tools”.

More About

- “Add Annotations to Graph Interactively”

See Also

Properties

Annotation Arrow Properties | Annotation Doublearrow Properties | Annotation Ellipse Properties | Annotation Line Properties | Annotation Rectangle Properties | Annotation Textarrow Properties | Annotation Textbox Properties

Introduced before R2006a

Annotation Arrow Properties

Control annotation arrow appearance and behavior

Annotation arrow properties control the appearance and behavior of an annotation arrow object. By changing property values, you can modify certain aspects of the arrow.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('arrow');
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Color` — Arrow color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Arrow color, specified as a three-element RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you set the color to `'none'`, then the arrow is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`

Long Name	Short Name	RGB Triplet
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

LineStyle — Style of arrow stem

'-' (default) | '--' | ':' | '-.' | 'none'

Style of arrow stem, specified as one of the strings listed in this table.

String	Line Style	Result
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No stem	No stem

LineWidth — Width of arrow stem

0.5 (default) | positive value

Width of arrow stem, specified as a positive value in point units. One point equals 1/72 inch.

Example: 0.75

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

HeadStyle — Style of arrowhead

'vback2' (default) | string

Style of the arrowhead, specified as one of the strings in this table.

String	Result	String	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	
`'vback2'` (default)		`'rose'`	
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`HeadLength` — Length of arrowhead

10 (default) | scalar numeric value

Length of the arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The arrowhead extends backwards from the (*x*, *y*) coordinate defined by the first two elements in the Position property.

Example: 15

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`HeadWidth` — Width of arrowhead

10 (default) | scalar numeric value

Width of the arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: 15

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Location and Size

X — Beginning and ending x-coordinates

`[0.3 0.4]` (default) | two-element vector

Beginning and ending *x*-coordinates, specified as a two-element vector of the form `[x_begin x_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.5]`

Y — Beginning and ending y-coordinates

`[0.3 0.4]` (default) | two-element vector

Beginning and ending *y*-coordinates, specified as a two-element vector of the form `[y_begin y_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.5]`

Position — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin dx dy]`. The first two elements specify the coordinates for the beginning of the arrow. The second two elements specify the slope of the arrow.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.5 0.5 0.2 0.3]`

Units — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the Position property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume Units is set to the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the annotation arrow, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

See Also

`annotation`

More About

- “Access Property Values”
- “Graphics Object Properties”

Annotation Double Arrow Properties

Control annotation double arrow appearance and behavior

Annotation double arrow properties control the appearance and behavior of an annotation double arrow object. By changing property values, you can modify certain aspects of the double arrow.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('doublearrow');
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Color` — Arrow color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Arrow color, specified as a three-element RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you set the color to `'none'`, then the arrow is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`

Long Name	Short Name	RGB Triplet
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

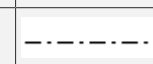
Example: 'blue'

Example: [0 0 1]

LineStyle — Style of arrow stem

'-' (default) | '--' | ':' | '-.' | 'none'

Style of arrow stem, specified as one of the strings listed in this table.

String	Line Style	Result
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No stem	No stem

LineWidth — Width of arrow stem

0.5 (default) | positive value

Width of arrow stem, specified as a positive value in point units. One point equals 1/72 inch.

Example: 0.75

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Head1Style — Style of first arrowhead

'vback2' (default) | string

Style of the first arrowhead, specified as one of the head style strings in this table. The first arrowhead is located at the point `(x_begin,y_begin)` determined by the X and Y properties.

String	Result	String	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	
`'vback2'` (default)		`'rose'`	
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`Head2Style` — Style of second arrowhead

`'vback2'` (default) | string

Style of the second arrowhead, specified as one of the head style strings in this table. The second arrowhead is located at the point `(x_end,y_end)` determined by the X and Y properties.

String	Result	String	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	

String	Result	String	Result
`'vback2'` (default)		`'rose'`	
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`Head1Length` — Length of first arrowhead

10 (default) | scalar numeric value

Length of the first arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The first arrowhead extends backwards from the point `(x_begin,y_begin)` determined by the X and Y properties.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Head2Length` — Length of second arrowhead

10 (default) | scalar numeric value

Length of the second arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The second arrowhead extends backwards from the point `(x_end,y_end)` determined by the X and Y properties.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Head1Width` — Width of first arrowhead

10 (default) | scalar numeric value

Width of the first arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The first arrowhead is located at the point `(x_begin,y_begin)` determined by the X and Y properties.

Example: 15

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Head2Width — Width of second arrowhead

10 (default) | scalar numeric value

Width of the second arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The second arrowhead is located at the point (x_end,y_end) determined by the X and Y properties.

Example: 15

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Location and Size

X — Beginning and ending x-coordinates

[0.3 0.4] (default) | two-element vector

Beginning and ending *x*-coordinates, specified as a two-element vector of the form [x_begin x_end].

By default, the units are normalized to the figure. The lower-left corner of the figure maps to (0,0) and the upper-right corner maps to (1,1). To change the units, use the Units property.

Example: [0.2 0.3]

Y — Beginning and ending y-coordinates

[0.3 0.4] (default) | two-element vector

Beginning and ending *y*-coordinates, specified as a two-element vector of the form [y_begin y_end].

By default, the units are normalized to the figure. The lower-left corner of the figure maps to (0,0) and the upper-right corner maps to (1,1). To change the units, use the Units property.

Example: [0.2 0.3]

`Position` — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin dx dy]`. The first two elements specify the coordinates for the beginning of the arrow. The second two elements specify the slope of the arrow.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.5 0.5 0.2 0.3]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

`Units`	Description
`'normalized'` (default)	Normalized with respect to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the Position property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume Units is set to the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the annotation double arrow, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

See Also

`annotation`

More About

- "Access Property Values"
- "Graphics Object Properties"

Annotation Ellipse Properties

Control annotation ellipse appearance and behavior

Annotation ellipse properties control the appearance and behavior of an annotation ellipse object. By changing property values, you can modify certain aspects of the ellipse.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('ellipse');
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Color` — Outline color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Outline color, specified as an RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the `Color` as `'none'`, then the annotation ellipse outline is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`

Long Name	Short Name	RGB Triplet
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

FaceColor — Fill color

'none' (default) | RGB triplet | color string

Fill color, specified as 'none', an RGB triplet, or a color string. The 'none' option makes the fill invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

LineStyle — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width of ellipse outline, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Location and Size

`Position` — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x y length height]`. The first two elements specify the coordinates of the lower-left corner of the smallest rectangle that enclose the ellipse. The second two elements specify length and height of the rectangle.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3 0.4 0.5]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the Position property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume Units is set to the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the annotation ellipse, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

See Also

`annotation`

More About

- “Access Property Values”
- “Graphics Object Properties”

Annotation Line Properties

Control annotation line appearance and behavior

Annotation line properties control the appearance and behavior of an annotation line object. By changing property values, you can modify certain aspects of the line.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('line');
c = h.Color;
h.Color = 'red';
```

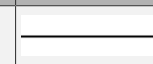
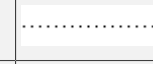
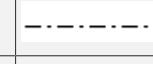
If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | color string

Line color, specified as an RGB triplet or a color string. The default RGB triplet value of `[0 0 0]` corresponds to black.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

Location and Size

`X` — Beginning and ending x-coordinates

`[0.3 0.4]` (default) | two-element vector

Beginning and ending *x*-coordinates, specified as a two-element vector of the form `[x_begin x_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

Y — Beginning and ending y-coordinates

`[0.3 0.4]` (default) | two-element vector

Beginning and ending *y*-coordinates, specified as a two-element vector of the form `[y_begin y_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

Position — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin dx dy]`. The first two elements specify the coordinates of the starting point of the line. The second two elements specify the slope of the line.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property. To change the units, use the `Units` property.

Example: `[0.2 0.2 0.3 0.3]`

Units — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.

Units	Description
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the Position property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume Units is set to the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the annotation line, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

See Also

`annotation`

More About

- "Access Property Values"
- "Graphics Object Properties"

Annotation Rectangle Properties

Control annotation rectangle appearance and behavior

Annotation rectangle properties control the appearance and behavior of an annotation rectangle object. By changing property values, you can modify certain aspects of the rectangle.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('rectangle');
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Color` — Outline color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Outline color, specified as an RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the `Color` as `'none'`, then the annotation rectangle outline is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`

Long Name	Short Name	RGB Triplet
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

FaceColor — Fill color

'none' (default) | RGB triplet | color string

Fill color, specified as 'none', an RGB triplet, or a color string. The 'none' option makes the fill invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

FaceAlpha — Fill transparency

1 (default) | value in range [0,1]

Fill transparency, specified as a scalar value in the range [0,1]. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	—·—·—·—
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width of rectangle outline, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Location and Size

`Position` — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x y length height]`. The first two elements specify the coordinates of the lower left corner of the rectangle. The second two elements specify the length and height of the rectangle.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3 0.4 0.5]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

`Units`	Description
`'normalized'` (default)	Normalized with respect to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the Position property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume Units is set to the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the annotation rectangle, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

See Also

`annotation`

More About

- "Access Property Values"
- "Graphics Object Properties"

Annotation Text Arrow Properties

Control annotation text arrow appearance and behavior

Annotation text arrow properties control the appearance and behavior of an annotation text arrow object. By changing property values, you can modify certain aspects of the text arrow.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('textarrow');
s = h.FontSize;
h.FontSize = 12;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Text

`String` — Text to display

`''` (default) | character array | cell array | numeric value

Text to display, specified as a character array, cell array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: `123`

To include numeric variables with text, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the Interpreter property.

To create multiline text:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.

- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create a string with a new line character, such as `sprintf('first line \n second line')`. This property converts strings with new line characters to cell arrays.

Text that contains only a numeric value is converted to a string using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note: The words `default`, `factory`, and `remove` are reserved words that will not appear in text when quoted as a normal string. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`

Modifier	Description	Example of String
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsui`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrighta	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	°
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\vee	∨	\langle	⟨	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

`TextColor` — Text color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you set the color to `'none'`, then the text is invisible.

Note: Setting the Color property changes the `TextColor` property to the same value, unless you explicitly set the `TextColor` property.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

TextRotation — Text rotation angle in degrees

0 (default) | scalar numeric value

Text rotation angle in degrees, specified as a scalar numeric value. Set this property to a positive value to rotate the text counterclockwise. Angles are absolute and not relative to previous rotations. A rotation of 0 degrees is always horizontal.

Example: 90

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Font Style

FontAngle — Character slant

'normal' (default) | 'italic'

Character slant, specified as 'normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Note: The 'oblique' value has been removed. Use 'italic' instead.

FontName — Font name

'Helvetica' (default) | 'FixedWidth' | system supported font name

Font name, specified as the name of the font to use or the string 'FixedWidth'. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`FontSize` — Font size

`10` (default) | scalar value greater than 0

Font size, specified as a scalar value greater than 0 in point units. One point equals 1/72 inch. To change the font units, use the FontUnits property.

Example: `12`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`FontUnits` — Font size units

`'points'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

`Units`	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'normalized'`	Interpreted as a fraction of the axes height. If you resize the axes, MATLAB modifies the font size accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the axes height.

Units	Description
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

If you set both the font size and the font units in one function call, you must set the FontUnits property first so that the axes correctly interprets the specified font size.

`FontWeight` — Thickness of text characters

`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

Text Box

`TextLineWidth` — Width of text box outline

`0.5` (default) | scalar numeric value

Width of text box outline, specified as a scalar numeric value in point units. One point equals `1/72` inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TextEdgeColor` — Color of text box outline

`'none'` (default) | RGB triplet | color string

Color of text box outline, specified as `'none'`, a three-element RGB triplet, or a color string. If the color is `'none'`, the box outline is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`TextBackgroundColor` — Color of text box background

`'none'` (default) | RGB triplet | color string

Color of text box background, specified as `'none'`, a three-element RGB triplet, or a color string. If the color is `'none'`, the background color is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`

Long Name	Short Name	RGB Triplet
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

TextMargin — Space around text within text box

2 (default) | scalar numeric value

Space around the text within the text box, specified as a scalar numeric value in pixel units.

Example: 10

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Arrow

Color — Arrow color

[0 0 0] (default) | RGB triplet | color string | 'none'

Arrow color, specified as a three-element RGB triplet, a color string, or 'none'. The default RGB triplet value of [0 0 0] corresponds to black. If you set the color to 'none', then the arrow is invisible.

Note: Setting this property also changes the text color if you have not explicitly set the text color using the TextColor property.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

LineStyle — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
'-'	Solid line	————
'--'	Dashed line	- - - - -
':'	Dotted line	.....
'-.'	Dash-dotted line	_._._._.-
'none'	No line	No line

LineWidth — Width of arrow stem

0.5 (default) | scalar numeric value

Width of arrow stem, specified as a scalar numeric value greater than zero in point units. One point equals 1/72 inch. The default value is 0.5 points.

Example: 0.75

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`HeadStyle` — Style of arrowhead

`'vback2'` (default) | head style string

Style of the arrowhead, specified as one of the head style strings in this table.

String	Result	String	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	
`'vback2'` (default)		`'rose'`	
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`HeadLength` — Length of arrowhead

10 (default) | scalar numeric value

Length of the arrowhead, specified as a scalar numeric value in point units. One point equals `1/72` inch.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`HeadWidth` — Width of arrowhead

10 (default) | scalar numeric value

Width of the arrowhead, specified as a scalar numeric value in point units. One point equals `1/72` inch.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Location and Size

`X` — Beginning and ending x-coordinates

`[0.3 0.4]` (default) | two-element vector

Beginning and ending *x*-coordinates for the arrow, specified as a two-element vector of the form `[x_begin x_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

`Y` — Beginning and ending y-coordinates

`[0.3 0.4]` (default) | two-element vector

Beginning and ending *y*-coordinates for the arrow, specified as a two-element vector of the form `[y_begin y_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

`Position` — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin length height]`. The first two elements specify the coordinates of the beginning of the arrow. The second two elements specify the length and height of the arrow. The text box extends from the beginning of the arrow.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.2 0.3 0.1]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

`Units`	Description
`'normalized'` (default)	Normalized with respect to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the figure window.

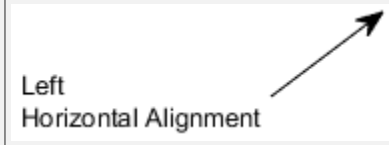
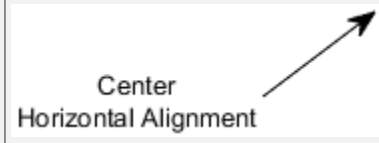
This property affects the Position property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume Units is set to the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the annotation text arrow, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

`HorizontalAlignment` — Horizontal alignment of text

`'left'` (default) | `'center'` | `'right'`

Horizontal alignment of the text, specified as one of the values in this table. This property is useful when aligning multiple lines of text.

Value	Result
`'left'`	
`'center'`	
`'right'`	

`VerticalAlignment` — Vertical alignment of text with respect to arrow

`'top'` (default) | `'cap'` | `'middle'` | `'baseline'` | `'bottom'`

Vertical alignment of the text with respect to the end of the arrow, specified as `'top'`, `'cap'`, `'middle'`, `'baseline'`, or `'bottom'`.

See Also

`annotation`

More About

- “Access Property Values”
- “Graphics Object Properties”

Annotation Text Box Properties

Control annotation text box appearance and behavior

Annotation text box properties control the appearance and behavior of an annotation text box object. By changing property values, you can modify certain aspects of the text box.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('textbox');
s = h.FontSize;
h.FontSize = 12;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Text

`String` — Text to display

`''` (default) | character array | cell array | numeric value

Text to display, specified as a character array, cell array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: `123`

To include numeric variables with text, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the Interpreter property.

To create multiline text:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.

- Use `sprintf` to create a string with a new line character, such as `sprintf('first line \n second line')`. This property converts strings with new line characters to cell arrays.

Text that contains only a numeric value is converted to a string using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note: The words `default`, `factory`, and `remove` are reserved words that will not appear in text when quoted as a normal string. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`

Modifier	Description	Example of String
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsui`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ε	`\Delta`	Δ	`\spadesuit`	♠

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\zeta	ζ	\Theta	Θ	\leftrighta	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	°
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

Font Style

`Color` — Text color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you set the color to `'none'`, then the text is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`

Long Name	Short Name	RGB Triplet
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

FontAngle — Character slant

'normal' (default) | 'italic'

Character slant, specified as 'normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Note: The 'oblique' value has been removed. Use 'italic' instead.

FontName — Font name

'Helvetica' (default) | 'FixedWidth' | system supported font name

Font name, specified as the name of the font to use or the string 'FixedWidth'. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string 'FixedWidth'. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The 'FixedWidth' value relies on the root FixedWidthFontName property. Setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

Example: 'Cambria'

FontSize — Font size

10 (default) | scalar value greater than 0

Font size, specified as a scalar value greater than 0 in point units. One point equals 1/72 inch. To change the font units, use the FontUnits property.

Example: 12

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

`FontUnits` — Font size units

`'points'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'normalized'`	Interpreted as a fraction of the axes height. If you resize the axes, MATLAB modifies the font size accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the axes height.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

If you set both the font size and the font units in one function call, you must set the FontUnits property first so that the axes correctly interprets the specified font size.

`FontWeight` — Thickness of text characters

`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

Text Box

`LineStyle` — Line style of box outline

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style of box outline, specified as one of the strings listed in this table.

String	Line Style
`'-'`	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line
`'none'`	Box outline is invisible

`LineWidth` — Width of box outline

`0.5` (default) | scalar numeric value

Width of box outline, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`EdgeColor` — Color of box outline

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Color of box outline, specified as a three-element RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you set the color to `'none'`, then the box outline is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

`[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`BackgroundColor` — Color of text box background

`'none'` (default) | RGB triplet | color string

Color of text box background, specified as `'none'`, a three-element RGB triplet, or a color string. If you set the background color to `'none'`, then the background is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`

Long Name	Short Name	RGB Triplet
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

FaceAlpha — Transparency of background color

1 (default) | scalar value between 0 and 1 inclusive

Transparency of the background color, specified as a scalar value between 0 and 1. If the value is 1, then the color is opaque. To add transparency, set the property to a value closer to 0, where 0 is completely transparent.

FitBoxToText — Option to fit box width and height to text

'on' (default) | 'off'

Option to fit the box width and height to the text, specified as one of these values:

- 'on' — Resize the text box to fit the text.
- 'off' — Wrap the text strings to fit the width of the text box. Wrapping can cause some of the text string to extend below the text box.

If you resize a text box when in plot edit mode, or if you change the Position property, then the FitBoxToText property changes to 'off'.

Margin — Space around text within the text box

5 (default) | scalar numeric value

The space around the text within the text box, specified as a scalar numeric value in pixel units.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Location and Size

Position — Size and location

[0.3 0.3 0.1 0.1] (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin length height]`. The first two elements specify the coordinates for the lower-left corner of the text box. The second two elements specify the length and height of the text box.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Note: If the `FitBoxToText` property is set to `'on'` and you change the `String` property, then the `Position` property might not reflect the latest changes until the next time the screen refreshes. To ensure that the position value reflects the latest changes, call `drawnow` before querying the position when working in a script or function.

Example: `[0.2 0.3 0.4 0.5]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

`Units`	Description
`'normalized'` (default)	Normalized with respect to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the Position property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume Units is set to the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the annotation text box, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

`HorizontalAlignment` — Horizontal alignment of text within text box

`'left'` (default) | `'center'` | `'right'`

Horizontal alignment of the text within the text box, specified as one of the values in this table.

Value	Result
`'left'`	
`'center'`	
`'right'`	

`VerticalAlignment` — Vertical alignment of text within text box

`'top'` (default) | `'middle'` | `'bottom'` | `'baseline'` | `'cap'`

Vertical alignment of the text within the text box, specified as one of the values in this table.

Value	Result
`'top'` or `'cap'`	

Value	Result
`'middle'`	
`'bottom'` or `'baseline'`	

See Also

`annotation`

More About

- "Access Property Values"
- "Graphics Object Properties"

ans

Most recent answer

Syntax

```
ans
```

Description

The MATLAB software creates the `ans` variable automatically when you specify no output argument.

Examples

The statement

```
2+2
```

is the same as

```
ans = 2+2
```

See Also

`display`

Introduced before R2006a

any

Determine if any array elements are nonzero

Syntax

```
B = any(A)
B = any(A,dim)
```

Description

`B = any(A)` tests along the first array dimension of `A` whose size does not equal 1, and determines if any element is a nonzero number or logical `1` (`true`). In practice, `any` is a natural extension of the logical OR operator.

- If `A` is a vector, then `B = any(A)` returns logical `1` (`true`) if any of the elements of `A` is a nonzero number or is logical `1`, and returns logical `0` (`false`) if all the elements are zero.
- If `A` is a nonempty, nonvector matrix, then `B = any(A)` treats the columns of `A` as vectors, returning a row vector of logical `1`s and `0`s.
- If `A` is an empty 0-by-0 matrix, `any(A)` returns logical `0` (`false`).
- If `A` is a multidimensional array, `any(A)` acts along the first array dimension whose size does not equal 1 and returns an array of logical values. The size of this dimension becomes `1`, while the sizes of all other dimensions remain the same.

`B = any(A,dim)` tests elements along dimension `dim`. The dim input is a positive integer scalar.

Examples

Test Matrix Columns

Create a 3-by-3 matrix.

```
A = [0 0 3;0 0 3;0 0 3]
```

```
A =

     0     0     3
     0     0     3
     0     0     3
```

Test each column for nonzero elements.

```
B = any(A)

B =

     0     0     1
```

Reduce a Logical Vector to a Single Condition

Create a vector of decimal values and test which values are less than 0.5.

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69];
B = (A < 0.5)

B =

0   0   1   1   1   1   0
```

The output is a vector of logical values. The `any` function reduces such a vector of logical values to a single condition. In this case, `B = any(A < 0.5)` yields logical `1`.

This makes `any` particularly useful in `if` statements.

```
if any(A < 0.5)
    %do something
else
    %do something else
end
```

The code is executed depending on a single condition, rather than a vector of possibly conflicting conditions.

Test Arrays of Any Dimension

Create a 3-by-7-by-5 multidimensional array and test to see if any of its elements are greater than 3.

```
A = rand(3,7,5) * 5;
```

```
B = any(A(:) > 3)

B =

     1
```

You can also test the array for elements that are less than zero.

```
B = any(A(:) < 0)

B =

     0
```

The syntax `A(:)` turns the elements of `A` into a single column vector, so you can use this type of statement on an array of any size.

Test Matrix Rows

Create a 3-by-3 matrix.

```
A = [0 0 3;0 0 3;0 0 3]

A =

     0     0     3
     0     0     3
     0     0     3
```

Test the rows of `A` for nonzero elements by specifying `dim = 2`.

```
B = any(A,2)

B =

     1
     1
     1
```

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The `any` function ignores elements of `A` that are `NaN` (Not a Number).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

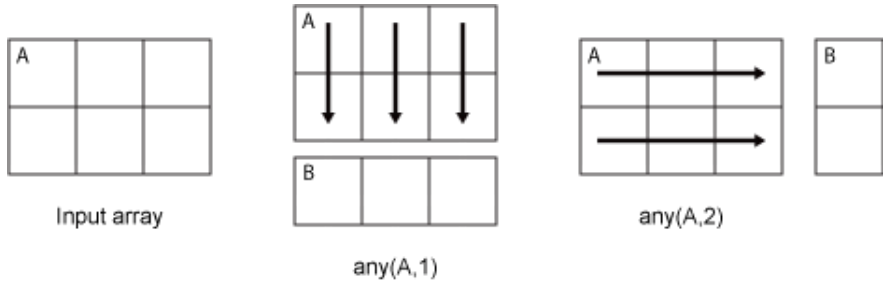
`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `any(A,1)` works on successive elements in the columns of `A` and returns a row vector of logical values.
- `any(A,2)` works on successive elements in the rows of `A` and returns a column vector of logical values.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`B` — Logical array

scalar | vector | matrix | multidimensional array

Logical array, returned as a scalar, vector, matrix, or multidimensional array. The dimension of `A` acted on by `any` has size `1` in `B`.

More About

- "Reduce Logical Arrays to Single Value"

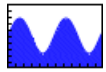
See Also

`all` | `colon (:)` | `or` | `prod` | `sum`

Introduced before R2006a

area

Filled area 2-D plot



Syntax

```
area(Y)
area(X,Y)
area(...,basevalue)
area(...,'PropertyName',PropertyValue,...)
area(axes_handle,...)
h = area(...)
```

Description

An area graph displays elements in `Y` as one or more curves and fills the area beneath each curve. When `Y` is a matrix, the curves are stacked showing the relative contribution of each row element to the total height of the curve at each x interval.

`area(Y)` plots the vector `Y` or plots each column in matrix `Y` as a separate curve and stacks the curves. The *x*-axis automatically scales to `1:size(Y,1)`.

`area(X,Y)` For vectors `X` and `Y`, `area(X,Y)` is the same as `plot(X,Y)` except that the area between `0` and `Y` is filled. When `Y` is a matrix, `area(X,Y)` plots the columns of `Y` as filled areas. For each `X`, the net result is the sum of corresponding values from the rows of `Y`.

If `X` is a vector, `length(X)` must equal `length(Y)`. If `X` is a matrix, `size(X)` must equal `size(Y)`.

`area(...,basevalue)` specifies the base value for the area fill. The default `basevalue` is `0`.

`area(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the patch graphics object created by `area`.

`area(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = area(...)` returns handles of `area` graphics objects.

Creating an area graph of an *m*-by-*n* matrix creates *n* area objects (that is, one per column), whereas a 1-by-*n* vector creates one area object.

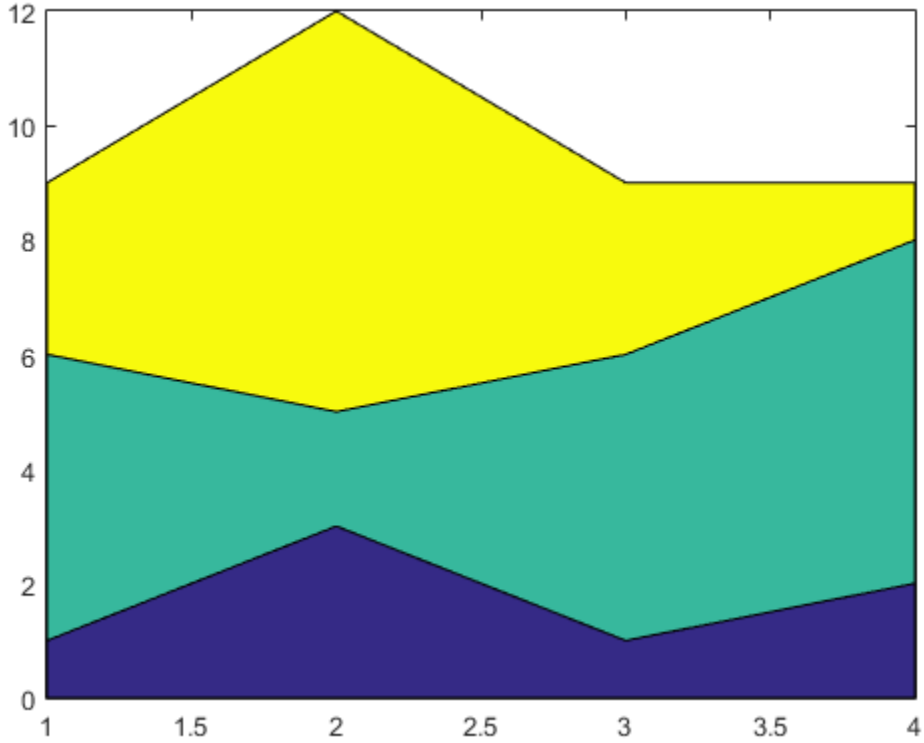
Some area object properties that you set on an individual area object set the values for all area objects in the graph. See Area Properties for information on specific properties.

Examples

Create Area Graph

Plot the data in matrix Y as an area graph.

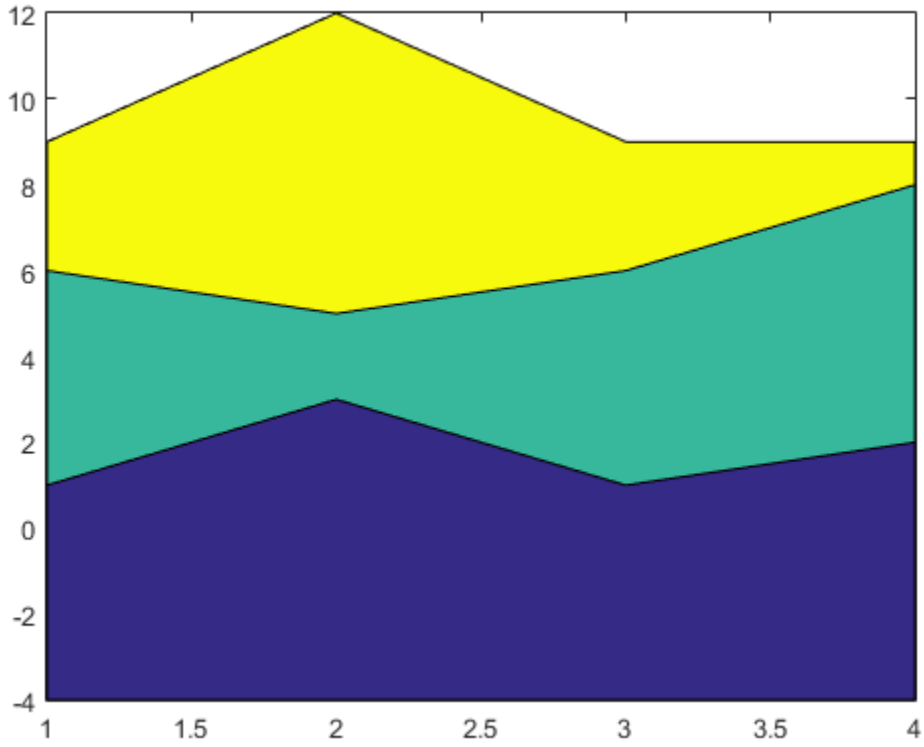
```
Y = [1, 5, 3;
     3, 2, 7;
     1, 5, 3;
     2, 6, 1];
figure
area(Y)
```



Adjust Base Value of Area Graph

By default, `area` uses the *y*-axis as the base value. Change the base value by setting the `basevalue` input argument to -4.

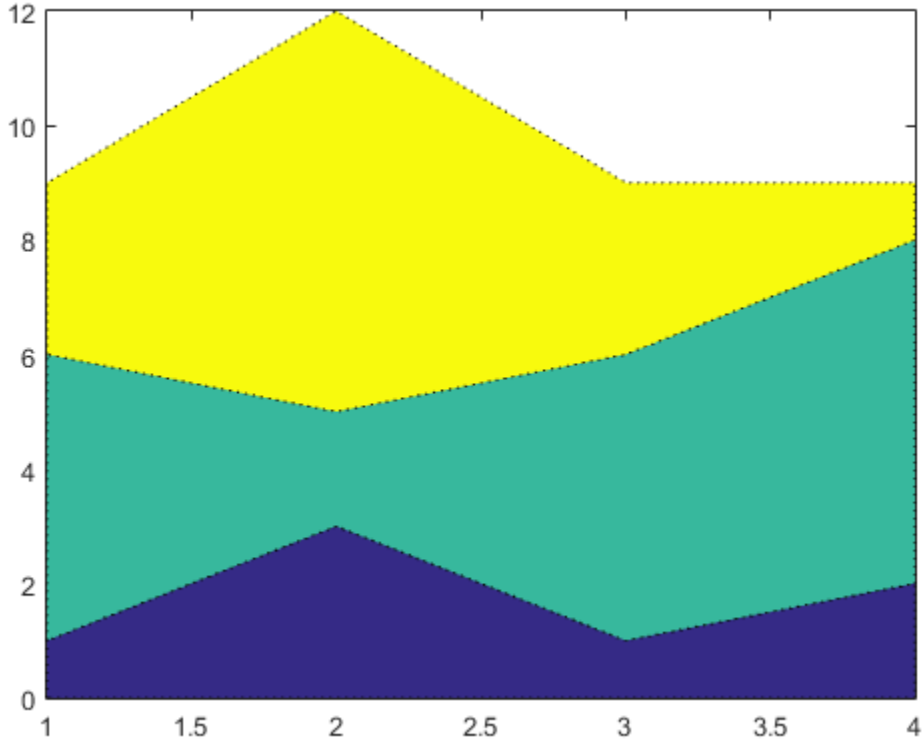
```
Y = [1, 5, 3;
     3, 2, 7;
     1, 5, 3;
     2, 6, 1];
figure
basevalue = -4;
area(Y,basevalue)
```



Specify Color and Line Style for Area Plot

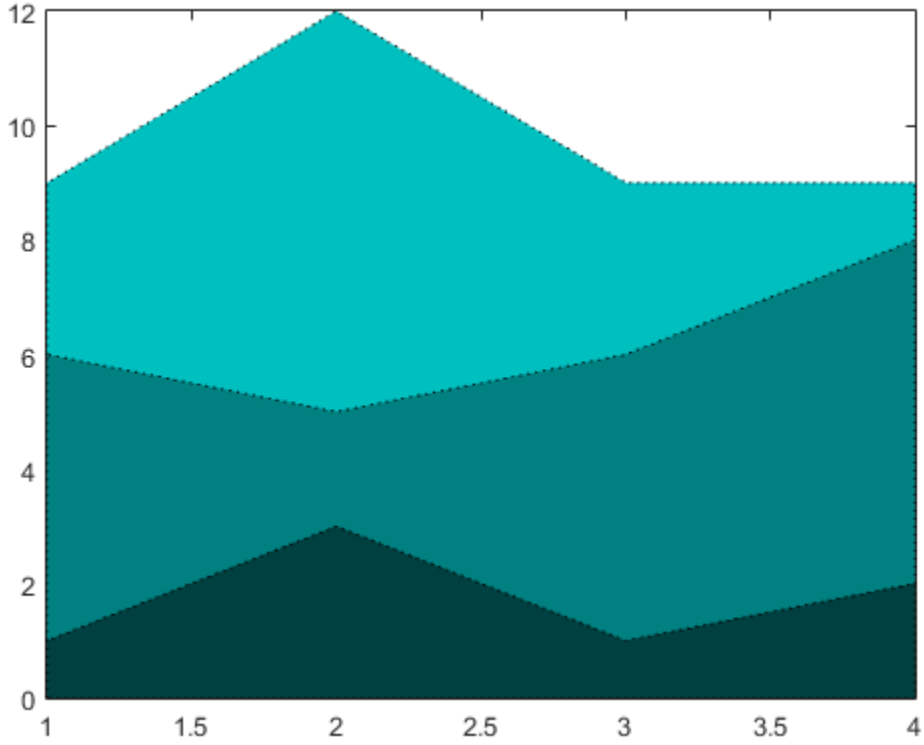
Create an area plot of `Y` and use a dotted line style. Return the three area objects in array `h`. The `area` function creates one area object for each column in `Y`.

```
Y = [1, 5, 3;
     3, 2, 7;
     1, 5, 3;
     2, 6, 1];
h = area(Y,'LineStyle',':');
```



Change the area colors using RGB triplet color values. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
h(1).FaceColor = [0 0.25 0.25];
h(2).FaceColor = [0 0.5 0.5];
h(3).FaceColor = [0 0.75 0.75];
```



More About

- “Compare Data Sets Using Overlayed Area Graphs”

See Also

Functions
`bar` | `plot` | `sort`

Properties
Area Properties

Introduced before R2006a

Area Properties

Control area appearance and behavior

Area properties control the appearance and behavior of an area object. By changing property values, you can modify certain aspects of the area.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = area(1:10);
c = h.EdgeColor;
h.EdgeColor = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`EdgeColor` — Area outline color

`[0 0 0]` (default) | RGB triplet | color string | `'none'` | `'flat'`

Area outline color, specified as one of these values:

- `'none'` — Do not draw the outline.
- `'flat'` — Use colors from the axes colormap.
- RGB triplet or a color string — Specify a custom color. The default RGB triplet value of `[0 0 0]` corresponds to black.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

FaceColor — Area fill color

'flat' (default) | 'none' | RGB triplet | color string

Area fill color, specified as one of these values:

- 'flat' — Use colors from the axes colormap.
- 'none' — Do not use any color for the fill, which allows the background to show through.
- RGB triplet or a color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]

Long Name	Short Name	RGB Triplet
'black	'k'	[0 0 0]

LineStyle — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
'-'	Solid line	————
'--'	Dashed line	- - - - -
':'	Dotted line	.....
'-.'	Dash-dotted line	—·—·—·—
'none'	No line	No line

LineWidth — Area outline width

0.5 (default) | scalar numeric value

Area outline width, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: 1.5

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

AlignVertexCenters — Sharp vertical and horizontal lines

'off' (default) | 'on'

Sharp vertical and horizontal lines, specified as 'off' or 'on'.

If the associated figure has a GraphicsSmoothing property set to 'on' and a Renderer property set to 'opengl', then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the AlignVertexCenters property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Baseline

`BaseLine` — Baseline

baseline object

Baseline object. For a list of baseline properties, see Baseline Properties.

`BaseValue` — Baseline location

`0` (default) | numeric scalar value

Baseline location, specified as a numeric scalar value.

`ShowBaseLine` — Baseline visibility

`'on'` (default) | `'off'`

Baseline visibility, specified as one of these values:

- `'on'` — Show the baseline.
- `'off'` — Hide the baseline.

Data

`XData` — Values along x-axis

`[ ]` (default) | vector

Values along *x*-axis, specified as a vector. The input argument `X` to the `area` function determines the *x*-values. If you do not specify `X`, then `area` uses the indices of the values in `YData` to set the *x*-values. `XData` and `YData` must have equal lengths.

Setting this property sets the associate mode property XDataMode to manual.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`YData` — Values along y-axis

`[ ]` (default) | vector

Values along *y*-axis, specified as a vector. The input argument `Y` to the `area` function determines the *y*-values. `XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`XDataSource` — Variable linked to `XData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`XDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData`.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or pass the input argument `X` to the `area` function.

Visibility

`Visible` — Visibility of area

`'on'` (default) | `'off'`

Visibility of area, specified as one of these values:

- `'on'` — Display the area.
- `'off'` — Hide the area without deleting it. You still can access the properties of an invisible area object.

`Clipping` — Clipping of area to axes limits

`'on'` (default) | `'off'`

Clipping of area to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the area that are outside the axes limits.
- `'off'` — Display the entire area, even if parts of it appear outside the axes limits. Parts of the area might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the area that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'area'`

Type of graphics object, returned as `'area'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Tag to associate with area

`''` (default) | string

Tag to associate with the area, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

Data Types: `char`

`UserData` — Data to associate with area

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the area object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the area.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form

`'dataN'`, where `N` is the number assigned to the area object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the area from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the area object in the legend as one entry (default).
   - `'off'` — Do not include the area object in the legend.
   - `'children'` — Include only children of the area object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of area

axes object | group object | transform object

Parent of area, specified as an axes, group, or transform object.

`Children` — Children of area

empty `GraphicsPlaceholder` array

The area has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of area object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The area object handle is always visible.
- `'off'` — The area object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The area object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the area at the command-line, but allows callback functions to access it.

If the area object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the area. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The area object — You can access properties of the area object from within the callback function.

- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the area. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the area when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the area.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.

- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The Visible property must be set to `'on'`. The HitTest property determines if the area responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the area passes the click to the object below it in the current view of the figure window. The HitTest property of the area has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the area. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the area that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the area object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for area

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for area, specified as one of these values:

- `'off'` — Click the area plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the area plot to select it, that is, anywhere within the rectangle that encloses the area plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the area is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the area tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the area. Setting the `CreateFcn` property on an existing area has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during area creation. MATLAB executes the callback after creating the area and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The area object — You can access properties of the area object from within the callback function. You also can access the area object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the area. MATLAB executes the callback before destroying the area so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The area object — You can access properties of the area object from within the callback function. You also can access the area object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of area

`'off'` (default) | `'on'`

Deletion status of area, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the area begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the area no longer exists.

Check the value of the BeingDeleted property to verify that the area is not about to be deleted before querying or modifying it.

See Also

`area`

More About

- "Access Property Values"
- "Graphics Object Properties"

array2table

Convert homogeneous array to table

Syntax

```
T = array2table(A)
T = array2table(A,Name,Value)
```

Description

`T = array2table(A)` converts the `m`-by-`n` array, `A`, to an `m`-by-`n` table, `T`. Each column of `A` becomes a variable in `T`.

`array2table` uses the input array name appended with the column number for the variable names in the table. If these names are not valid MATLAB identifiers, `array2table` uses strings of the form `'Var1',...,'VarN'`, where *N* is the number of columns in `A`.

`T = array2table(A,Name,Value)` creates a table from an array, `A`, with additional options specified by one or more Name,Value pair arguments.

For example, you can specify row names or variable names to include in the table.

Examples

Convert Numeric Array to Table

Create an array of numeric data.

```
A = [1 4 7; 2 5 8; 3 6 9]
```

```
A =

     1     4     7
     2     5     8
```

```
     3     6     9
```

Convert the array, A, to a table.

```
T = array2table(A)
```

```
T =

    A1    A2    A3
    __    __    __

    1     4     7
    2     5     8
    3     6     9
```

The table has variable names that append the column number to the input array name, A.

Convert Array to Table Including Variable Names

Create an array of numeric data.

```
A = [1 12 30.48; 2 24 60.96; 3 36 91.44]
```

```
A =

    1.0000   12.0000   30.4800
    2.0000   24.0000   60.9600
    3.0000   36.0000   91.4400
```

Convert the array, A, to a table and include variable names.

```
T = array2table(A,...
    'VariableNames',{'Feet' 'Inches' 'Centimeters'})
```

```
T =

    Feet    Inches    Centimeters
    ____    ______    ___________

    1       12        30.48
    2       24        60.96
    3       36        91.44
```

- “Access Data in a Table”

Input Arguments

A — Input array

matrix

Input array, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `cell`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RowNames',{'row1','row2','row3'}` uses the row names, `row1`, `row2`, and `row3` for the table, `T`.

'RowNames' — Row names for T

`{}` (default) | cell array of nonempty, distinct strings

Row names for `T`, specified as the comma-separated pair consisting of `'RowNames'` and a cell array of nonempty, distinct strings. The number of strings must equal the number of rows, `size(A,1)`.

'VariableNames' — Variable names for T

cell array of nonempty, distinct strings

Variable names for `T`, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of nonempty, distinct strings. The number of strings must equal the number of variables, `size(A,2)`.

Furthermore, the strings must be valid MATLAB identifiers. If valid MATLAB identifiers are not available for use as variable names, MATLAB uses a cell array of *N* strings of the form `{'Var1' ... 'Var`*N*`'}` where *N* is the number of variables. You can determine valid MATLAB variable names using the function `isvarname`.

Output Arguments

T — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

More About

Tips

- If `A` is a cell array, use `cell2table(A)` to create a table from the contents of the cells in `A`. Each variable in the table is numeric or a cell array of strings. `array2table(A)` creates a table where each variable is a column of cells.

See Also

`cell2table` | `isvarname` | `struct2table` | `table` | `table2array`

arrayfun

Apply function to each element of array

Syntax

```
[B1,...,Bm] = arrayfun(func,A1,...,An)
[B1,...,Bm] = arrayfun(func,A1,...,An,Name,Value)
```

Description

`[B1,...,Bm] = arrayfun(func,A1,...,An)` calls the function specified by function handle `func` and passes elements from arrays `A1,...,An`, where `n` is the number of inputs to function `func`. Output arrays `B1,...,Bm`, where `m` is the number of outputs from function `func`, contain the combined outputs from the function calls. The `i`th iteration corresponds to the syntax `[B1(i),...,Bm(i)] = func(A1(i),...,An(i))`. The `arrayfun` function does not perform the calls to function `func` in a specific order.

`[B1,...,Bm] = arrayfun(func,A1,...,An,Name,Value)` calls function `func` with additional options specified by one or more Name,Value pair arguments. Possible values for `Name` are `'UniformOutput'` or `'ErrorHandler'`.

Input Arguments

`func`

Handle to a function that accepts `n` input arguments and returns `m` output arguments.

If function `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

`A1,...,An`

Arrays that contain the `n` inputs required for function `func`. Each array must have the same dimensions. Arrays can be numeric, character, logical, cell, structure, or user-defined object arrays.

If any input `A` is a user-defined object array, and you overloaded the `subsref` or `size` methods, `arrayfun` requires that:

- The `size` method returns an array of type `double`.
- The object array supports linear indexing.
- The product of the sizes returned by the `size` method does not exceed the limit of the array, as defined by linear indexing into the array.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'UniformOutput'`

Logical value, as follows:

`true (1)`	Indicates that for all inputs, each output from function `func` is a cell array or a scalar value that is always of the same type and size. The `arrayfun` function combines the outputs in arrays `B1,...,Bm`, where `m` is the number of function outputs. Each output array is of the same type as the individual function outputs.
`false (0)`	Requests that the `arrayfun` function combine the outputs into cell arrays `B1,...,Bm`. The outputs of function `func` can be of any size or type.

Default: `true`

`'ErrorHandler'`

Handle to a function that catches any errors that occur when MATLAB attempts to execute function `func`. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.

`message`	Error message text.
`index`	Linear index corresponding to the element of the input cell array at the time of the error.

- The set of input arguments to function `func` at the time of the error.

Output Arguments

`B1,...,Bm`

Arrays that collect the `m` outputs from function `func`. Each array `B` is the same size as each of the inputs `A1,...,An`.

Function `func` can return output arguments of different classes. However, if `UniformOutput` is `true` (the default):

- The individual outputs from function `func` must be scalar values (numeric, logical, character, or structure) or cell arrays.
- The class of a particular output argument must be the same for each set of inputs. The class of the corresponding output array is the same as the class of the outputs from function `func`.

Examples

To run the examples in this section, create a nonscalar structure array with arrays of different sizes in field `f1`.

```
s(1).f1 = rand(3, 6);
s(2).f1 = magic(12);
s(3).f1 = ones(5, 10);
```

Count the number of elements in each `f1` field.

```
counts = arrayfun(@(x) numel(x.f1), s)
```

The syntax `@(x)` creates an anonymous function. This code returns

```
counts =
    18   144    50
```

Compute the size of each array in the `f1` fields.

```
[nrows, ncols] = arrayfun(@(x) size(x.f1), s)
```

This code returns

```
nrows =
     3    12     5
ncols =
     6    12    10
```

Compute the mean of each column in the `f1` fields of `s`. Because the output is nonscalar, set `UniformOutput` to `false`.

```
averages = arrayfun(@(x) mean(x.f1), s, 'UniformOutput', false)
```

This code returns

```
averages =
    [1x6 double]    [1x12 double]    [1x10 double]
```

Create additional nonscalar structures `t` and `u`, and test for equality between the arrays in fields `f1` across structures `s`, `t`, and `u`.

```
t = s;  t(1).f1(:)=0;
u = s;  u(2).f1(:)=0;

same = arrayfun(@(x,y,z) isequal(x.f1, y.f1, z.f1), s, t, u)
```

This code returns

```
same =
     0     0     1
```

See Also

`structfun` | `cellfun` | `spfun` | `function_handle` | `cell2mat`

Introduced before R2006a

ascii

Class: FTP

Set FTP transfer type to ASCII

Syntax

```
ascii(ftpobj)
```

Description

`ascii(ftpobj)` sets the download and upload FTP mode to ASCII, which converts new line characters. Use this method only for text files, including HTML pages and Rich Text Format (RTF) files.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

Examples

Connect to the MathWorks FTP server, and switch from binary (default) to ASCII mode:

```
mw=ftp('ftp.mathworks.com');
ascii(mw)
```

See Also

`binary` | `ftp`

Introduced before R2006a

asec

Inverse secant in radians

Syntax

```
Y = asec(X)
```

Description

`Y = asec(X)` returns the "Inverse Secant" on page 1-456 ($\sec^{-1}$) of the elements of X. The `asec` function operates element-wise on arrays. For real elements of X in the interval `[-Inf,-1]` and `[1,Inf]`, `asec` returns values in the interval `[0,pi]`. For real values of X in the interval `[-1,1]` and for complex values of X, `asec` returns complex values. All angles are in radians.

Examples

Inverse Secant of a Value

```
asec(-2.8)
```

```
ans =
    1.9360
```

Inverse Secant of a Vector of Complex Values

Find the inverse secant of the elements of vector `x`. The `asec` function acts on `x` element-wise.

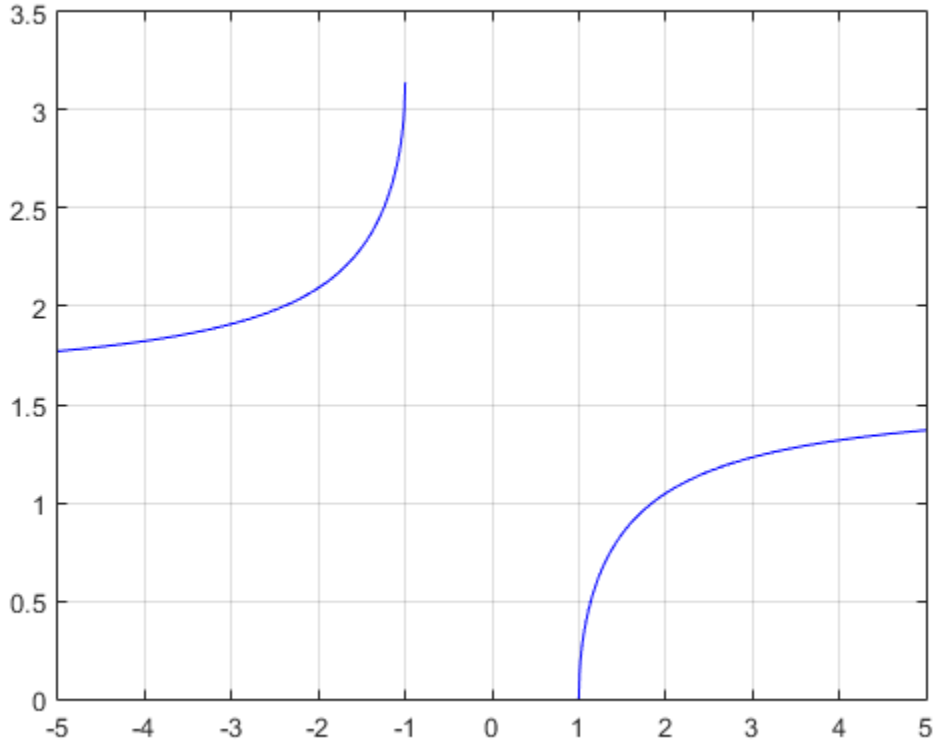
```
x = [0.5i 1+3i -2.2+i];
Y = asec(x)
```

```
Y =
   1.5708 + 1.4436i   1.4749 + 0.2970i   1.9503 + 0.1833i
```

Plot the Inverse Secant Function

Plot the inverse secant function over the intervals $-5 \le x \le -1$ and $1 \le x \le 5$.

```
x1 = -5:0.01:-1;
x2 = 1:0.01:5;
plot(x1,asec(x1),'b')
hold on
plot(x2,asec(x2),'b')
grid on
```



Input Arguments

X — Numeric input

number | vector | matrix | multidimensional array

Numeric input, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Secant

The inverse secant is defined as

$$\sec^{-1}(z) = \cos^{-1}\left(\frac{1}{z}\right).$$

See Also

`asecd` | `asech` | `sec`

Introduced before R2006a

asecd

Inverse secant in degrees

Syntax

```
Y = asecd(X)
```

Description

`Y = asecd(X)` returns the inverse secant ($\sec^{-1}$) of the elements of X in degrees. The function's domain and range include complex values. For real elements of X in the domain [-Inf,1] and [1,Inf], `asecd` returns values in the range [0, 180]. For values of X outside this range, `asecd` returns complex values.

Examples

Inverse Secant of Vector

```
x = [10 1 Inf];
y = asecd(x)
```

```
y =

   84.2608         0   90.0000
```

The `asecd` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Secant of Complex Value

```
asecd(1+i)
```

```
ans =
```

```
64.0864 +30.4033i
```

Input Arguments

X — Secant of angle

scalar value | vector | matrix | N-D array

Secant of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `asecd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

See Also

`asec` | `sec` | `secd`

Introduced before R2006a

asech

Inverse hyperbolic secant

Syntax

```
Y = asech(X)
```

Description

`Y = asech(X)` returns the inverse hyperbolic secant for each element of `X`.

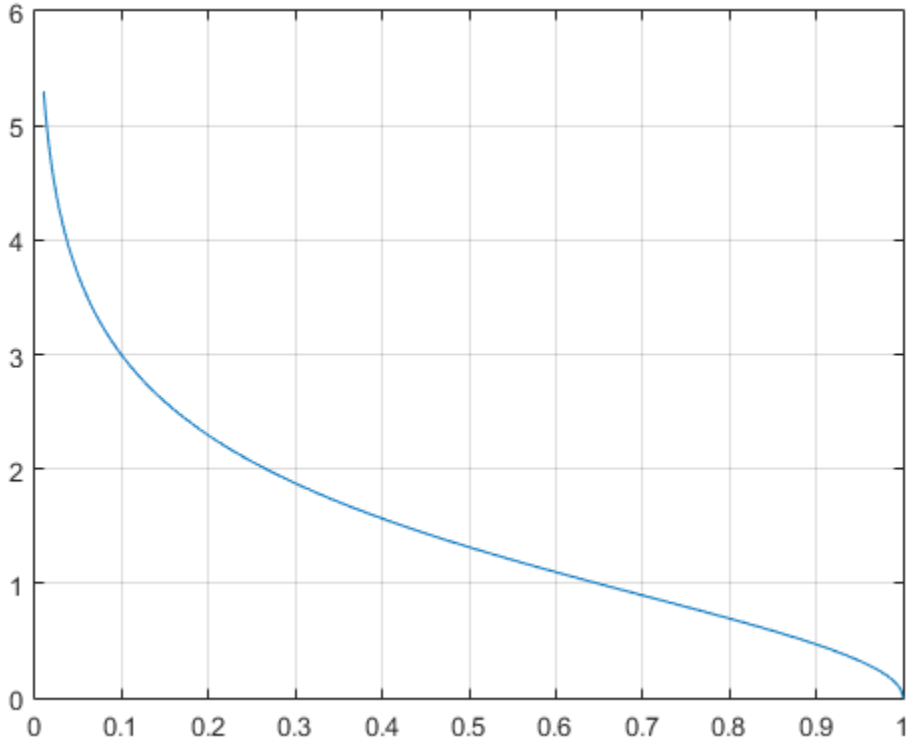
The `asech` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph of Inverse Hyperbolic Secant Function

Graph the inverse hyperbolic secant over the domain $0.01 \le x \le 1$.

```
x = 0.01:0.001:1;
plot(x,asech(x)), grid on
```



More About

Inverse Hyperbolic Secant

The inverse hyperbolic secant can be defined as

$$\operatorname{sech}^{-1}(z) = \cosh^{-1}\left(\frac{1}{z}\right).$$

See Also

`asec` | `sech` | `asinh` | `acosh`

Introduced before R2006a

asin

Inverse sine in radians

Syntax

```
y = asin(x)
```

Description

`y = asin(x)` returns the "Inverse Sine" on page 1-463 ($\sin^{-1}$) of the elements of `x`. The `asin` function operates element-wise on arrays. For real elements of `x` in the interval `[-1,1]`, `asin(x)` returns values in the interval `[-pi/2,pi/2]`. For real elements of `x` outside the interval `[-1,1]` and for complex values of `x`, `asin(x)` returns complex values. All angles are in radians.

Examples

Inverse Sine of a Value

```
asin(0.5)
```

```
ans =
    0.5236
```

Inverse Sine of a Vector of Complex Values

Find the inverse sine of the elements of vector `x`. The `asin` function acts on `x` element-wise.

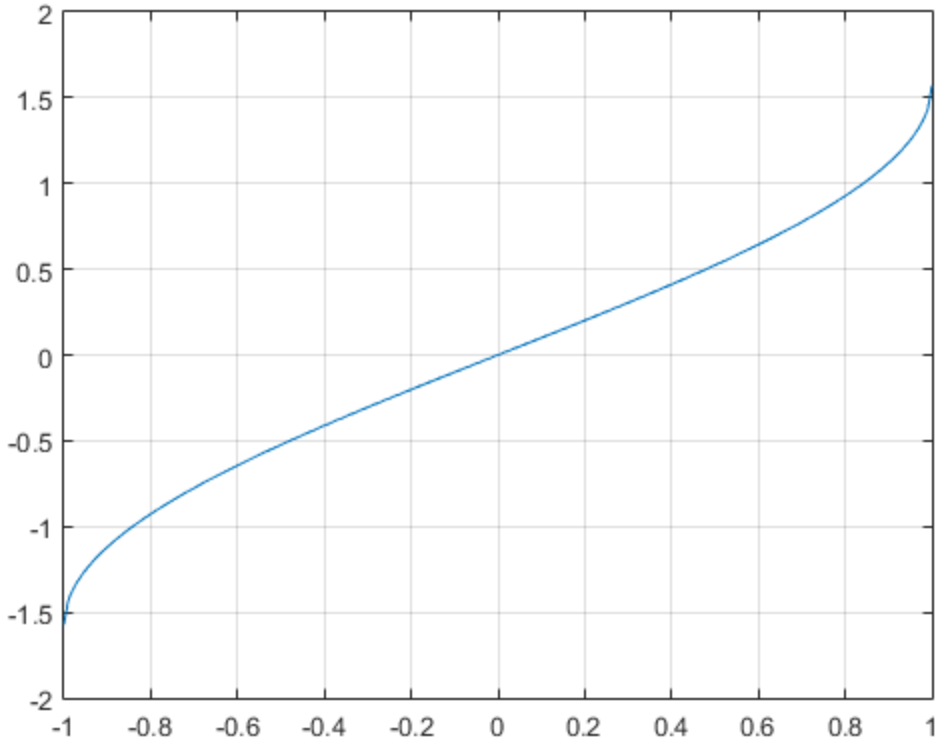
```
x = [0.5i 1+3i -2.2+i];
y = asin(x)
```

```
y =
   0.0000 + 0.4812i   0.3076 + 1.8642i  -1.1091 + 1.5480i
```

Graph of the Inverse Sine Function

Graph the inverse sine over the domain $-1 \le x \le 1$.

```
x = -1:.01:1;
plot(x,asin(x))
grid on
```



More About

Inverse Sine

The inverse sine is defined as

$$\sin^{-1}(z) = -i\log\left[iz + (1 - z^2)^{1/2}\right].$$

See Also

`asind` | `sin` | `sind`

Introduced before R2006a

asind

Inverse sine in degrees

Syntax

```
Y = asind(X)
```

Description

`Y = asind(X)` returns the inverse sine ($\sin^{-1}$) of the elements of X in degrees. The function's domain and range include complex values. For real elements of X in the domain [-1,1], `asind` returns values in the range [-90,90]. For values of X outside this range, `asind` returns complex values.

Examples

Inverse Sine of Scalar

Show that the inverse sine of 1 is exactly 90°.

```
asind(1)
```

```
ans =

    90
```

Round-Trip Calculation for Complex Angles

Show that the inverse sine, followed by sine, returns the original values of X.

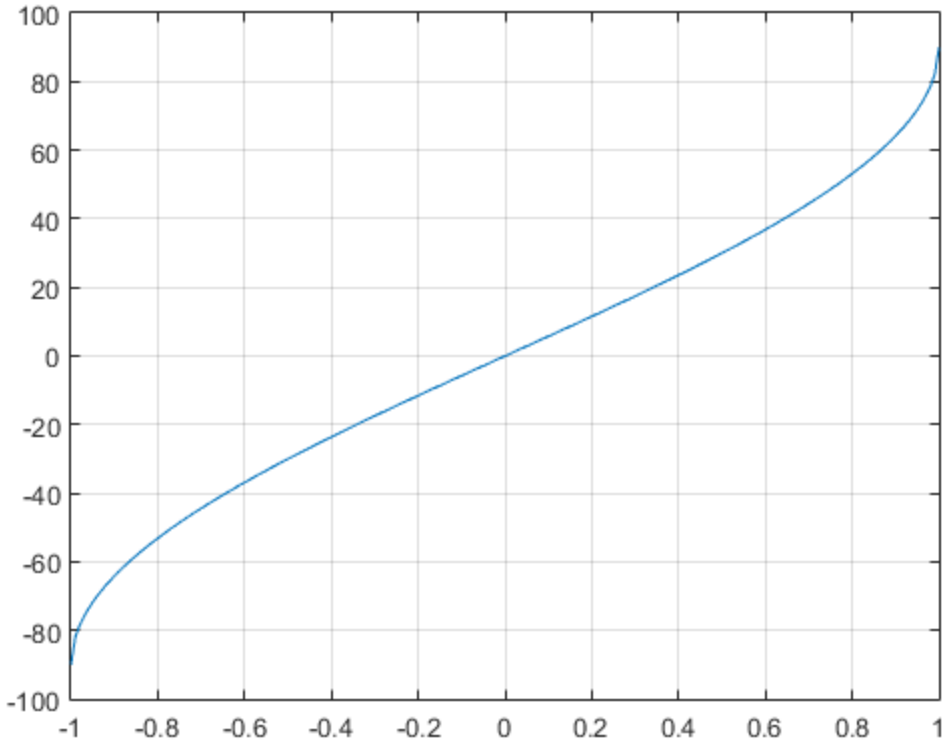
```
sind(asind([2 3]))
```

```
ans =

    2.0000    3.0000
```

Graph of Inverse Sine Function

Plot the inverse sine function over the domain $-1 \le x \le 1$.

```
x = -1:.01:1;
plot(x,asind(x))
grid on
```



Input Arguments

X — Sine of angle

scalar value | vector | matrix | N-D array

Sine of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `asind` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`

Complex Number Support: Yes

Output Arguments

Y — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

See Also

`asin` | `sin` | `sind`

Introduced before R2006a

asinh

Inverse hyperbolic sine

Syntax

```
Y = asinh(X)
```

Description

`Y = asinh(X)` returns the inverse hyperbolic sine for each element of `X`.

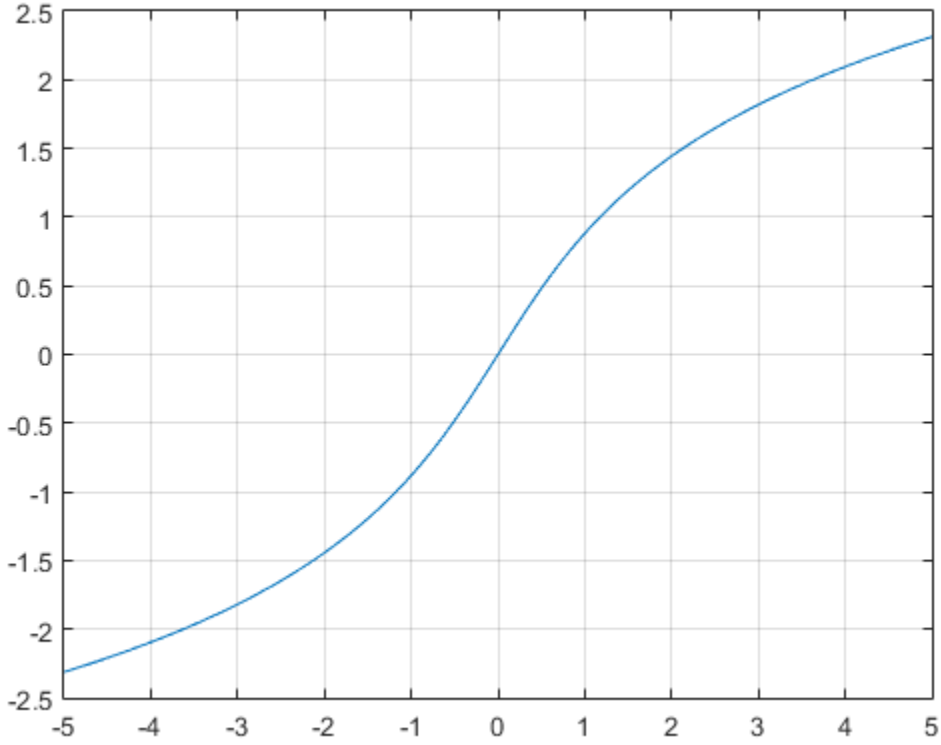
The `asinh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph of Inverse Hyperbolic Sine Function

Graph the inverse hyperbolic sine over the domain $-5 \le x \le 5$.

```
x = -5:.01:5;
plot(x,asinh(x)), grid on
```



More About

Inverse Hyperbolic Sine

The inverse hyperbolic sine can be defined as

$$\sinh^{-1}(z) = \log\left[z + (z^2 + 1)^{1/2}\right].$$

See Also

`asin` | `sinh` | `acosh`

Introduced before R2006a

assert

Throw error if condition false

Syntax

```
assert(cond)

assert(cond,msg)
assert(cond,msg,A1,...,An)

assert(cond,msgID,msg)
assert(cond,msgID,msg,A1,...,An)
```

Description

`assert(cond)` throws an error if `cond` is false.

`assert(cond,msg)` throws an error and displays the error message, `msg`, if `cond` is false.

`assert(cond,msg,A1,...,An)` displays an error message that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function, if `cond` is false. Each conversion character in `msg` is converted to one of the values `A1,...,An`.

`assert(cond,msgID,msg)` throws an error, displays the error message, msg, and includes an error identifier on the exception, if `cond` is false. The identifier enables you to distinguish errors and to control what happens when MATLAB encounters the errors.

`assert(cond,msgID,msg,A1,...,An)` includes an error identifier on the exception and displays a formatted error message.

Examples

Value in Expected Range

Assert that the value, `x`, is greater than a specified minimum value.

```
minVal = 7;
x = 26;

assert(minVal < x)
```

The expression evaluates as true, and the assertion passes.

Assert that the value of `x` is between the specified minimum and maximum values.

```
maxVal = 13;

assert((minVal < x) && (x < maxVal))
```

```
Assertion failed.
```

The expression evaluates as false. The assertion fails and MATLAB throws an error.

Expected Data Type

Assert that the product of two numbers is a double-precision number.

```
a = 13;
b = single(42);
c = a*b;

assert(isa(c,'double'),'Product is not type double.')
```

```
Product is not type double.
```

Enhance the error message to display the data type of `c`.

```
assert(isa(c,'double'),'Product is type %s, not double.',class(c))
```

```
Product is type single, not double.
```

Expected Code Conditions

Use the `assert` function to test for conditions that should not happen in normal code execution. If the coefficients are numeric, the computed roots should be numeric. A quadratic equation using the specified coefficients and computed roots should be zero.

```
function x = quadraticSolver(C)

validateattributes(C,{'numeric'},{'size',[1 3]})
```

```
a = C(1);
b = C(2);
c = C(3);

x(1) = (-b+sqrt(b^2-4*a*c))/(2*a);
x(2) = (-b-sqrt(b^2-4*a*c))/(2*a);
assert(isnumeric(x),'quadraticSolver:nonnumericRoots',...
    'Computed roots are not numeric')

y1 = a*x(1)^2+b*x(1)+c;
y2 = a*x(2)^2+b*x(2)+c;
assert(y1 == 0,'quadraticSolver:root1Error','Error in first root')
assert(isequal(y2,0),'quadraticSolver:root2Error','Error in second root')

end
```

- " Capture Information About Exceptions"

Input Arguments

`cond` — Condition to assert

MATLAB expression

Condition to assert, specified as a valid MATLAB expression. If `cond` is false, the `assert` function throws an error. `cond` can include relational operators (such as `<` or `==`) and logical operators (such as `&&`, `||`, or `~`). Use the logical operators `and` and `or` to create compound expressions. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Example: `a<0`

Example: `exist('myfunction.m','file')`

`msg` — Information about assertion failure

string

Information about the assertion failure, specified as a string. This message displays as the error message. To format the string, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see "Formatting Strings".

Note: You must specify more than one input argument with `assert` if you want MATLAB to convert special characters (such as `\t`, `\n`, `%s`, and `%d`) in the error message string.

Example: `'Assertion condition failed.'`

`A1,...,An` — Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array. This input argument provides the values that correspond to and replace the conversion specifiers in `msg`.

`msgID` — Identifier for assertion failure

string

Identifier for the assertion failure, specified as a string. Use the message identifier to help identify the source of the error or to control a selected subset of the errors in your program.

The message identifier includes a `component` and `mneumonic`. The identifier must always contain a colon and follows this simple format: `component:mnemonic`. The `component` and `mneumonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No whitespace characters can appear anywhere in msgID. For more information, see “Message Identifiers”.

Example: `'MATLAB:singularMatrix'`

Example: `'MATLAB:narginchk:notEnoughInputs'`

More About

Tips

- When you issue an error, MATLAB captures information about it and stores it in a data structure that is an object of the `MException` class. You can access information in the exception object by using `try/catch`. Or, if your program terminates because of an exception and returns control to the Command Prompt, you can use `MException.last`.

- If an assertion failure occurs within a `try` block, MATLAB does not cease execution of the program. In this case, MATLAB passes control to the `catch` block.

See Also

`MException` | `error`

Introduced in R2007a

assignin

Assign value to variable in specified workspace

Syntax

```
assignin(ws, 'var', val)
```

Description

`assignin(ws, 'var', val)` assigns the value `val` to the variable *`var`* in the workspace `ws`. The *`var`* input must be the array name only; it cannot contain array indices. If *`var`* does not exist in the specified workspace, `assignin` creates it. `ws` can have a value of `'base'` or `'caller'` to denote the MATLAB base workspace or the workspace of the caller function.

The `assignin` function is particularly useful for these tasks:

- Exporting data from a function to the MATLAB workspace
- Within a function, changing the value of a variable that is defined in the workspace of the caller function (such as a variable in the function argument list)

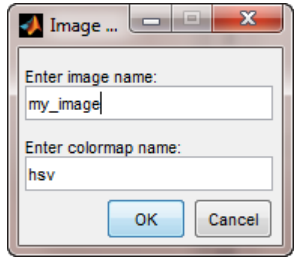
Examples

Example 1

This example creates a dialog box for the image display function, prompting a user for an image name and a colormap name. The `assignin` function is used to export the user-entered values to the MATLAB workspace variables `imfile` and `cmap`.

```
prompt = {'Enter image name:', 'Enter colormap name:'};
title = 'Image display - assignin example';
lines = 1;
def = {'my_image', 'hsv'};
answer = inputdlg(prompt, title, lines, def);
assignin('base', 'imfile', answer{1});
```

```
assignin('base', 'cmap', answer{2});
```



Example 2

`assignin` does not assign to specific elements of an array. The following statement generates an error:

```
X = 1:8;
assignin('base', 'X(3:5)', -1);
```

However, you can use the `evalin` function to do this:

```
evalin('base','X(3:5) = -1')
X =
     1     2    -1    -1    -1     6     7     8
```

More About

Tips

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the currently running function. Note that the base and caller workspaces are equivalent in the context of a function that is invoked from the MATLAB command line. For more information, see “Base and Function Workspaces”.

See Also

`evalin`

Introduced before R2006a

atan

Inverse tangent in radians

Syntax

```
Y = atan(X)
```

Description

`Y = atan(X)` returns the "Inverse Tangent" on page 1-480 ($\tan^{-1}$) of the elements of X. The `atan` function operates element-wise on arrays. For real elements of X, `atan(X)` returns values in the interval `[-pi/2,pi/2]`. For complex values of X, `atan(X)` returns complex values. All angles are in radians.

Examples

Inverse Tangent of a Value

```
atan(0.8)
```

```
ans =
    0.6747
```

Inverse Tangent of a Vector of Complex Values

Find the inverse tangent of the elements of vector `x`. The `atan` function acts on `x` element-wise.

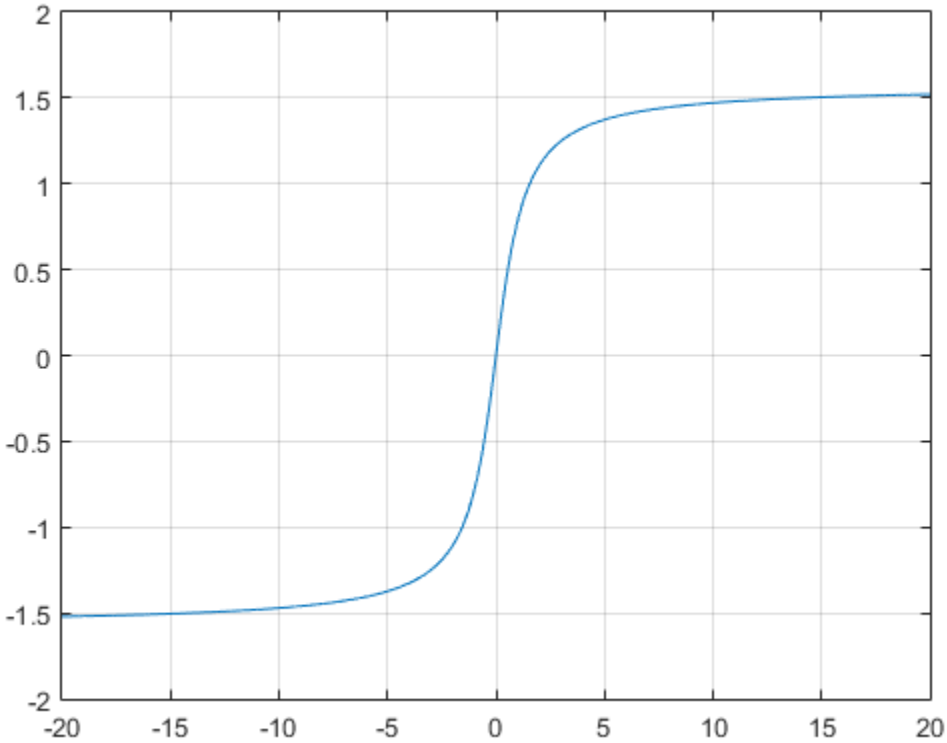
```
x = [0.5i 1+3i -2.2+i];
Y = atan(x)
```

```
Y =
   0.0000 + 0.5493i   1.4615 + 0.3059i  -1.2019 + 0.1506i
```

Plot the Inverse Tangent Function

Plot the inverse tangent function over the interval $-20 \leq x \leq 20$.

```
x = -20:0.01:20;
plot(x,atan(x))
grid on
```



Input Arguments

X — Numeric input

number | vector | matrix | multidimensional array

Numeric input, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Complex Number Support: Yes

More About

Inverse Tangent

The inverse tangent is defined as

$$\tan^{-1}(z) = \frac{i}{2}\log\left(\frac{i+z}{i-z}\right).$$

See Also

`atan2` | `atand` | `atanh` | `tan`

Introduced before R2006a

atan2

Four-quadrant inverse tangent

Syntax

```
P = atan2(Y,X)
```

Description

`P = atan2(Y,X)` returns the "Four-Quadrant Inverse Tangent" on page 1-484 ($\tan^{-1}$) of Y and X, which must be real. The `atan2` function acts on `Y` and `X` element-wise to return `P`, which is the same size as Y and X.

Examples

Find Four-Quadrant Inverse Tangent of a Point

Find the four-quadrant inverse tangent of the point `y = 4`, `x = -3`.

```
atan2(4,-3)
```

```
ans =
    2.2143
```

Convert Complex Number to Polar Coordinates

Convert `4 + 3i` into polar coordinates.

```
z = 4 + 3i;
r = abs(z)
theta = atan2(imag(z),real(z))
```

```
r =
     5
theta =
```

```
    0.6435
```

The radius `r` and the angle `theta` are the polar coordinate representation of `4 + 3i`.

Alternatively, use `angle` to calculate `theta`.

```
theta = angle(z)
```

```
theta =
    0.6435
```

Convert `r` and `theta` back into the original complex number.

```
z = r*exp(i*theta)
```

```
z =
   4.0000 + 3.0000i
```

Plot Four-Quadrant Inverse Tangent

Plot `atan2(Y,X)` for `-4<Y<4` and `-4<X<4`.

Define the interval to plot over.

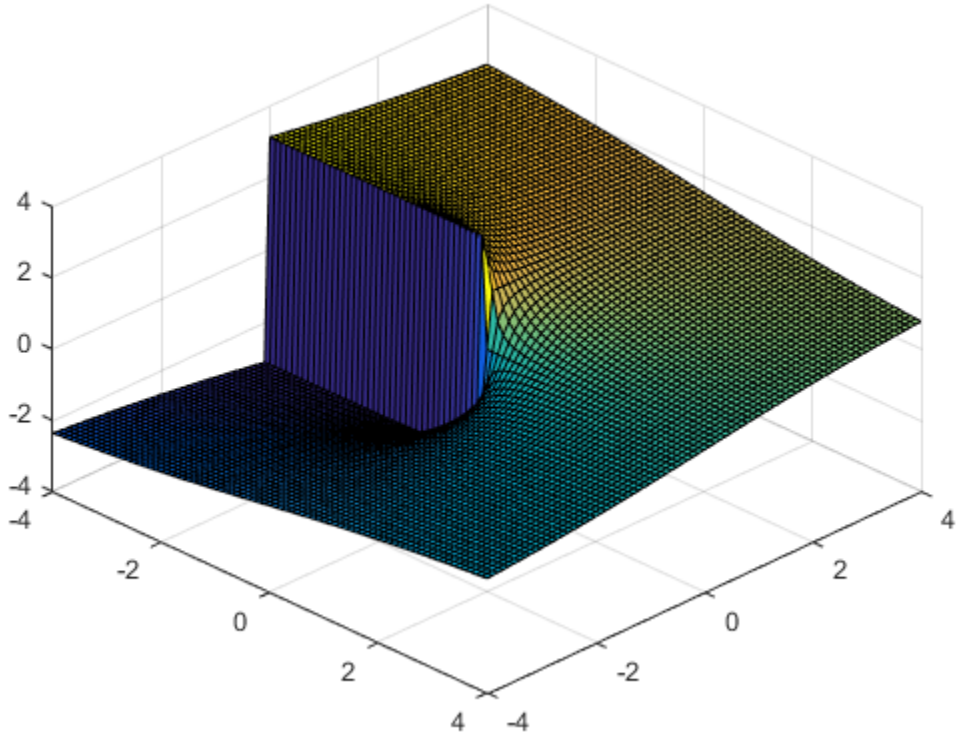
```
[X,Y] = meshgrid(-4:0.1:4,-4:0.1:4);
```

Find `atan2(Y,X)` over the interval.

```
P = atan2(Y,X);
```

Use `surf` to generate a surface plot of the function. Note that `plot` plots the discontinuity that exists at `Y=0` for all `X<0`.

```
surf(X,Y,P);
view(45,45);
```



Input Arguments

Y — Real valued input

number | vector | matrix | multidimensional array

Real valued input, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

X — Real valued input

number | vector | matrix | multidimensional array

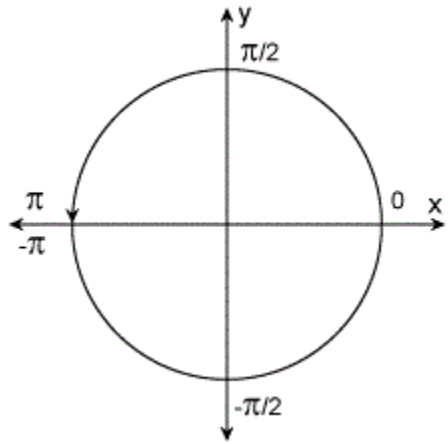
Real valued input, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

More About

Four-Quadrant Inverse Tangent

The four-quadrant tangent inverse, `atan2(Y,X)`, returns values in the closed interval `[-pi,pi]` based on the values of Y and X as shown in the graphic.



In contrast, `atan(Y,X)` returns results which are limited to the interval `(-pi/2,pi/2)`, which is the right side of this diagram.

See Also

`angle` | `atan` | `atan2d` | `atanh` | `tan`

Introduced before R2006a

atan2d

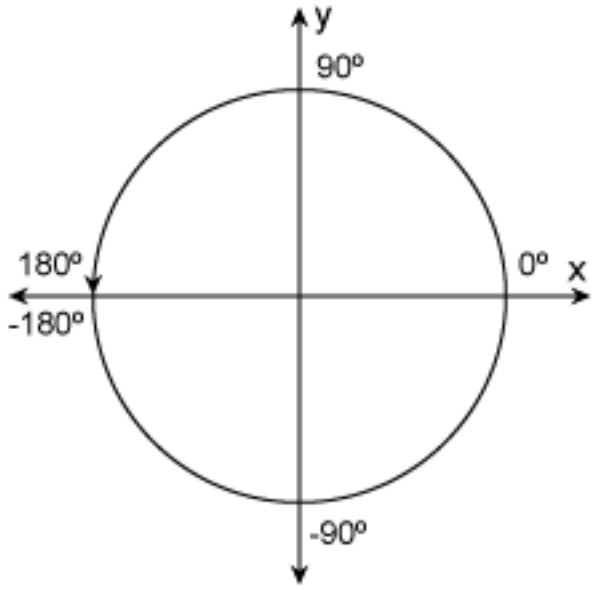
Four-quadrant inverse tangent in degrees

Syntax

```
D = atan2d(Y,X)
```

Description

`D = atan2d(Y,X)` returns the four-quadrant inverse tangent of points specified in the *x-y* plane. The result, `D`, is expressed in degrees.



Examples

Inverse Tangent of Four Points on the Unit Circle

```
x = [1 0 -1 0];
y = [0 1 0 -1];
d = atan2d(y,x)
```

```
d =

     0    90   180   -90
```

Input Arguments

Y — *y*-coordinates

scalar value | vector | matrix | N-D array

y-coordinates, specified as a real-valued scalar, vector, matrix, or N-D array.

Data Types: `single` | `double`

X — *x*-coordinates

scalar value | vector | matrix | N-D array

x-coordinates, specified as a real-valued scalar, vector, matrix, or N-D array.

Data Types: `single` | `double`

Output Arguments

D — Angles in degrees

scalar value | vector | matrix | N-D array

Angles in degrees, returned as a scalar, vector, matrix, or N-D array. These angles correspond to the points defined by `X` and `Y`, and they lie in the closed interval [–180,180].

More About

Tips

- Use `atand(Y/X)` for the inverse tangent with results on the interval [–90, 90].

See Also

atan | atan2 | atand | tan | tand

atand

Inverse tangent in degrees

Syntax

```
Y = atand(X)
```

Description

`Y = atand(X)` returns the inverse tangent ($\tan^{-1}$) of the elements of X in degrees. The function's domain and range include complex values. For real elements of X in the domain [-Inf, Inf], `atand` returns values in the range [-90, 90]. For complex values of X, `atand` returns complex values.

Examples

Inverse Tangent of Vector

```
x = [-50 -20 0 20 50];
y = atand(x)
```

```
y =

  -88.8542  -87.1376         0   87.1376   88.8542
```

The `atand` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Tangent of Complex Value

```
atand(10+i)
```

```
ans =
```

```
84.3450 + 0.5618i
```

Input Arguments

X — Tangent of angle

scalar value | vector | matrix | N-D array

Tangent of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `atand` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

See Also

`atan` | `atan2d` | `tan` | `tand`

Introduced before R2006a

atanh

Inverse hyperbolic tangent

Syntax

```
Y = atanh(X)
```

Description

The `atanh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

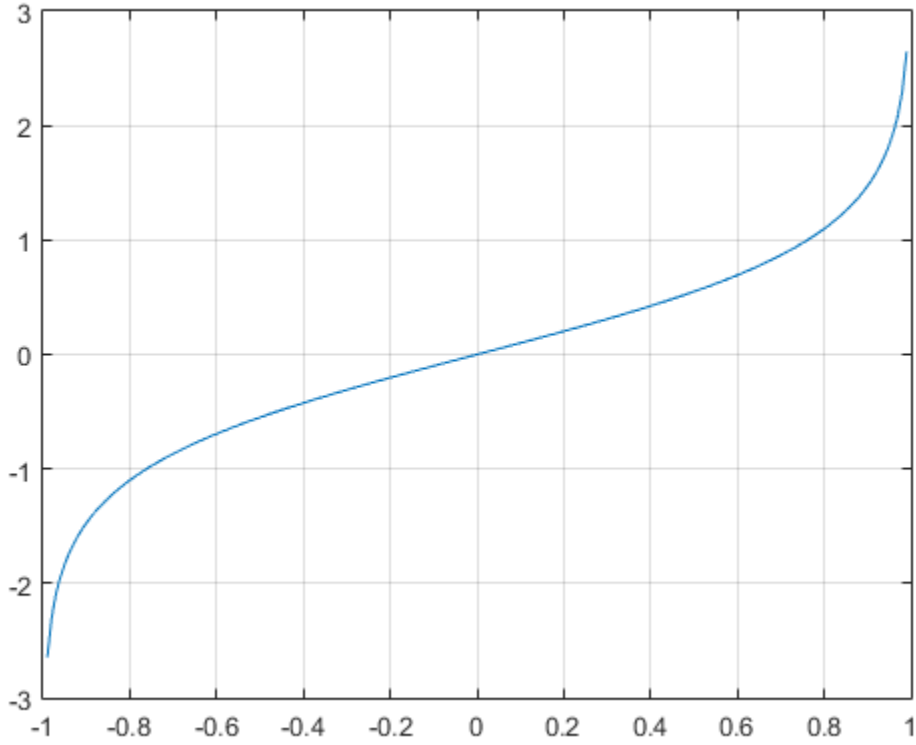
`Y = atanh(X)` returns the inverse hyperbolic tangent for each element of `X`.

Examples

Graph of Inverse Hyperbolic Tangent Function

Graph the inverse hyperbolic tangent over the domain $-1 < x < 1$.

```
x = -0.99:0.01:0.99;
plot(x,atanh(x)), grid on
```



More About

Inverse Hyperbolic Tangent

The inverse hyperbolic tangent can be defined as

$$\tanh^{-1}(z) = \frac{1}{2}\log\left(\frac{1+z}{1-z}\right).$$

See Also

`atan2` | `atan` | `tanh` | `asinh` | `acosh` | `tan`

Introduced before R2006a

audiodevinfo

Information about audio device

Syntax

```
info = audiodevinfo
nDevices = audiodevinfo(IO)
name = audiodevinfo(IO,ID)
DriverVersion = audiodevinfo(IO,ID,'DriverVersion')
support = audiodevinfo(IO,ID,Fs,nBits,nChannels)

ID = audiodevinfo(IO,name)
ID = audiodevinfo(IO,Fs,nBits,nChannels)
```

Description

`info = audiodevinfo` returns information about the input and output audio devices on the system.

`nDevices = audiodevinfo(IO)` returns the number of input devices on the system if `IO` is `1`, and returns the number of output devices on the system if `IO` is `0`.

`name = audiodevinfo(IO,ID)` returns the name of the audio device specified by the device identifier, `ID`.

`DriverVersion = audiodevinfo(IO,ID,'DriverVersion')` returns the name of the driver for the audio device specified by `ID`.

`support = audiodevinfo(IO,ID,Fs,nBits,nChannels)` returns `1` if the input or output audio device specified by `ID` supports the sample rate, number of bits, and number of channels specified by the values of `Fs`, `nBits`, and `nChannels`, respectively. Otherwise, `support` is `0`.

`ID = audiodevinfo(IO,name)` returns the device identifier of the input or output audio device identified by the device name, `name`. If no device is found with the specified name, then `audiodevinfo` returns an error.

`ID = audiodevinfo(IO,Fs,nBits,nChannels)` returns the device identifier of the first input or output device that supports the sample rate, number of bits, and the number of channels specified by the values of `Fs`, `nBits`, and `nChannels`, respectively. If no supporting device is found, then `ID` is `-1`.

Examples

View Information About Audio Devices

Call `audiodevinfo` with no inputs to view information about the input and output audio devices on a system.

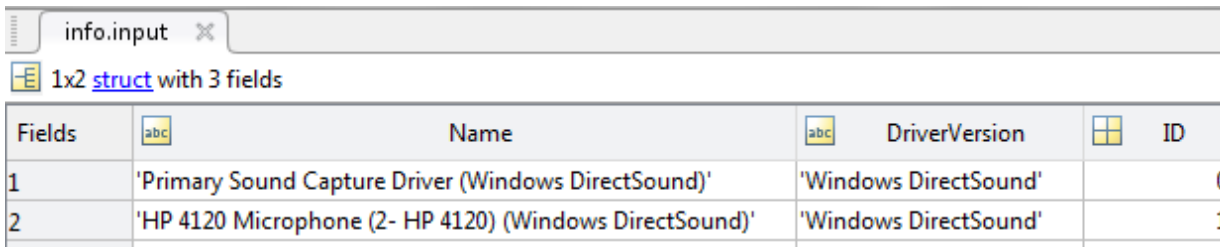
```
info = audiodevinfo
```

```
info =

     input: [1x2 struct]
    output: [1x3 struct]
```

`audiodevinfo` returns a structure containing two fields, `input` and `output`.

View the contents of the `input` field in the Variables editor.

```
openvar info.input
```



The Variables editor displays the input audio device names, driver used, and device identifiers. The values on your system might differ from this example.

View Number of Output Devices

View the number of output audio devices on the system, using an `IO` value of `0` to indicate output.

```
nDevices = audiodevinfo(0)
```

```
nDevices =

     3
```

This example shows three output devices, but your system might vary.

Check Support for Input Device

Check if the input audio device identified by the `ID` value, `0`, supports a sample rate of 44100 hertz, with 16 bits per sample, and two channels.

```
support = audiodevinfo(1,0,44100,16,2)
```

```
support =

     1
```

The input device supports the specified sample rate, number of bits and number of channels.

Input Arguments

`IO` — Input or output device

`1 | 0`

Input or output device, specified as `1` to indicate input, or `0` to indicate output.

`ID` — Audio device identifier

integer

Audio device identifier, specified as an integer. The device can be an input or output audio device.

`Fs` — Sample rate

scalar

Sample rate, in hertz, specified as a positive scalar.

Example: `44100`

Data Types: `single | double`

`nBits` — Number of bits per sample

scalar

Number of bits per sample, specified as a scalar.

Example: `16`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nChannels` — Number of audio channels

`1` | `2`

Number of audio channels, specified as `1` (mono) or `2` (stereo).

`name` — Name of input or output device

string

Name of the input or output audio device, specified as a string.

Data Types: `char`

Output Arguments

`info` — Information about audio devices

structure array

Information about audio devices, returned as a structure array containing two fields, `input` and `output`. Each field is an array of structures, with each structure containing information about one of the audio input or output devices on the system. The individual device structure fields are:

- `Name` — Name of the device, returned as a string.
- `DriverVersion` — Name of the driver used to communicate with the device, returned as a string.
- `ID` — Device identifier, returned as a scalar.

See Also

`audioplayer` | `audiorecorder`

audioplayer

Create object for playing audio

Syntax

```
player = audioplayer(Y,Fs)
player = audioplayer(Y,Fs,nBits)
player = audioplayer(Y,Fs,nBits,ID)
player = audioplayer(recorder)
player = audioplayer(recorder,ID)
```

Description

`player = audioplayer(Y,Fs)` creates an `audioplayer` object for signal `Y`, using sample rate `Fs`. The function returns a handle to the `audioplayer` object, `player`.

`player = audioplayer(Y,Fs,nBits)` uses `nBits` bits per sample for signal `Y`.

`player = audioplayer(Y,Fs,nBits,ID)` uses the audio device identified by `ID` for output.

`player = audioplayer(recorder)` creates an `audioplayer` object using audio recorder object `recorder`.

`player = audioplayer(recorder,ID)` creates an object from `recorder` that uses the audio device identified by`ID` for output.

Input Arguments

`Y`

Audio signal represented by a vector or two-dimensional array containing `single`, `double`, `int8`, `uint8`, or `int16` values.

The value range of the input sample depends on the data type. The following table lists these ranges.

Data Type	Sample Value Range
`int8`	-128 to 127
`uint8`	0 to 255
`int16`	-32768 to 32767
`single`	-1 to 1
`double`	-1 to 1

`Fs`

Sampling rate in Hz. Valid values depend on the specific audio hardware installed. Typical values supported by most sound cards are 8000, 11025, 22050, 44100, 48000, and 96000 Hz.

`nBits`

Bits per sample. Specify only when signal Y is represented by floating-point values. Valid values depend on the audio hardware installed: `8`, `16`, or `24`.

Default: `16`

`ID`

Device identifier. To obtain the ID of a device, use the `audiodevinfo` function.

Default: `-1` (default device)

`recorder`

Audio recorder object created by `audiorecorder`.

Methods

Note: When calling any method, include the `audioplayer` object name using function syntax, such as `stop(player)`.

`get`	Query properties of `audioplayer` object.

`isplaying`	Query whether playback is in progress: returns `true` or `false`.
`pause`	Pause playback.
`play`	Play audio from beginning to end.
`playblocking`	Play, and do not return control until playback completes.
`resume`	Restart playback from paused position.
`set`	Set properties of `audioplayer` object.
`stop`	Stop playback.

See the reference pages for `get`, `play`, `playblocking`, and `set` for additional syntax options.

Properties

`BitsPerSample`	Number of bits per sample. (Read-only)
`CurrentSample`	Current sample that the audio output device is playing. If the device is not playing, `CurrentSample` is the next sample to play with `play` or `resume`. (Read-only)
`DeviceID`	Identifier for audio device. (Read-only)
`NumberOfChannels`	Number of audio channels. (Read-only)
`Running`	Status of the audio player: `'on'` or `'off'`. (Read-only)
`SampleRate`	Sampling frequency in Hz.
`TotalSamples`	Total length of the audio data in samples. (Read-only)
`Tag`	String that labels the object.
`Type`	Name of the class: `'audioplayer'`. (Read-only)
`UserData`	Any type of additional data to store with the object.

The following four properties apply to callback functions. The first two inputs to your callback function must be the `audioplayer` object and an *`event`* structure.

`StartFcn`	Function to execute one time when playback starts.
`StopFcn`	Function to execute one time when playback stops.

`TimerFcn`	Function to execute repeatedly during playback. To specify time intervals for the repetitions, use the `TimerPeriod` property.
`TimerPeriod`	Time in seconds between `TimerFcn` callbacks. Default: `.05`

Examples

Load and play a sample audio file of Handel's "Hallelujah Chorus:"

```
load handel;
player = audioplayer(y, Fs);
play(player);
```

More About

- "Characteristics of Audio Files"
- "Play Audio"

See Also

`audiodevinfo` | `sound` | `audiorecorder`

Introduced before R2006a

audiorecorder

Create object for recording audio

Syntax

```
recorder = audiorecorder
recorder = audiorecorder(Fs,nBits,nChannels)
recorder = audiorecorder(Fs,nBits,nChannels,ID)
```

Description

`recorder = audiorecorder` creates an 8000 Hz, 8-bit, 1-channel `audiorecorder` object.

`recorder = audiorecorder(Fs,nBits,nChannels)` sets the sample rate `Fs` (in Hz), the sample size `nBits`, and the number of channels `nChannels`.

`recorder = audiorecorder(Fs,nBits,nChannels,ID)` sets the audio input device to the device specified by `ID`.

Input Arguments

`Fs`

Sampling rate in Hz. Valid values depend on the specific audio hardware installed. Typical values supported by most sound cards are 8000, 11025, 22050, 44100, 48000, and 96000 Hz.

Default: 8000

`nBits`

Bits per sample. Valid values depend on the audio hardware installed: `8`, `16`, or `24`.

Default: 8

`nChannels`

The number of channels: `1` (mono) or `2` (stereo).

Default: 1

`ID`

Device identifier. To obtain the ID of a device, use the `audiodevinfo` function.

Default: `-1` (default device)

Methods

Note: When calling any method, include the `audiorecorder` object name using function syntax, such as `stop(recorder)`.

`get`	Query properties of `audiorecorder` object.
`getaudiodata`	Create an array that stores the recorded signal values.
`getplayer`	Create an `audioplayer` object.
`isrecording`	Query whether recording is in progress: returns `true` or `false`.
`pause`	Pause recording.
`play`	Play recorded audio. This method returns an `audioplayer` object.
`record`	Start recording.
`recordblocking`	Record, and do not return control until recording completes. This method requires a second input for the length of the recording in seconds: `recordblocking(recorder,length)`
`resume`	Restart recording from paused position.
`set`	Set properties of `audiorecorder` object.
`stop`	Stop recording.

See the reference pages for `get`, `getaudiodata`, `play`, `record`, `recordblocking`, and `set` for additional syntax options.

Properties

`BitsPerSample`	Number of bits per sample. (Read-only)
`CurrentSample`	Current sample that the audio input device is recording. If the device is not recording, `CurrentSample` is the next sample to record with `record` or `resume`. (Read-only)
`DeviceID`	Identifier for audio device. (Read-only)
`NumberOfChannels`	Number of audio channels. (Read-only)
`Running`	Status of the audio recorder: `'on'` or `'off'`. (Read-only)
`SampleRate`	Sampling frequency in Hz. (Read-only)
`TotalSamples`	Total length of the audio data in samples. (Read-only)
`Tag`	String that labels the object.
`Type`	Name of the class: `'audiorecorder'`. (Read-only)
`UserData`	Any type of additional data to store with the object.

The following four properties apply to callback functions. The first two inputs to your callback function must be the `audiorecorder` object and an *`event`* structure.

`StartFcn`	Function to execute one time when recording starts.
`StopFcn`	Function to execute one time when recording stops.
`TimerFcn`	Function to execute repeatedly during recording. To specify time intervals for the repetitions, use the `TimerPeriod` property.
`TimerPeriod`	Time in seconds between `TimerFcn` callbacks. Default: `.05`

`audiorecorder` ignores any specified values for these properties, which will be removed in a future release.

`BufferLength`	Length of buffer in seconds.
`NumberOfBuffers`	Number of buffers.

Examples

Create an `audiorecorder` object for CD-quality audio in stereo, and view its properties:

```
recObj = audiorecorder(44100, 16, 2);
get(recObj)
```

Collect a sample of your speech with a microphone, and plot the signal data:

```
% Record your voice for 5 seconds.
recObj = audiorecorder;
disp('Start speaking.')
recordblocking(recObj, 5);
disp('End of Recording.');

% Play back the recording.
play(recObj);

% Store data in double-precision array.
myRecording = getaudiodata(recObj);

% Plot the waveform.
plot(myRecording);
```

More About

Tips

- To use an `audiorecorder` object, your system must have a properly installed and configured sound card.
- `audiorecorder` is not intended for long, high-sample-rate recording. `audiorecorder` uses system memory for storage and does not use disk buffering. When you attempt a large recording, your MATLAB performance sometimes degrades over time.

- "Characteristics of Audio Files"
- "Record Audio"
- "Record or Play Audio within a Function"

See Also

`audiodevinfo` | `sound` | `audioplayer`

Introduced before R2006a

aufinfo

Information about NeXT/SUN (`.au`) sound file

Note: `aufinfo` will be removed in a future release. Use `audioinfo` instead.

Syntax

```
[m d] = aufinfo(aufile)
```

Description

`[m d] = aufinfo(aufile)` returns information about the contents of the AU sound file specified by the string `aufile`.

`m` is the string `'Sound (AU) file'`, if `filename` is an AU file. Otherwise, it contains an empty string (`''`).

`d` is a string that reports the number of samples in the file and the number of channels of audio data. If `filename` is not an AU file, it contains the string `'Not an AU file'`.

See Also

`audioread` | `audioinfo`

Introduced before R2006a

auread

Read NeXT/SUN (`.au`) sound file

Note: `auread` will be removed in a future release. Use `audioread` instead.

Syntax

```
y = auread(aufile)
[y,Fs] = auread(aufile)
[y,Fs,nbits] = auread(aufile)
[ ___ ] = auread(aufile,N)
[ ___ ] = auread(aufile,[N1 N2])
siz = auread(aufile,'size')
```

Description

`y = auread(aufile)` loads a sound file specified by the string *`aufile`*, returning the sampled data in *`y`*. The `.au` extension is appended if no extension is given. Amplitude values are in the range `[-1,+1]`. `auread` supports multichannel data in the following formats:

- 8-bit mu-law
- 8-, 16-, and 32-bit linear
- Floating-point

`[y,Fs] = auread(aufile)` returns the sample rate (*`Fs`*) in Hertz used to encode the data in the file.

`[y,Fs,nbits] = auread(aufile)` returns the number of bits per sample (*`nbits`*).

`[ ___ ] = auread(aufile,N)` returns only the first *`N`* samples from each channel in the file.

`[ ___ ] = auread(aufile,[N1 N2])` returns only samples *`N1`* through *`N2`* from each channel in the file.

siz = auread(*aufile*,'size') returns the size of the audio data contained in the file in place of the actual audio data, returning the vector *siz* = [*samples channels*].

Examples

Create a sound file from the example file handel.mat, and read portions of the file back into MATLAB:

```
% Create .au file in current folder.
load handel.mat

hfile = 'handel.au';
auwrite(y, Fs, hfile)
clear y Fs

% Read the data back into MATLAB, and listen to audio.
[y, Fs, nbits] = auread(hfile);
sound(y, Fs);

% Pause before next read and playback operation.
duration = numel(y) / Fs;
pause(duration + 2)

% Read and play only the first 2 seconds.
nsamples = 2 * Fs;
[y2, Fs] = auread(hfile, nsamples);
sound(y2, Fs);
pause(4)

% Read and play the middle third of the file.
sizeinfo = auread(hfile, 'size');

tot_samples = sizeinfo(1);
startpos = tot_samples / 3;
endpos = 2 * startpos;

[y3, Fs] = auread(hfile, [startpos endpos]);
sound(y3, Fs);
```

See Also

audiowrite | audioinfo | audioplayer | audiorecorder | sound | audioread

Introduced before R2006a

auwrite

Write NeXT/SUN (`.au`) sound file

Note: `auwrite` will be removed in a future release.

Syntax

```
auwrite(y,aufile)
auwrite(y,Fs,aufile)
auwrite(y,Fs,N,aufile)
auwrite(y,Fs,N,method,aufile)
```

Description

`auwrite(y,aufile)` writes a sound file specified by the string *`aufile`*. The data should be arranged with one channel per column. Amplitude values outside the range `[-1,+1]` are clipped prior to writing. `auwrite` supports multichannel data for 8-bit mu-law and 8- and 16-bit linear formats.

`auwrite(y,Fs,aufile)` specifies the sample rate of the data in Hertz.

`auwrite(y,Fs,N,aufile)` selects the number of bits in the encoder. Allowable settings are *`N`* `= 8` and *`N`* `= 16`.

`auwrite(y,Fs,N,method,aufile)` allows selection of the encoding method, which can be either `'mu'` or `'linear'`. Note that mu-law files must be 8-bit. By default, *`method`* `= 'mu'`.

See Also

`audioread` | `audiowrite`

Introduced before R2006a

avifile

Create new Audio/Video Interleaved (AVI) file

Note: `avifile` has been removed. Use `VideoWriter` instead.

Syntax

```
aviobj = avifile(filename)
avifile(filename, ParameterName, ParameterValue)
```

Description

`aviobj = avifile(filename)` creates an `avifile` object, giving it the name specified in *`filename`*, using default values for all `avifile` object properties. If *`filename`* does not include an extension, `avifile` appends `.avi` to the file name. AVI is a file format for storing audio and video data.

`avifile` returns a handle to an AVI file object *`aviobj`*. Use this object to refer to the AVI file in other functions. An AVI file object supports properties and methods that control aspects of the AVI file created.

`aviobj` = `avifile(`*`filename`*`, `*`ParameterName`*`, `*`ParameterValue`*`)` accepts one or more comma-separated parameter name/value pairs. Set parameter values before any calls to `addframe`. The following table lists the available parameters and values.

Parameter Name	Value	Default
`'colormap'`	An *`m`*-by-3 matrix defining the colormap for indexed AVI movies, where *`m`* is no more than 256 (236 for `Indeo` compression). Valid only when the `'compression'` is `'MSVC'`, `'RLE'`, or `'None'`.	No default

Parameter Name	Value	Default
`'compression'`	A text string specifying the compression codec to use. To create an uncompressed file, specify a value of `'None'`. On UNIX® operating systems, the only valid value is `'None'`. On Windows systems, valid values include: • `'MSVC'` • `'RLE'` • `'Cinepak'` on 32-bit systems. • `'Indeo3'` or `'Indeo5'` on 32-bit Windows XP systems. Alternatively, specify a custom compression codec on Windows systems using the four-character code that identifies the codec (typically included in the codec documentation). If MATLAB cannot find the specified codec, it returns an error.	`'Indeo5'` on Windows systems. `'None'` on UNIX systems.
`'fps'`	A scalar value specifying the speed of the AVI movie in frames per second (fps).	15 fps
`'keyframe'`	For compressors that support temporal compression, the number of key frames per second.	2.1429 key frames per second
`'quality'`	A number from 0 through 100. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes. Valid only for compressed movies.	75
`'videoname'`	A descriptive name for the video stream, no more than 64 characters.	*`filename`*

Examples

Create the AVI file `example.avi`:

```
aviobj = avifile('example.avi','compression','None');

t = linspace(0,2.5*pi,40);
fact = 10*sin(t);
fig=figure;
[x,y,z] = peaks;
for k=1:length(fact)
    h = surf(x,y,fact(k)*z);
    axis([-3 3 -3 3 -80 80])
    axis off
    caxis([-90 90])

    F = getframe(fig);
    aviobj = addframe(aviobj,F);
end
close(fig);
aviobj = close(aviobj);
```

Alternatives

Use `VideoWriter` rather than `avifile` to create AVI files. `VideoWriter` supports files larger than 2 GB, and by default, creates files with Motion JPEG compression, which all platforms support.

More About

Tips

- On some Windows systems, including all 64-bit systems, the default Indeo® 5 codec is not available. MATLAB issues a warning, and creates an uncompressed file.
- On 32-bit Windows XP systems, MATLAB can create AVI files compressed with Indeo 3 and Indeo 5 codecs. However, Microsoft Windows XP Service Pack 3 (SP3) with Security Update 954157 disables playback of Indeo 3 and Indeo 5 codecs in Windows Media Player and Internet Explorer®. Consider specifying a `compression` value of `'None'`.
- `avifile` cannot write files larger than 2 GB.
- You can use dot notation to set `avifile` object properties. For example, set the `quality` property to 100:

```
aviobj = avifile('myavifile');
aviobj.quality = 100;
```

All property names of an `avifile` object are the same as the parameter names, except for the `keyframe` parameter, which corresponds to the `KeyFramePerSec` property. For example, change `keyframe` to `2.5`:

```
aviobj.KeyFramePerSec = 2.5;
```

See Also

`VideoWriter` | `addframe (avifile)` | `close (avifile)`

Introduced before R2006a

aviinfo

Information about Audio/Video Interleaved (AVI) file

Note: `aviinfo` will be removed in a future release. Use `VideoReader` and the `get` method instead.

Syntax

```
fileinfo = aviinfo(filename)
```

Description

`fileinfo = aviinfo(filename)` returns a structure whose fields contain information about the AVI file specified in the string `filename`. If `filename` does not include an extension, then `.avi` is used. The file must be in the current working directory or in a directory on the MATLAB path.

The set of fields in the `fileinfo` structure is shown below.

Field Name	Description
`AudioFormat`	String containing the name of the format used to store the audio data, if audio data is present
`AudioRate`	Integer indicating the sample rate in Hertz of the audio stream, if audio data is present
`Filename`	String specifying the name of the file
`FileModDate`	String containing the modification date of the file
`FileSize`	Integer indicating the size of the file in bytes
`FramesPerSecond`	Integer indicating the desired frames per second
`Height`	Integer indicating the height of the AVI movie in pixels

Field Name	Description
`ImageType`	String indicating the type of image. Either `'truecolor'` for a truecolor (RGB) image, or `'indexed'` for an indexed image.
`NumAudioChannels`	Integer indicating the number of channels in the audio stream, if audio data is present
`NumFrames`	Integer indicating the total number of frames in the movie
`NumColormapEntries`	Integer specifying the number of colormap entries. For a truecolor image, this value is 0 (zero).
`Quality`	Number between 0 and 100 indicating the video quality in the AVI file. Higher quality numbers indicate higher video quality; lower quality numbers indicate lower video quality. This value is not always set in AVI files and therefore can be inaccurate.
`VideoCompression`	String containing the compressor used to compress the AVI file. If the compressor is not Microsoft Video 1, Run Length Encoding (RLE), Cinepak, or Intel® Indeo, `aviinfo` returns the four-character code that identifies the compressor.
`Width`	Integer indicating the width of the AVI movie in pixels

See also

`mmfileinfo`, `VideoReader`, `VideoWriter`

Introduced before R2006a

aviread

Read Audio/Video Interleaved (AVI) file

Note `aviread` has been removed. Use `VideoReader` instead.

Syntax

```
mov = aviread(filename)
mov = aviread(filename, index)
```

Description

`mov = aviread(filename)` reads the AVI movie *`filename`* into the MATLAB movie structure *`mov`*. If *`filename`* does not include an extension, then `.avi` is used. Use the `movie` function to view the movie *`mov`*. On UNIX platforms, *`filename`* must be an uncompressed AVI file.

`mov` has two fields, `cdata` and `colormap`. The content of these fields varies depending on the type of image.

Image Type	cdata Field	colormap Field
`Truecolor`	Height-by-width-by-3 array of `uint8` values	Empty
`Indexed`	Height-by-width array of `uint8` values	`m`-by-3 array of `double` values

`aviread` supports 8-bit frames, for indexed and grayscale images, 16-bit grayscale images, or 24-bit truecolor images. Note, however, that `movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale image frames.

`mov = aviread(filename, index)` reads only the frames specified by *`index`*. *`index`* can be a single index or an array of indices into the video stream. In AVI files, the first frame has the index value 1, the second frame has the index value 2, and so on.

See also

`mmfileinfo`, `movie`, `VideoReader`, `VideoWriter`

Introduced before R2006a

axes

Create axes graphics object

Syntax

```
axes
axes('PropertyName',propertyvalue,...)
axes(h)
h = axes(...)
```

Properties

For a list of properties, see Axes Properties.

Description

`axes` creates an axes graphics object in the current figure using default property values. `axes` is the low-level function for creating axes graphics objects. MATLAB automatically creates an axes, if one does not already exist, when you issue a command that creates a graph.

`axes('PropertyName',propertyvalue,...)` creates an axes object having the specified property values. For a description of the properties, see Axes Properties. MATLAB uses default values for any properties that you do not explicitly define as arguments. The `axes` function accepts property name/property value pairs, structure arrays, and cell arrays as input arguments (see the `set` and `get` commands for examples of how to specify these data types). While the basic purpose of an axes object is to provide a coordinate system for plotted data, axes properties provide considerable control over the way MATLAB displays data.

`axes(h)` makes existing axes `h` the current axes and brings the figure containing it into focus. It also makes `h` the first axes listed in the figure's `Children` property and sets the figure's `CurrentAxes` property to `h`. The current axes is the target for functions that draw image, line, patch, rectangle, surface, and text graphics objects.

If you want to make an axes the current axes without changing the state of the parent figure, set the `CurrentAxes` property of the figure containing the axes:

```
set(figure_handle,'CurrentAxes',axes_handle)
```

This command is useful if you want a figure to remain minimized or stacked below other figures, but want to specify the current axes.

`h = axes(...)` returns the handle of the created axes object.

Use the `set` function to modify the properties of an existing axes or the `get` function to query the current values of axes properties. Use the `gca` command to obtain the handle of the current axes.

The `axis` (not `axes`) function provides simplified access to commonly used properties that control the scaling and appearance of axes.

Set default axes properties on the figure and root levels:

```
set(groot,'DefaultAxesPropertyName',PropertyValue,...)
set(gcf,'DefaultAxesPropertyName',PropertyValue,...)
```

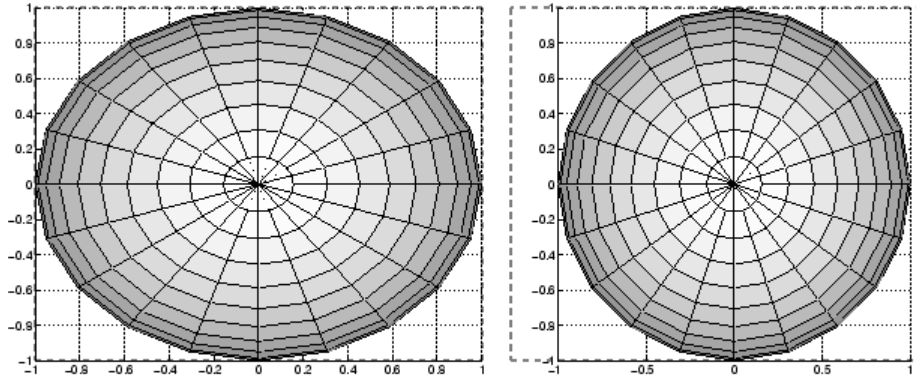
`PropertyName` is the name of the axes property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access axes properties.

Stretch-to-Fill

By default, MATLAB stretches the axes to fill the axes position rectangle (the rectangle defined by the last two elements in the `Position` property). This results in graphs that use the available space in the rectangle. However, some 3-D graphs (such as a sphere) appear distorted because of this stretching, and are better viewed with a specific three-dimensional aspect ratio.

Stretch-to-fill is active when the `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all auto (the default). However, stretch-to-fill is turned off when the `DataAspectRatio`, `PlotBoxAspectRatio`, or `CameraViewAngle` is user-specified, or when one or more of the corresponding modes is set to manual (which happens automatically when you set the corresponding property value).

This picture shows the same sphere displayed both with and without the stretch-to-fill. The dotted lines show the axes rectangle.



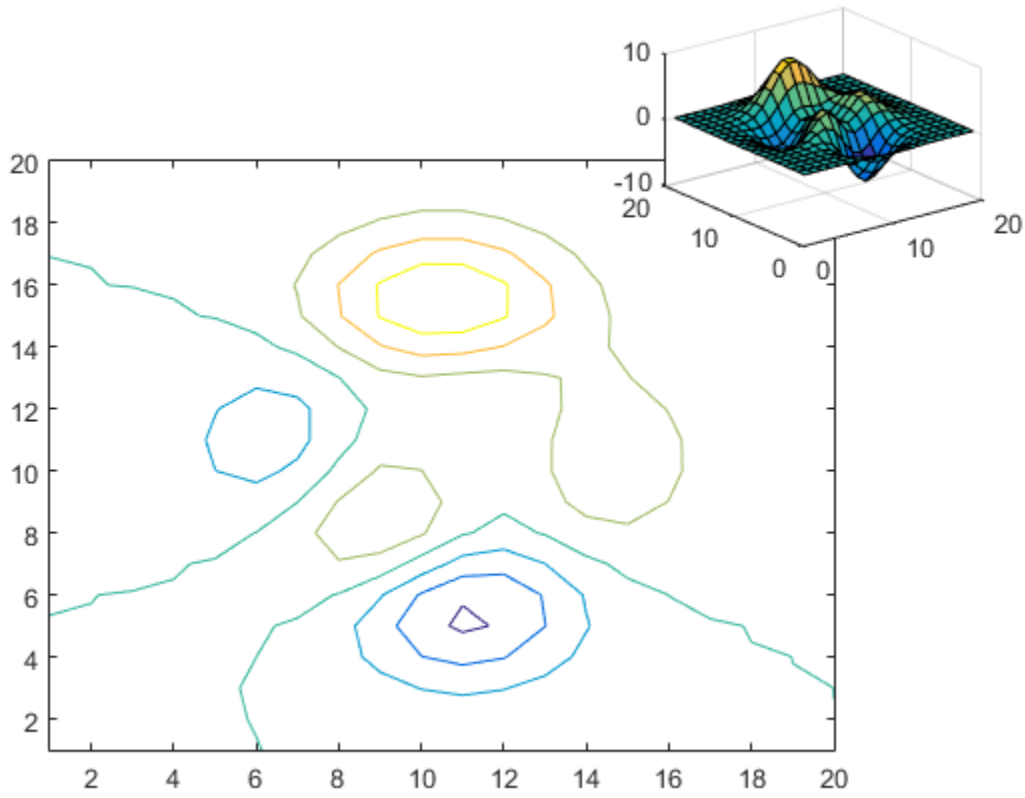
When stretch-to-fill is disabled, MATLAB sets the size of the axes to be as large as possible within the constraints imposed by the `Position` rectangle without introducing distortion. In the picture above, the height of the rectangle constrains the axes size.

Examples

Define Multiple Axes in Figure Window

Define multiple axes in a single figure window.

```
axes('Position',[0.1,0.1,0.7,0.7])
contour(peaks(20))

axes('Position',[0.65,0.7,0.28,0.28])
surf(peaks(20))
```



Alternatives

To create a figure select **New > Figure** from the figure window **File** menu. To add an axes to a figure, click one of the *New Subplots* icons in the Figure Palette, and slide right to select an arrangement of new axes. For details, see “Customize Graph Using Plot Tools”.

See Also

Functions

`axis` | `cla` | `clf` | `figure` | `gca` | `grid` | `subplot` | `title` | `view` | `xlabel` | `ylabel` | `zlabel`

Properties

Axes Properties

Introduced before R2006a

Axes Properties

Control axes appearance and behavior

Axes properties control the appearance and behavior of an axes object. By changing property values, you can modify certain aspects of the axes.

Starting in R2014b, you can use dot notation to query and set properties.

```
ax = gca;
c = ax.Color;
ax.Color = 'blue';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Color` — Color of axes back planes

`[1 1 1]` (default) | RGB triplet | color string | `'none'`

Color of axes back planes, specified as an RGB triplet, a color string, or `'none'`. If you set the color to `'none'`, then the axes is invisible and the figure color shows through.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`

Long Name	Short Name	RGB Triplet
'black	'k'	[0 0 0]

Example: [0 0 1]

Example: 'b'

Example: 'blue'

Box — Axes box outline

'off' (default) | 'on'

Axes box outline, specified as one of these values:

- 'off' — Do not display the box outline around the axes.
- 'on' — Display the box outline around the axes. In a 3-D view, the outline appears around the axes back planes. Use the BoxStyle property to change the extent of the outline.

The XColor, YColor, and ZColor properties control the color of the outline.

BoxStyle — Style of axes box outline

'back' (default) | 'full'

Style of axes box outline, specified as one of these values:

- 'back' — Outline the back planes of the 3-D box.
- 'full' — Outline the entire 3-D box.

The BoxStyle property affects only 3-D views.

XColor, YColor, ZColor — Color of axes outline and tick marks

[0.15 0.15 0.15] (default) | RGB triplet | color string | 'none'

Color of the axes outline in the *x*, *y*, or *z* direction and the associated tick marks, specified as an RGB triplet, a color string, or 'none'. If you set the color to 'none', then the axes outline is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

`[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Setting this property sets the associated mode property to manual.

Example: `[1 1 0]`

Example: `'y'`

Example: `'yellow'`

`XColorMode`, `YColorMode`, `ZColorMode` — Selection mode for axes outline color

`'auto'` (default) | `'manual'`

Selection mode for the axes outline color, specified as one of these values:

- `'auto'` — Use the default color. For related information about grid-line selection, see the `GridColorMode` property.
- `'manual'` — Use the manually specified color. To specify the color, set the XColor, YColor, or ZColor property.

`LineWidth` — Width of axes outline, tick marks, and grid lines

`0.5` (default) | scalar value

Width of axes outline, tick marks, and grid lines, specified as a scalar value in point units. One point equals 1/72 inch.

Example: `1.5`

Individual Axis Lines

`XAxisLocation` — Location of x-axis

`'bottom'` (default) | `'top'`

Location of the *x*-axis, specified as one of these values:

- `'bottom'` — Display *x*-axis at bottom of axes.
- `'top'` — Display *x*-axis at top of axes.

This property applies only to 2-D views.

`YAxisLocation` — Location of y-axis

`'left'` (default) | `'right'`

Location of *y*-axis, specified as one of these values:

- `'left'` — Display *y*-axis on right side of axes.
- `'right'` — Display *y*-axis on left side of axes.

This property applies only to 2-D views.

`XDir`, `YDir`, `ZDir` — Direction of increasing values along axis

`'normal'` (default) | `'reverse'`

Direction of increasing values along axis, specified as one of these values:

- `'normal'` — Normal direction of increasing values:
  - *x*-axis values increase from left to right.
  - *y*-axis values increase from bottom to top (2-D view) or front to back (3-D view).
  - *z*-axis values increase pointing out of the screen (2-D view) or from bottom to top (3-D view).
- `'reverse'` — Reverse direction of increasing values:
  - *x*-axis values decrease from left to right.
  - *y*-axis values decrease from bottom to top (2-D view) or front to back (3-D view).
  - *z*-axis values decrease pointing out of the screen (2-D view) or from bottom to top (3-D view).

XScale, YScale, ZScale — Scale of values along axis

'linear' (default) | 'log'

Scale of values along axis, specified as 'linear' or 'log'.

XLim, YLim, ZLim — Minimum and maximum axis limits

[0 1] (default) | two-element vector of the form [min max]

Minimum and maximum *x*-axis, *y*-axis, or *z*-axis limits, specified as a two-element vector of the form [min max].

If the associated mode property is set to 'auto', then MATLAB chooses the axis limits. If you assign a value to this property, then MATLAB sets the mode to 'manual' and does not automatically compute the limits.

XLimMode, YLimMode, ZLimMode — Selection mode for axis limits

'auto' (default) | 'manual'

Selection mode for the axis limits, specified as one of these values:

- 'auto' — Select axis limits based on the data plotted, that is, the total span of the XData, YData, or ZData of all the objects displayed in the axes.
- 'manual' — Use manually specified axis limits. To specify the axis limits, set the XLim, YLim, or ZLim property.

Tick Values and Labels

XTick, YTick, ZTick — Tick mark locations

[ ] (default) | vector of increasing values

Tick mark locations, specified as a vector of increasing values.

If the associated mode property is set to 'auto', then MATLAB chooses the tick values. If you assign a value to this property, then MATLAB sets the mode to 'manual' and does not automatically recompute the tick values.

Example: [2 4 6 8 10]

Example: 0:10:100

Data Types: single | double

XTickMode, YTickMode, ZTickMode — Selection mode for tick mark locations

'auto' (default) | 'manual'

Selection mode for the tick mark locations, specified as one of these values:

- 'auto' — Select the tick mark locations based on the range of data for the axis.
- 'manual' — Use manually specified tick mark locations. To specify the values, set the XTick, YTick, or ZTick property.

XTickLabel, YTickLabel, ZTickLabel — Tick mark labels

'' (default) | cell array of strings

Tick mark labels, specified as a cell array of strings. If you do not specify enough strings for all the ticks marks, then the axes cycles through the specified strings. Tick mark labels support TeX and LaTeX markup. See the TickLabelInterpreter property for more information.

If the associated mode property is set to 'auto', then MATLAB chooses the labels. If you assign a value to this property, then MATLAB sets the mode to 'manual' and does not automatically choose the labels.

XTickLabelMode, YTickLabelMode, ZTickLabelMode — Selection mode for tick mark labels

'auto' (default) | 'manual'

Selection mode for the tick mark labels, specified as one of these values:

- 'auto' — Label tick marks with numeric values that span the range of the plotted data.
- 'manual' — Use manually specified tick mark labels. To specify the labels, set the XTickLabel, YTickLabel, or ZTickLabel property.

TickLabelInterpreter — Interpretation of characters in tick labels

'tex' (default) | 'latex' | 'none'

Interpretation of tick label characters, specified as one of these values:

- 'tex' — Interpret strings using a subset of TeX markup.
- 'latex' — Interpret strings using LaTeX markup.
- 'none' — Display literal characters

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Modifiers remain in effect until the end of the string, except for superscripts and subscripts which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	Φ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	ß	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	v	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The FontName, FontWeight, and FontAngle properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at http://www.latex-project.org/.

`XTickLabelRotation`, `YTickLabelRotation`, `ZTickLabelRotation` — Rotation of tick labels

`0` (default) | scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees. Negative values give clockwise rotation.

Example: `90`

`XMinorTick`, `YMinorTick`, `ZMinorTick` — Display of minor tick marks

`'off'` (default) | `'on'`

Display of minor tick marks, specified as one of these values:

- `'off'` — Do not display minor tick marks.
- `'on'` — Display minor tick marks between the major tick marks on the axis. The space between the major tick marks determines the number of minor tick marks.

`TickLength` — Tick mark length

`[0.01 0.025]` (default) | two-element vector

Tick mark length, specified as a two-element vector of the form `[2Dlength 3Dlength]`. The first element is the tick mark length in 2-D views and the second element is the tick mark length in 3-D views. Specify the values in units normalized relative to the longest of the visible *x*-axis, *y*-axis, or *z*-axis lines.

Example: `[0.02 0.035]`

`TickDir` — Tick mark direction

`'in'` (default) | `'out'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct tick marks inward from the axis lines. This is the default for 2-D views.
- `'out'` — Direct tick marks outward from the axis lines. This is the default for 3-D views.

If the associated mode property is set to `'auto'`, then MATLAB chooses the tick label direction. If you set this property, then MATLAB sets the mode to `'manual'` and does not automatically choose the tick label direction.

`TickDirMode` — Selection mode for `TickDir`

`'auto'` (default) | `'manual'`

Selection mode for the TickDir property, specified as one of these values:

- `'auto'` — Use default tick direction.
- `'manual'` — Use manually specified tick mark direction. To specify the tick mark direction, set the TickDir property.

Grid Lines

`XGrid`, `YGrid`, `ZGrid` — Display of grid lines

`'off'` (default) | `'on'`

Display of grid lines, specified as one of these values:

- `'off'` — Do not display the grid lines.
- `'on'` — Display grid lines perpendicular to the axis, for example, along lines of constant *x*, *y*, or *z* values. Use the `grid` command to set all three properties `'on'` or `'off'`.

`XMinorGrid`, `YMinorGrid`, `ZMinorGrid` — Display of minor grid lines

`'off'` (default) | `'on'`

Display of minor grid lines, specified as one of these values:

- `'off'` — Do not display grid lines.
- `'on'` — Display grid lines aligned with the minor tick marks of the axis. You do not need to enable minor ticks to display minor grid lines.

`GridLineStyle` — Line style for grid lines

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style used for grid lines, specified as one of the strings shown in this table.

String	Line Style
`'-'`	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line
`'none'`	No line

To display the grid lines, use the `grid on` command or set the XGrid, YGrid, or ZGrid property to `'on'`.

`MinorGridLineStyle` — Line style for minor grid lines

`':'` (default) | `'-'` | `'--'` | `'-.'` | `'none'`

Line style used for minor grid lines, specified as one of the strings shown in this table.

String	Line Style
`'-'`	Solid line

String	Line Style
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dotted line
'none'	No line

To display minor grid lines, use the `grid minor` command or set the XGridMinor, YGridMinor, or ZGridMinor property to `'on'`.

`GridColor` — Color of grid lines

`[0.15 0.15 0.15]` (default) | RGB triplet | color string | `'none'`

Color of grid lines, specified as an RGB triplet, a color string, or `'none'`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

To set the colors for the axes box outline, use the XColor, YColor, and ZColor properties.

Setting this property sets the associated mode property to manual.

Example: `[0 0 1]`

Example: `'b'`

Example: `'blue'`

`GridColorMode` — Selection mode for `GridColor`

`'auto'` (default) | `'manual'`

Selection mode for the GridColor property, specified as `'auto'` or `'manual'`. The color is based on the values of the GridColorMode, `XColorMode`, `YColorMode`, and `ZColorMode` properties.

These tables list the grid line colors for different combinations of color modes.

GridColorMode Property	XColorMode Property	x-Axis Grid-Line Color
`'auto'`	`'auto'`	Use `GridColor` property.
`'auto'`	`'manual'`	Use `XGridColor` property.
`'manual'`	`'auto'`	Use `GridColor` property.
`'manual'`	`'manual'`	Use `GridColor` property.

GridColorMode Property	YColorMode Property	y-Axis Grid-Line Color
`'auto'`	`'auto'`	Use `GridColor` property.
`'auto'`	`'manual'`	Use `YGridColor` property.
`'manual'`	`'auto'`	Use `GridColor` property.
`'manual'`	`'manual'`	Use `GridColor` property.

GridColorMode Property	ZColorMode Property	z-Axis Grid-Line Color
`'auto'`	`'auto'`	Use `GridColor` property.
`'auto'`	`'manual'`	Use `ZGridColor` property.
`'manual'`	`'auto'`	Use `GridColor` property.
`'manual'`	`'manual'`	Use `GridColor` property.

`MinorGridColor` — Color of minor grid lines

`[0.1 0.1 0.1]` (default) | RGB triplet | color string

Color of minor grid lines, specified as an RGB triplet, a color string, or `'none'`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0 0 1]`

Example: `'b'`

Example: `'blue'`

`MinorGridColorMode` — Selection mode for `MinorGridColor`

`'auto'` (default) | `'manual'`

Selection mode for the MinorGridColor property, specified as `'auto'` or `'manual'`. The minor grid line color is based on the values of the MinorGridColorMode, `XColorMode`, `YColorMode`, and `ZColorMode` properties.

These tables list the minor grid line colors for different combinations of color modes.

MinorGridColorMode Property	XColorMode Property	x-Axis Minor Grid-Line Color
`'auto'`	`'auto'`	Use `MinorGridColor` property.
`'auto'`	`'manual'`	Use `XGridColor` property.

MinorGridColorMode Property	XColorMode Property	x-Axis Minor Grid-Line Color
`'manual'`	`'auto'`	Use `MinorGridColor` property.
`'manual'`	`'manual'`	Use `MinorGridColor` property.

MinorGridColorMode Property	YColorMode Property	y-Axis Minor Grid-Line Color
`'auto'`	`'auto'`	Use `MinorGridColor` property.
`'auto'`	`'manual'`	Use `YGridColor` property.
`'manual'`	`'auto'`	Use `MinorGridColor` property.
`'manual'`	`'manual'`	Use `MinorGridColor` property.

MinorGridColorMode Property	ZColorMode Property	z-Axis Minor Grid-Line Color
`'auto'`	`'auto'`	Use `MinorGridColor` property.
`'auto'`	`'manual'`	Use `ZGridColor` property.
`'manual'`	`'auto'`	Use `MinorGridColor` property.
`'manual'`	`'manual'`	Use `MinorGridColor` property.

`GridAlpha` — Grid-line transparency

`0.15` (default) | value in the range `[0,1]`

Grid-line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Setting this property sets the associated mode property to manual.

Example: `0.5`

`GridAlphaMode` — Selection mode for `GridAlpha`

`'auto'` (default) | `'manual'`

Selection mode for the GridAlpha property, specified as one of these values:

- `'auto'` — Use default transparency value of `0.15`.
- `'manual'` — Use the manually specified transparency value. To specify the value, set the GridAlpha property.

`MinorGridAlpha` — Minor grid-line transparency

`0.25` (default) | value in the range `[0,1]`

Minor grid-line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Setting this property sets the associated mode property to manual.

Example: `0.5`

`MinorGridAlphaMode` — Selection mode for `MinorGridAlpha`

`'auto'` (default) | `'manual'`

Selection mode for the `MinorGridAlpha` property, specified as one of these values:

- `'auto'` — Use the default transparency value of `0.15`.
- `'manual'` — Use the manually specified transparency value. To specify the value, set the MinorGridAlpha property.

`Layer` — Placement of grid lines and tick marks in relation to graphic objects

`'bottom'` (default) | `'top'`

Placement of grid lines and tick marks in relation to graphic objects, specified as one of these values:

- `'bottom'` — Display tick marks and grid lines under graphics objects.
- `'top'` — Display tick marks and grid lines over graphics objects.

This property affects only 2-D views.

Font Style

`FontName` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`FontSize` — Font size

`10` (default) | scalar numeric value

Font size, specified as a scalar numeric value. The FontSize, LabelFontSizeMultiplier, and TitleFontSizeMultiplier properties determine the size of the text used for the axis labels and the title. The FontUnits property determines the units used to interpret the font size.

Example: `12`

`TitleFontSizeMultiplier` — Scale factor for title font size

`1.1` (default) | numeric value greater than 0

Scale factor for title font size, specified as a numeric value greater than 0. The axes applies this scale factor to the value of the FontSize property to determine the font size for the title.

Example: `1.75`

`LabelFontSizeMultiplier` — Scale factor for label font size

`1.1` (default) | numeric value greater than 0

Scale factor for label font size, specified as a numeric value greater than 0. The axes applies this scale factor to the value of the FontSize property to determine the font size for the *x*-axis, *y*-axis, and *z*-axis labels.

Example: `1.5`

`FontUnits` — Font size units

`'points'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'normalized'`	Interpreted as a fraction of the axes height. If you resize the axes, MATLAB modifies the font size accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the axes height.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

If you set both the font size and the font units in one function call, you must set the FontUnits property first so that the axes correctly interprets the specified font size.

`FontAngle` — Character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Note: The `'oblique'` value has been removed. Use `'italic'` instead.

FontWeight — Thickness of text characters

'normal' (default) | 'bold'

Thickness of the text characters, specified as one of these values:

- 'normal' — Default weight as defined by the particular font
- 'bold' — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The 'light' and 'demi' font weight values have been removed. Use 'normal' instead.

TitleFontWeight — Thickness of title text

'bold' (default) | 'normal'

Thickness of the title text, specified as one of these values:

- 'bold' — Thicker characters outlines than normal.
- 'normal' — Default weight as defined by the particular font.

FontSmoothing — Text smoothing

'on' (default) | 'off'

Text smoothing, specified as one of these values:

- 'on' — Use antialiasing to make text appear smoother on the screen.
- 'off' — Do not use antialiasing. Use this setting if the text seems blurry. In certain cases, smoothed text blends against the background color and results in a blurring effect.

Title and Axis Labels

Title — Text object for axes title

text object (default)

Text object for axes title. Refer to this text object to change properties of the title.

```
ax = gca;
ax.Title.String = 'My Graph Title';
ax.Title.FontWeight = 'normal';
```

Note: To access the axes title text object, use the Title property or the `title` function. This text object is not contained in the axes `Children` property, it cannot be returned by `findobj`, and it does not use default values defined for text objects.

`XLabel`, `YLabel`, `ZLabel` — Text object for axis label

text object (default)

Text object for *x*-axis, *y*-axis, or *z*-axis label. Refer to this text object to change properties of the axis label.

```
ax = gca;
ax.YLabel.String = 'Y Axis';
ax.YLabel.FontSize = 12;
```

Note: To access the axis label text objects, use the XLabel, YLabel, and ZLabel properties or the `xlabel`, `ylabel`, and `zlabel` functions. These text objects are not contained in the axes `Children` property, they cannot be returned by `findobj`, and they do not use default values defined for text objects.

Line Colors and Line Styles

`ColorOrder` — Colors for multiline plots

seven predefined colors (default) | three-column matrix of RGB triplets

Colors for multiline plots, specified as a three-column matrix of RGB triplets. Each row of the matrix defines one color in the color order. The default color order has seven colors.

Default Color Order	Associated RGB Triplets

	`[ 0 0.4470 0.7410`
	`0.8500 0.3250 0.0980`
	`0.9290 0.6940 0.1250`
	`0.4940 0.1840 0.5560`

Default Color Order	Associated RGB Triplets
	0.4660 0.6740 0.1880 0.3010 0.7450 0.9330 0.6350 0.0780 0.1840]

The functions that create line plots cycle through the colors to color each line. The hold state for the axes affects the colors used.

- If the hold state is off (when the NextPlot property is set to `'replace'`), then high-level plotting functions such as `plot` reset the color order to the default colors before plotting. If you want to specify new colors for the color order and do not want high-level plotting functions to reset them, then set the NextPlot property to `'replacechildren'`. Alternatively, you can specify a new default value for the ColorOrder property on the root using the `set` function. For example, use `set(groot,'defaultAxesColorOrder',[0 1 0; 0 0 1])`.
- If the hold state for the axes is on (when the NextPlot property is set to `'add'`), then plotting functions cycle through the colors starting from the place in the color order where the last plot ended.

Data Types: `single` | `double`

`LineStyleOrder` — Line styles and markers for multiline plots

`'-'` solid line (default) | cell array of specifiers

Line styles and markers for multiline plots, specified as a cell array of one or more specifiers in this table, for example, `{'-*',':','o'}`.

Specifier	Line Style
`'-'`(default)	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line
`'+'`	Plus sign markers
`'o'`	Circle markers
`'*'`	Star markers
`'.'`	Point markers

Specifier	Line Style
'x'	Cross markers
's'	Square markers
'd'	Diamond markers
'^'	Upward-pointing triangle markers
'v'	Downward-pointing triangle markers
'>'	Right-pointing triangle markers
'<'	Left-pointing triangle markers
'p'	Five-pointed star (pentagram) markers
'h'	Six-pointed star (hexagram) markers

Combine strings to specify lines with markers, such as `'-*'`. The plot cycles through the line styles only after using all the colors contained in the `ColorOrder` property. If the hold state for the axes is off (when the NextPlot property of the axes is set to `'replace'`), then high-level plotting functions such as `plot` reset the line style order to the default line styles before plotting. If you want to specify a set of line styles that is different from the default and do not want high-level plotting functions to reset the `LineStyleOrder` property, then set NextPlot to `'replacechildren'`. Alternatively, you can specify a new default value for the LineStyleOrder property on the root using the `set` function. For example, use `set(groot,'defaultAxesLineStyleOrder',{'-*',':','o'})`.

Example: `{'-*',':','o'}`

`ColorOrderIndex` — Next color to use in color order

1 (default) | positive integer

Next color to use in the color order, specified as a positive integer. New plots added to the axes use colors based on the current value of the color order index. To use the first color for the next plot added to the axes, set the property to `1`.

After a `hold on` command, subsequently called plotting functions continue to cycle through the color-order, beginning with the current value of the color order index. Reset the color-order by setting the index to `1`.

If the specified index exceeds the number of colors defined by the current `ColorOrder` property, then plotting functions use the `mod` function to determine the index value as:

```
mod(ColorOrderIndex value, number of colors in ColorOrder)
```

Example: 5

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`LineStyleOrderIndex` — Next line style to use in line style order

1 (default) | positive integer

Next line style to use in line style order, specified as a positive integer. New plots added to the axes use line styles based on the current value of the line style order index. To use the first line style, set this property to `1`.

After a `hold on` command, subsequently called plotting functions continue to cycle through the line-style order, beginning with the current value of the line style order index.

If the specified index exceeds the number of line styles defined by the current `LineStyleOrder` property, plotting functions use the `mod` function to determine the index value as:

```
mod(LineStyleOrderIndex value,number of line styles  in LineStyleOrder)
```

Note: The default LineStyleOrder has only one line style.

Example: 1

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Color and Transparency Mapping

`CLim` — Color limits for objects using colormap

`[0 1]` (default) | two-element vector of the form `[cmin cmax]`

Color limits for objects in axes that use the colormap, specified as a two-element vector of the form `[cmin cmax]`. This property determines how data values map to the colors in the colormap where:

- `cmin` specifies the data value that maps to the first color in the colormap.
- `cmax` specifies the data value that maps to the last color in the colormap.

The axes linearly interpolates data values between `cmin` and `cmax` across the colormap. Values outside this range use either the first or last color, whichever is closest.

For information on changing the colormap, see the `colormap` function. For information on color mapping, see the `caxis` function.

If the associated mode property is set to `'auto'`, then MATLAB chooses the color limits. If you assign a value to this property, then MATLAB sets the mode to `'manual'` and does not automatically choose the color limits.

`CLimMode` — Selection mode for `CLim`

`'auto'` (default) | `'manual'`

Selection mode for the CLim property, specified as one of these values:

- `'auto'` — Set the CLim property to span the CData limits of the graphics objects displayed in the axes.
- `'manual'` — Use the manually specified values. To specify the values, set the CLim property. The axes does not change the values when the limits of the axes children change.

`ALim` — Alpha limits for images, patches, and surfaces with transparency

`[0 1]` (default) | two-element vector of the form `[amin amax]`

Alpha limits for images, patches, and surfaces with transparency, specified as a two-element vector of the form `[amin amax]`. This property determines how values in the AlphaData properties of image, patch, and surface objects map to the figure alpha map, where:

- `amin` specifies the data value mapped to the first alpha value in the figure alpha map.
- `amax` specifies the data value mapped to the last alpha value in the figure alpha map.

The axes linearly interpolates data values between `amin` and `amax` across the figure alpha map. Values outside this range use either the first or last alpha map value, whichever is closest.

The `Alphamap` property of the figure contains the alpha map. For more information, see the `alpha` function.

If the associated mode property is set to `'auto'`, then MATLAB chooses the alpha limits. If you set this property, then MATLAB sets the mode to `'manual'` and does not automatically choose the alpha limits.

`ALimMode` — Selection mode for `ALim`

`'auto'` (default) | `'manual'`

Selection mode for the ALim property, specified as one of these values:

- `'auto'` — Use values that span the AlphaData of the axes children contained in the Children property.
- `'manual'` — Use the manually specified values. To specify the values, set the ALim property.

`AmbientLightColor` — Background light color

`[1 1 1]` (default) | RGB triplet | color string

Background light color, specified as an RGB triplet or a color string. Ambient light is a directionless light that shines uniformly on all objects in the axes when there is a visible light object in the axes.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[1 0 1]`

Example: `'m'`

Example: `'magenta'`

Multiple Plots

`NextPlot` — Properties to reset when adding new plot

`'replace'` (default) | `'add'` | `'replacechildren'`

Properties to reset when adding a new plot to the axes, specified as one of these values:

- `'replace'` — Reset all axes properties, except Position and Units, to their default values and delete all axes children before displaying the new plot. Using this value is similar to using `cla reset`.
- `'add'` — Use the existing axes to add more plots to the axes and do not reset properties.
- `'replacechildren'` — Remove all child objects before adding new plots, but do not reset axes properties. This is similar to using `cla`.

The `newplot` function simplifies the use of the NextPlot property and is useful for writing custom graphing functions. Figures also have a `NextPlot` property.

`SortMethod` — Order for rendering objects

`'depth'` (default) | `'childorder'`

Order for rendering objects, specified as one of these values:

- `'depth'` — Draw objects in back-to-front order based on the current view. Use this value to ensure that objects in front of other objects are drawn correctly.
- `'childorder'` — Draw objects in the order in which they are created by graphics functions, without considering the relationship of the objects in three dimensions. This value can result in faster rendering, particularly if the figure is very large, but can also result in improper depth sorting of the objects displayed

Visibility

`Visible` — Visibility of axes

`'on'` (default) | `'off'`

Visibility of axes, specified as one of these values:

- `'on'` — Display the axes.
- `'off'` — Hide the axes without deleting it. You still can access the properties of an invisible axes object.

`Clipping` — Clipping of objects to axes limits

`'on'` (default) | `'off'`

Clipping of objects to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of plotted objects that are outside the axes limits.
- `'off'` — Display all plotted objects, even if parts of them appear outside the axes limits. Parts can appear outside the limits if you create a plot, set `hold on`, freeze the axis scaling, and then add a plot that is larger than the original plot.

The axes and individual objects in the axes have a `Clipping` property that controls the clipping behavior. If the `Clipping` property for the axes is `'on'`, then each individual object in the axes controls its own clipping behavior. To disable clipping for all objects in the axes, set the `Clipping` property for the axes to `'off'`. This table lists the results for different combinations of `Clipping` property values.

Clipping Property for Axes	Clipping Property for Individual Object	Result
`'on'`	`'on'`	Individual object is clipped (default)
`'on'`	`'off'`	Individual object is not clipped
`'off'`	`'on'`	No objects in axes are clipped
`'off'`	`'off'`	No objects in axes are clipped

`ClippingStyle` — Boundaries used for clipping

`'3dbox'` (default) | `'rectangle'`

Boundaries used for clipping, specified as one of these values:

- `'3dbox'` — Clip plotted objects to the six sides of the axes box defined by the axis limits.

- `'rectangle'` — Clip plotted objects to a rectangular boundary enclosing the axes in any given view.

The ClippingStyle property does not affect the display of objects if the `Clipping` property is set to `'off'` for either the object or the axes.

Location and Size

`Position` — Size and position of axes within figure or uipanel

`[0.1300 0.1100 0.7750 0.8150]` (default) | four-element vector

Size and position of the axes within the figure or uipanel that contains the axes, specified as a four-element vector of the form `[left bottom width height]`. The `left` and `bottom` elements define the distance from the lower-left corner of the container to the lower-left corner of the axes. The `width` and `height` elements are the axes dimensions.

By default, the values are measured in units normalized to the container. To change the units, set the `Units` property.

The axes dimensions are the largest possible values that conform to all other properties and do not extend outside the `Position` rectangle. Other axes properties that affect the axes size and shape include `DataAspectRatio`, `PlotBoxAspectRatio`, and `CameraViewAngle`.

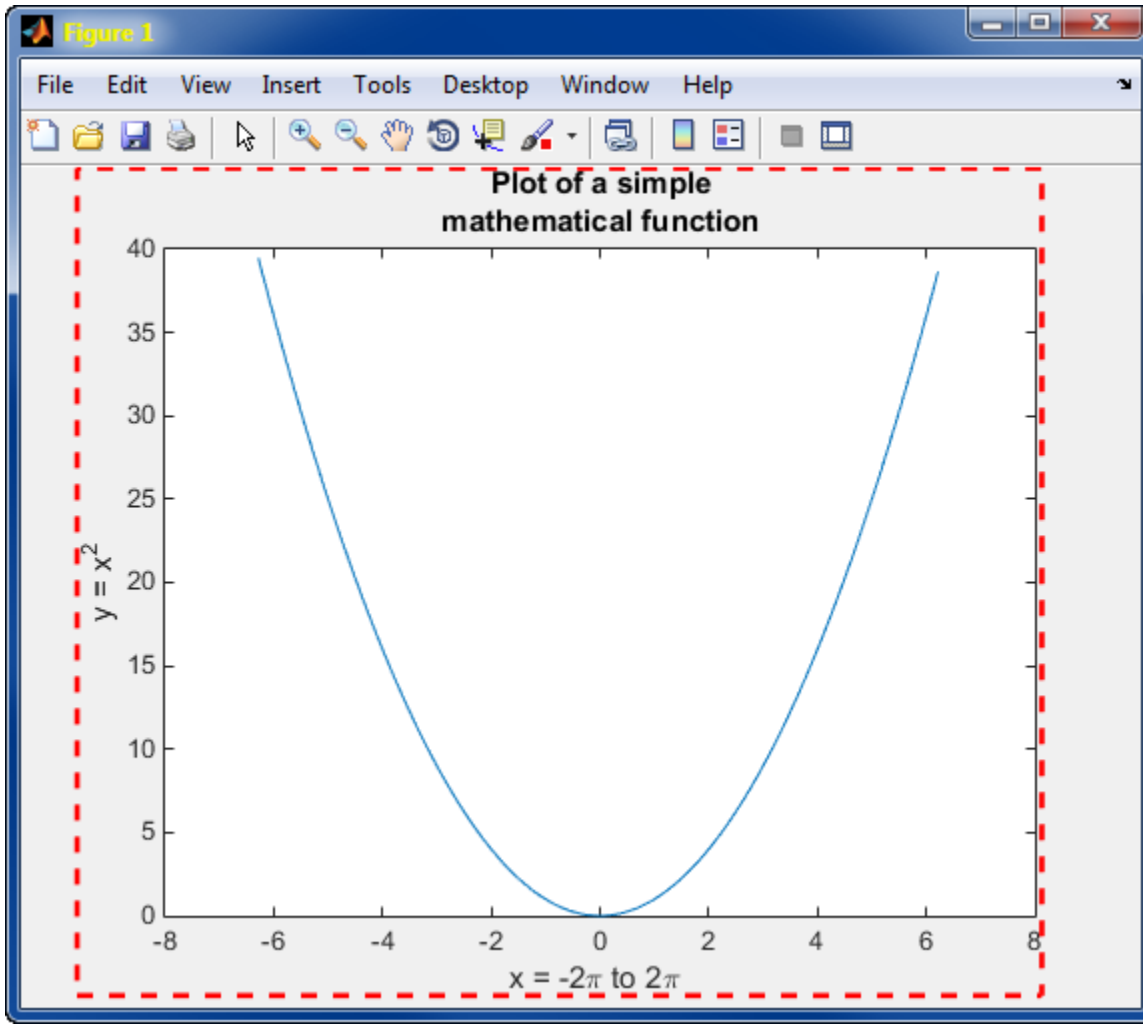
Example: `[0 0 1 1]`

`TightInset` — Margins for text labels

four-element vector of the form `[left bottom right top]`

Margins for the text labels, specified as a four-element vector of the form `[left bottom right top]`. The elements define the distances between the bounds of the Position property and the extent of the axes text labels and title. By default, the values are measured in units normalized to the figure or uipanel that contains the axes. To change the units, set the `Units` property.

The Position property and the TightInset property define the tightest bounding box that encloses the axes and its labels and title. The figure shows the region defined by combining the TightInset values and the Position values.



For more information, see “Axes Resize to Accommodate Titles and Labels”.

`OuterPosition` — Size and location of axes, including labels and margins

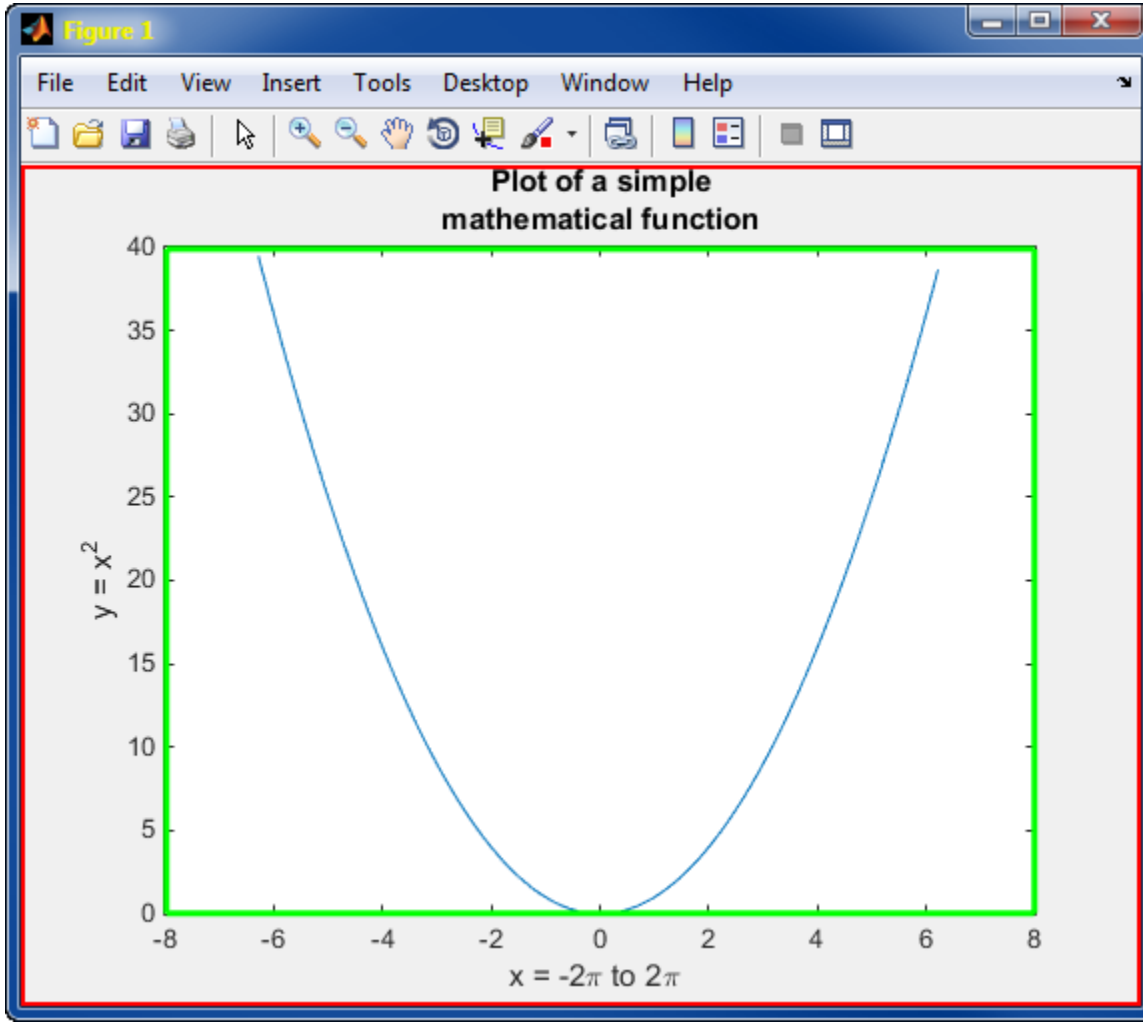
`[0 0 1 1]` (default) | four-element vector

Size and location of axes, including the labels and margins, specified as a four-element vector of the form `[left bottom width height]`. This vector defines the extents of

the rectangle that encloses the outer bounds of the axes. The `left` and `bottom` elements define the distance from the lower-left corner of the figure or uipanel that contains the axes to the lower-left corner of the rectangle. The `width` and `height` elements are the rectangle dimensions.

By default, the values are measured in units normalized to the container. To change the units, set the `Units` property. The default value of `[0 0 1 1]` includes the whole interior of the container.

The figure shows the region defined by the OuterPosition enclosed in the red rectangle. The green rectangle is the region defined by the Position property.



For more information, see “Axes Resize to Accommodate Titles and Labels”.

`ActivePositionProperty` — Position property to hold constant during resize operation
`'outerposition'` (default) | `'position'`

Position property to hold constant during resize operation, specified as one of these values:

- `'outerposition'` — Hold the `OuterPosition` property constant. Resizing the figure does not clip any of the text.
- `'position'` — Hold the `Position` property constant.

A figure can change size if you interactively change resize it or during a printing or exporting operation.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of the values in this table.

`Units`	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a uipanel. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

When specifying the units as a `Name,Value` pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `Position`.

Aspect Ratio

`Projection` — Type of projection onto 2-D screen

`'orthographic'` (default) | `'perspective'`

Type of projection onto 2-D screen, specified as one of these values:

- `'orthographic'` — Maintain the correct relative dimensions of graphics objects regarding the distance of a given point from the viewer, and draw lines that are parallel in the data on the screen.
- `'perspective'` — Incorporate foreshortening, which allows you to perceive depth in 2-D representations of 3-D objects. Perspective projection does not preserve the relative dimensions of objects. Instead, it displays a distant line segment smaller than a nearer line segment of the same length. Lines that are parallel in the data might not appear parallel on screen.

`DataAspectRatio` — Relative length of data units along each axis

`[1 1 1]` (default) | three-element vector of the form `[dx dy dz]`

Relative length of data units along each axis, specified as a three-element vector of the form `[dx dy dz]`. This vector defines the relative *x*, *y*, and *z* data scale factors. For example, specifying this property as `[1 2 1]` sets the length of one unit of data in the *x*-direction to be the same length as two units of data in the *y*-direction and one unit of data in the *z*-direction.

The DataAspectRatio property interacts with the `PlotBoxAspectRatio`, `XLimMode`, `YLimMode`, and `ZLimMode` properties to control how MATLAB scales the *x*-axis, *y*-axis, and *z*-axis.

Setting a value for the DataAspectRatio disables the "stretch-to-fill figure shape" behavior if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all set to `'auto'`.

If the associated mode property is set to `'auto'`, then MATLAB chooses the ratio values. If you set this property, then MATLAB sets the mode to `'manual'`.

Example: `[1 1 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DataAspectRatioMode` — Selection mode for `DataAspectRatio`

`'auto'` (default) | `'manual'`

Selection mode for the DataAspectRatio property, specified as one of these values:

- `'auto'` — Use values that make best use of the area provided by the figure.
- `'manual'` — Use the manually specified values. To specify the values, set the DataAspectRatio property. Changing DataAspectRatioMode to `'manual'` disables the "stretch-to-fill figure shape" behavior if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all `'auto'`.

  This table describes the behavior for various combinations of properties when you disable the stretch-to-fill behavior.

XLimitMode, YLimitMode, and ZLimitMode	DataAspectRatioMode	PlotBoxAspectRatioMode	Behavior
`'auto'`	`'auto'`	`'auto'`	The axes chooses limits that span the data range in all dimensions.
`'auto'`	`'auto'`	`'manual'`	The axes chooses limits that span the data range in all dimensions and modifies DataAspectRatio to achieve the specified PlotBoxAspectRatio within the limits.
`'auto'`	`'manual'`	`'auto'`	The axes chooses limits that span the data range in all dimensions and modifies PlotBoxAspectRatio to achieve the specified DataAspectRatio within the limits.
`'auto'`	`'manual'`	`'manual'`	The axes chooses limits that completely fit and center the plot within the specified PlotBoxAspectRatio given the specified

XLimitMode, YLimitMode, and ZLimitMode	DataAspectRatioMode	PlotBoxAspectRatioMode	Behavior
			DataAspectRatio (this might produce empty space around 2 of the 3 dimensions).
`'manual'`	`'auto'`	`'auto'`	The axes honors the specified limits and modifies the DataAspectRatio and PlotBoxAspectRatio as necessary.
`'manual'`	`'auto'`	`'manual`	The axes honors the specified limits and PlotBoxAspectRatio values and modifies the DataAspectRatio values as necessary.
`'manual'`	`'manual'`	`'auto'`	The axes honors the specified limits and DataAspectRatio values and modifies the PlotBoxAspectRatio values as necessary.
One set to `'manual'` and two set to `'auto'`	`'manual'`	`'manual'`	The axes selects the remaining limits to honor the specified aspect ratios and limit.
Two set to `'manual'` and one set to `'auto'`, or all three set to `'manual'`	`'manual'`	`'manual'`	The axes ignores the PlotBoxAspectRatio value and uses the specified limits and DataAspectRatio

`PlotBoxAspectRatio` — Relative length of each axis

`[1 1 1]` (default) | three-element vector of the form `[px py pz]`

Relative length of each axis, specified as a three-element vector of the form `[px py pz]` defining the relative *x*-axis, *y*-axis, and *z*-axis scale factors. The plot box is a box enclosing the axes data region as defined by the axis limits.

The PlotBoxAspectRatio property interacts with the DataAspectRatio, XLimMode, YLimMode, and ZLimMode properties. Setting the PlotBoxAspectRatio disables the "stretch-to-fill figure shape" behavior, if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all `'auto'`.

If the associated mode property is set to `'auto'`, then MATLAB chooses the ratio values. If you set this property, then MATLAB sets the mode to `'manual'`.

Example: `[1,0.75,0.75]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PlotBoxAspectRatioMode` — Selection mode for `PlotBoxAspectRatio`

`'auto'` (default) | `'manual'`

Selection mode for the PlotBoxAspectRatio property, specified as one of these values:

- `'auto'` — Use values that make best use of the area provided by the figure.
- `'manual'` — Use the manually specified values. To specify the values, set the PlotBoxAspectRatio property. Changing PlotBoxAspectRatioMode to `'manual'` disables the "stretch-to-fill figure shape" behavior if PlotBoxAspectRatioMode, DataAspectRatioMode, and CameraViewAngleMode are all `'auto'`.

This table describes the behavior for various combinations of properties when you disable the stretch-to-fill behavior.

XLimitMode, YLimitMode, and ZLimitMode	DataAspectRatioMode	PlotBoxAspectRatioMode	Behavior
`'auto'`	`'auto'`	`'auto'`	The axes chooses limits that span the data range in all dimensions.
`'auto'`	`'auto'`	`'manual'`	The axes chooses limits that span the data range in all dimensions and modifies DataAspectRatio to

XLimitMode, YLimitMode, and ZLimitMode	DataAspectRatioMode	PlotBoxAspectRatioMode	Behavior
			achieve the specified PlotBoxAspectRatio within the limits.
'auto'	'manual'	'auto'	The axes chooses limits that span the data range in all dimensions and modifies PlotBoxAspectRatio to achieve the specified DataAspectRatio within the limits.
'auto'	'manual'	'manual'	The axes chooses limits that completely fit and center the plot within the specified PlotBoxAspectRatio given the specified DataAspectRatio (this might produce empty space around 2 of the 3 dimensions).
'manual'	'auto'	'auto'	The axes honors the specified limits and modifies the DataAspectRatio and PlotBoxAspectRatio as necessary.
'manual'	'auto'	'manual	The axes honors the specified limits and PlotBoxAspectRatio values and modifies the DataAspectRatio values as necessary.

XLimitMode, YLimitMode, and ZLimitMode	DataAspectRatioMode	PlotBoxAspectRatioMode	Behavior
`'manual'`	`'manual'`	`'auto'`	The axes honors the specified limits and DataAspectRatio values and modifies the PlotBoxAspectRatio values as necessary.
One set to `'manual'` and two set to `'auto'`	`'manual'`	`'manual'`	The axes selects the remaining limits to honor the specified aspect ratios and limit.
Two set to `'manual'` and one set to `'auto'`, or all three set to `'manual'`	`'manual'`	`'manual'`	The axes ignores the PlotBoxAspectRatio value and uses the specified limits and DataAspectRatio

View

`CameraPosition` — Location of camera

three-element vector of the form `[x y z]`

Location of camera, or the viewpoint, specified as a three-element vector of the form `[x y z]`. This vector defines the axes coordinates of the location. Changing the CameraPosition property changes the point from which you view the axes.

The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target. For an illustration, see “Camera Graphics Terminology”.

If the `Projection` is `'perspective'`, then as you change the CameraPosition, the amount of perspective also changes.

If the associated mode property is set to `'auto'`, then MATLAB chooses the camera location. If you set this property, then MATLAB sets the mode to `'manual'`.

Data Types: `single | double`

CameraPositionMode — Selection mode for CameraPosition

'auto' (default) | 'manual'

Selection mode for the CameraPosition property, specified as one of these values:

- 'auto' — Calculate the CameraPosition value such that the camera lies a fixed distance from the target along the azimuth and elevation specified by the current view, as returned by the view function. Functions like rotate3d, zoom, and pan, change this mode to 'auto' to perform their actions.
- 'manual' — Use the manually specified value. To specify the value, set the CameraPosition property.

This table summarizes the camera behavior for various combinations of CameraViewAngleMode, CameraTargetMode, and CameraPositionMode property values.

CameraViewAngleMode	CameraTargetMode	CameraPositionMode	Behavior
'auto'	'auto'	'auto'	Sets CameraTarget to plot box centroid, sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.
'auto'	'auto'	'manual'	Sets CameraTarget to plot box centroid, sets CameraViewAngle to capture entire scene.
'auto'	'manual'	'auto'	Sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.
'auto'	'manual'	'manual'	Sets CameraViewAngle to capture entire scene.
'manual'	'auto'	'auto'	Sets CameraTarget to plot box centroid, sets CameraPosition along the view axis.
'manual'	'auto'	'manual'	Sets CameraTarget to plot box centroid.

CameraViewAngleMode	CameraTargetMode	CameraPositionMode	Behavior
'manual'	'manual'	'auto'	Sets CameraPosition along the view axis.
'manual'	'manual'	'manual'	Uses specified camera values

CameraTarget — Point used as camera target

three-element vector of the form `[x y z]`

Point used as camera target, specified as a three-element vector of the form `[x y z]`. This vector defines the axes coordinates of the point. The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target. For an illustration, see "Camera Graphics Terminology".

If the associated mode property is set to `'auto'`, then MATLAB chooses the camera target point. If you set this property, then MATLAB sets the mode to `'manual'`.

Data Types: `single | double`

CameraTargetMode — Selection mode for CameraTarget

`'auto'` (default) | `'manual'`

Selection mode for the CameraTarget property, specified as one of these values:

- `'auto'` — Position the camera target at the centroid of the axes plot box.
- `'manual'` — Use the manually specified camera target value. To specify a value, set the `CameraTarget` property.

This table summarizes the camera behavior for various combinations of CameraViewAngleMode, CameraTargetMode, and CameraPositionMode property values.

CameraViewAngleMode	CameraTargetMode	CameraPositionMode	Behavior
'auto'	'auto'	'auto'	Sets CameraTarget to plot box centroid, sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.
'auto'	'auto'	'manual'	Sets CameraTarget to plot box centroid, sets

CameraViewAngleMode	CameraTargetMode	CameraPositionMode	Behavior
			CameraViewAngle to capture entire scene.
'auto'	'manual'	'auto'	Sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.
'auto'	'manual'	'manual'	Sets CameraViewAngle to capture entire scene.
'manual'	'auto'	'auto'	Sets CameraTarget to plot box centroid, sets CameraPosition along the view axis.
'manual'	'auto'	'manual'	Sets CameraTarget to plot box centroid.
'manual'	'manual'	'auto'	Sets `CameraPosition` along the view axis.
'manual'	'manual'	'manual'	Uses specified camera values

`CameraUpVector` — Vector defining upwards direction

`[0 1 0]` (default for 2-D view) | `[0 0 1]` (default for 3-D view) | three-element direction vector of the form `[x y z]`

Vector defining upwards direction, specified as a three-element direction vector of the form `[x y z]`. For an illustration, see "Camera Graphics Terminology".

If the associated mode property is set to `'auto'`, then MATLAB chooses the vector values. If you set this property, then MATLAB sets the mode to `'manual'`.

Example: `[sin(45) cos(45) 1]`

`CameraUpVectorMode` — Selection mode for `CameraUpVector`

`'auto'` (default) | `'manual'`

Selection mode for the CameraUpVector property, specified as one of these values:

- `'auto'` — Use `[0 0 1]` for 3-D views so that the positive *z*-direction is up. Use `[0 1 0]` for 2-D views so that the positive *y*-direction is up.

- `'manual'` — Use the manually specified vector defining the upwards direction. To specify a value, set the `CameraUpVector` property.

`CameraViewAngle` — Field of view

`6.6086` (default) | scalar angle in range [0,180)

Field of view, specified as a scalar angle greater than 0 and less than or equal to 180. Changing the camera view angle affects the size of graphics objects displayed in the axes, but does not affect the degree of perspective distortion. The greater the angle, the larger the field of view and the smaller objects appear in the scene. For an illustration, see "Camera Graphics Terminology".

If the associated mode property is set to `'auto'`, then MATLAB chooses the field of view value. If you set this property, then MATLAB sets the mode to `'manual'`.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`CameraViewAngleMode` — Selection mode for `CameraViewAngle`

`'auto'` (default) | `'manual'`

Selection mode for the CameraViewAngle property, specified as one of these values:

- `'auto'` — Select the field of view as the minimum angle that captures the entire scene, up to 180 degrees.
- `'manual'` — Use the manually specified field of view. To specify a value, set the `CameraViewAngle` property.

  This table summarizes the camera behavior for various combinations of CameraViewAngleMode, CameraTargetMode, and CameraPositionMode property values.

`CameraViewAngleMode`	`CameraTargetMode`	`CameraPositionMode`	Behavior
`'auto'`	`'auto'`	`'auto'`	Sets CameraTarget to plot box centroid, sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.

CameraViewAngleMode	CameraTargetMode	CameraPositionMode	Behavior
'auto'	'auto'	'manual'	Sets CameraTarget to plot box centroid, sets CameraViewAngle to capture entire scene.
'auto'	'manual'	'auto'	Sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.
'auto'	'manual'	'manual'	Sets CameraViewAngle to capture entire scene.
'manual'	'auto'	'auto'	Sets CameraTarget to plot box centroid, sets CameraPosition along the view axis.
'manual'	'auto'	'manual'	Sets CameraTarget to plot box centroid.
'manual'	'manual'	'auto'	Sets `CameraPosition` along the view axis.
'manual'	'manual'	'manual'	Uses specified camera values

`View` — (obsolete) Azimuth and elevation of view

`[0 90]` (default) | two-element vector of the form `[azimuth elevation]`

Azimuth and elevation of view, specified as a two-element vector of the form `[azimuth elevation]` defined in degree units.

This property is obsolete. Use the CameraPosition, CameraTarget, CameraUpVector, and CameraViewAngle properties instead.

Example: `[45 45]`

Identifiers

`Type` — Type of graphics object

`'axes'`

Type of graphics object returned as the string `'axes'`.

`Tag` — Tag to associate with axes

`''` (default) | any string

Tag to associate with the axes, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with axes

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the axes object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent of axes

figure object | uipanel object | uitab object

Parent of axes, specified as figure object, uipanel object, or uitab object.

`Children` — Children of axes

empty `GraphicsPlaceholder` array | array of graphics objects

Children of axes, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property of the axes. To add a child to this list, set the `Parent` property of the child graphics object to the axes object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of axes object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The axes object handle is always visible.
- `'off'` — The axes object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The axes object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the axes at the command-line, but allows callback functions to access it.

If the axes object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`CurrentPoint` — Location of last mouse click

2-by-3 array

Location of last mouse click, specified as a 2-by-3 array in the axes coordinate system. The CurrentPoint property contains the coordinates of two points defined by the location of the pointer at the last mouse click, with respect to the requested axes.

If the click is within the axes in orthogonal projection, then the two points lie on the line that is perpendicular to the plane of the screen and that passes through the pointer. This is true for both 2-D and 3-D views.

The 3-D coordinates are the points, in the axes coordinate system, where this line intersects the front and back surfaces of the axes volume (which is defined by the axes x, y, and z limits).

The returned matrix is of the form:

```
[xfront yfront zfront
 xback  yback  zback]
```

The first row defines the point nearest to the camera position. The second row specified the point furthest from the camera position.

If the click is outside the axes in orthogonal projection, but within the figure, the returned values are:

- Back point — Point in the plane of the camera target (which is perpendicular to the viewing axis).
- Front point — Point in the camera position plane (which is perpendicular to the viewing axis).

These points lie on a line that passes through the pointer and is perpendicular to the camera target and camera position planes.

The values of the current point when using perspective projection can be different from the same point in orthographic projection because the shape of the axes volume can be different.

Clicking outside of the axes volume in perspective projection returns the front point as the current camera position at all times. Only the back point updates with the coordinates of a point that lies on a line extending from the camera position through the pointer and intersecting the camera target at that point.

For related information, see the axes `Projection`, `CameraPosition`, and `CameraTarget` properties. Also, see the figure `CurrentPoint` properties.

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the axes. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The axes object — You can access properties of the axes object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the axes. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the axes when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the axes.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The Visible property must be set to `'on'`. The HitTest property determines if the axes responds to the click or if an ancestor does.
- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'`. The HitTest property determines if the axes responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the axes passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The HitTest property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the axes. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the axes that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the axes object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the axes is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the axes tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the axes. Setting the `CreateFcn` property on an existing axes has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during axes creation. MATLAB executes the callback after creating the axes and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The axes object — You can access properties of the axes object from within the callback function. You also can access the axes object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the axes. MATLAB executes the callback before destroying the axes so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The axes object — You can access properties of the axes object from within the callback function. You also can access the axes object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of axes

`'off'` (default) | `'on'`

Deletion status of axes, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the axes begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the axes no longer exists.

Check the value of the BeingDeleted property to verify that the axes is not about to be deleted before querying or modifying it.

See Also

`axes` | `axis` | `box` | `caxis` | `cla` | `gca` | `grid`

More About

- "Access Property Values"
- "Graphics Object Properties"

axis

Set axis limits and appearance

Syntax

```
axis(limits)

axis style
axis mode
axis ydirection
axis visibility

lim = axis
[m,v,d] = axis('state')

___ = axis(ax,___)
```

Description

`axis(limits)` sets the *x*-axis and *y*-axis limits for the current axes. Specify `limits` as a four-element of the form `[xmin xmax ymin ymax]`. To additionally set the *z*-axis limits, specify a six-element vector. To additionally set the color limits, specify an eight-element vector.

`axis style` uses a predefined style to set the limits and scaling. For example, specify the style as `equal` to use equal data unit lengths along each axis.

`axis mode` sets whether the axes automatically chooses the limits or not. Specify the mode as `manual`, `auto`, or one of the semiautomatic options, for example, `'auto x'`.

`axis ydirection` controls the placement of the coordinate system origin and the direction of increasing *y* values. Set `ydirection` to `ij` to place the origin at the upper left corner of the axes. The *y* values increase from top to bottom. Set `ydirection` to `xy` to place the origin at the lower left corner. The *y* values increase from bottom to top. This is the default value.

`axis visibility` controls the visibility of the axes background. Set `visibility` to `off` to turn off the display of the axes background. Plots in the axes still display. Set `visibility` to `on` to display the axes background. This is the default value.

`lim = axis` returns the *x*-axis and *y*-axis limits for the current axes. For 3-D axes, it additionally returns the *z*-axis limits.

`[m,v,d] = axis('state')` returns the current settings for the axis limit selection, the axes visibility, and the *y*-axis direction.

Note: This syntax will be removed in a future release. Use the `XLimMode`, `YLimMode`, `ZLimMode`, `Visible`, and `YDir` properties of the axes to the get values instead.

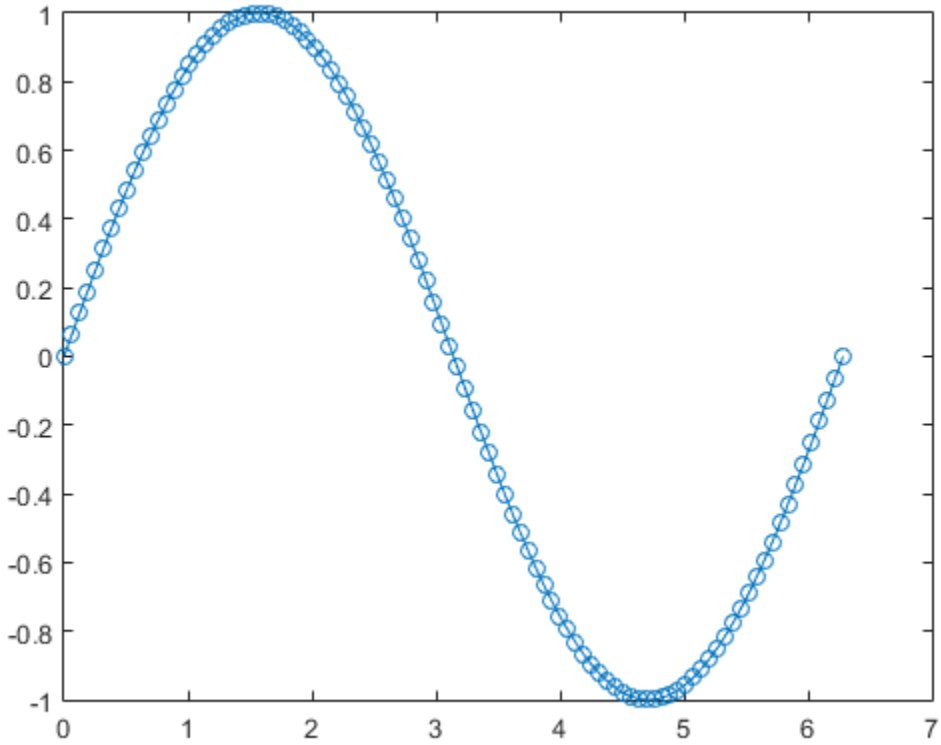
`___ = axis(ax, ___)` uses the axes specified by `ax` instead of the current axes. You can specify an axes with any of the input or output arguments in the previous syntaxes. When you specify an axes, use single quotes around input arguments that are character strings, for example, `axis(ax,'equal')`.

Examples

Set Axis Limits

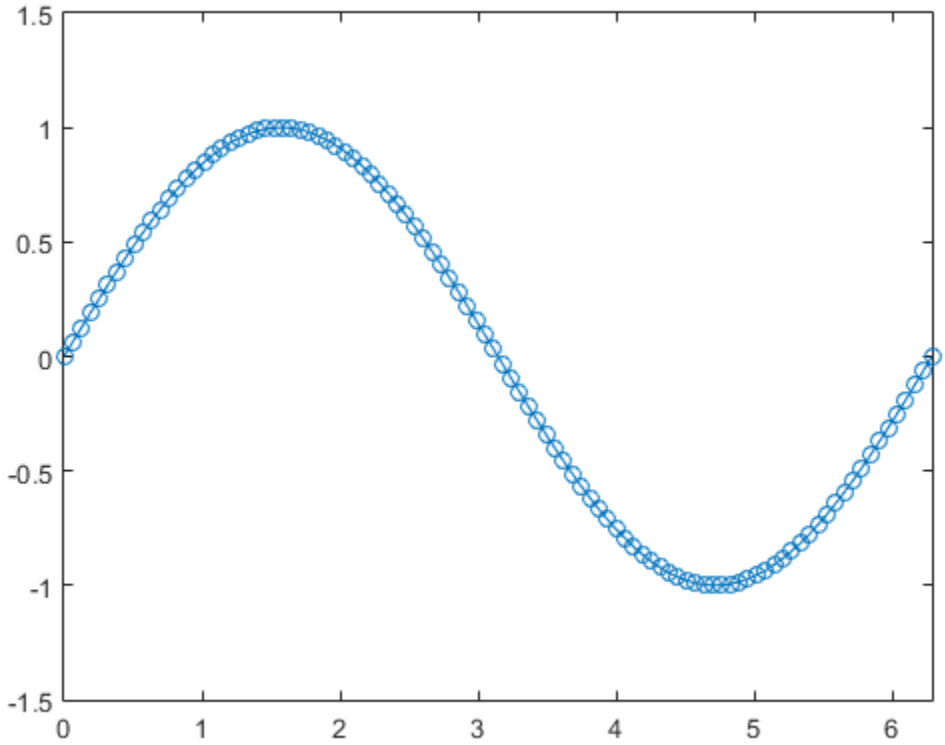
Plot the sine function.

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y,'-o')
```



Change the axis limits so that the *x*-axis ranges from 0 to 2π and the *y*-axis ranges from -1.5 to 1.5.

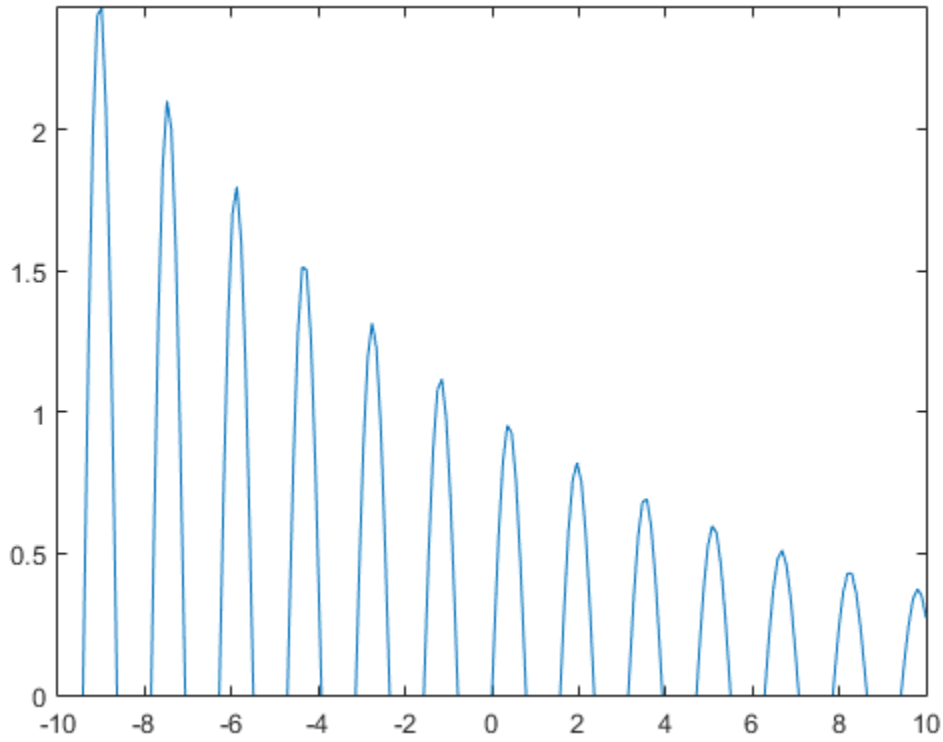
```
axis([0 2*pi -1.5 1.5])
```



Use Semiautomatic Axis Limits

Create a plot. Set the limits for the *x*-axis and set the minimum *y*-axis limit. Use an automatically calculated value for the maximum *y*-axis limit.

```
x = linspace(-10,10,200);
y = sin(4*x)./exp(.1*x);
plot(x,y)
axis([-10 10 0 inf])
```

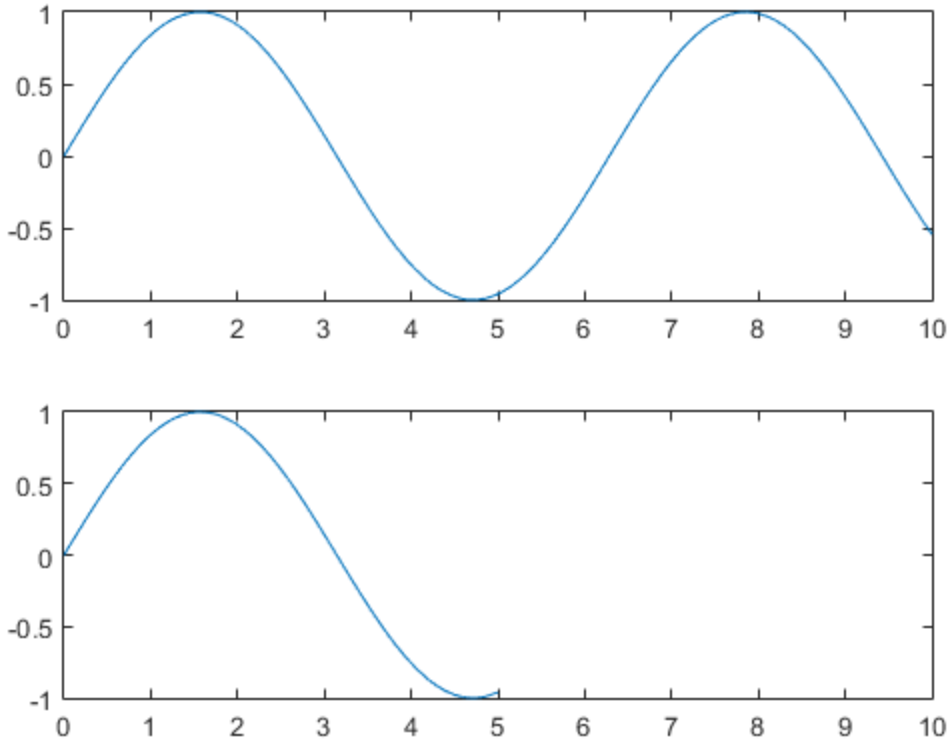


Set Axis Limits for Multiple Axes

Create a figure with two subplots. Plot a sine wave in each subplot. Then, set the axis limits for the subplots to the same values.

```
x1 = linspace(0,10,100);
y1 = sin(x1);
ax1 = subplot(2,1,1);
plot(ax1,x1,y1)

x2 = linspace(0,5,100);
y2 = sin(x2);
ax2 = subplot(2,1,2);
plot(ax2,x2,y2)
```

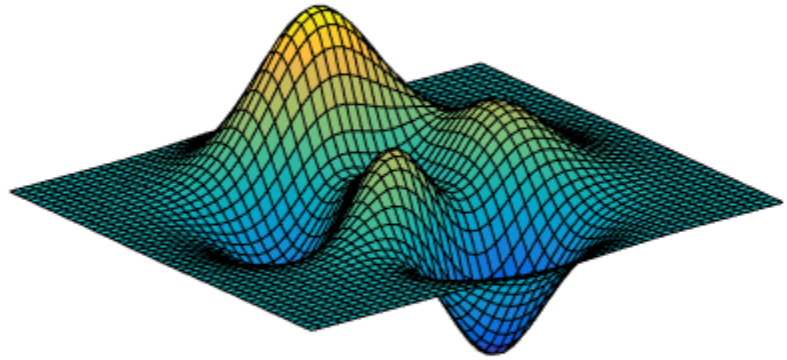
```
axis([ax1 ax2],[0 10 -1 1])
```



Display Plot Without Axes Background

Plot a surface without displaying the axes lines and background.

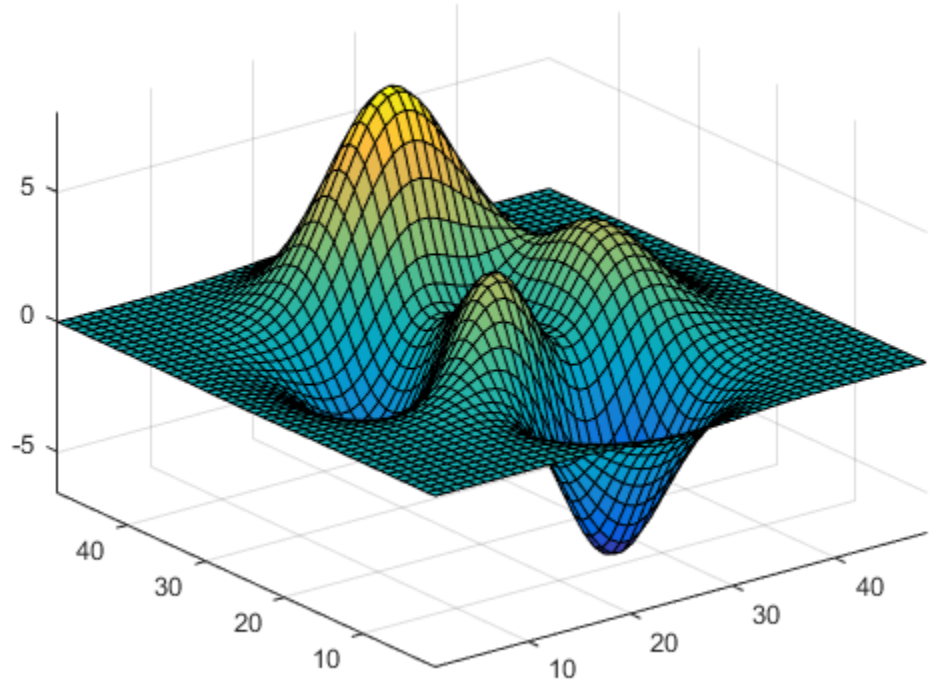
```
surf(peaks)
axis off
```



Use Tight Axis Limits and Return Values

Plot a surface. Set the axis limits to equal the range of the data so that the plot extends to the edges of the axes.

```
surf(peaks)
axis tight
```



Return the values of the current axis limits.

```
l = axis
```

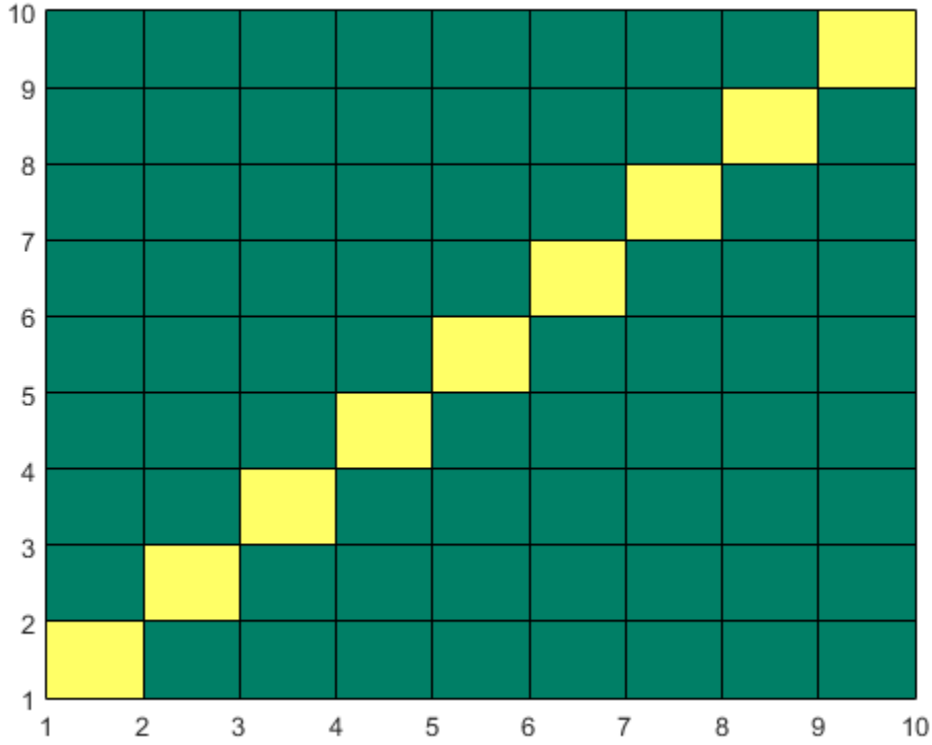
```
l =

    1.0000   49.0000    1.0000   49.0000   -6.5466    8.0752
```

Change Direction of Coordinate System

Create a checkerboard plot and change the direction of the coordinate system.

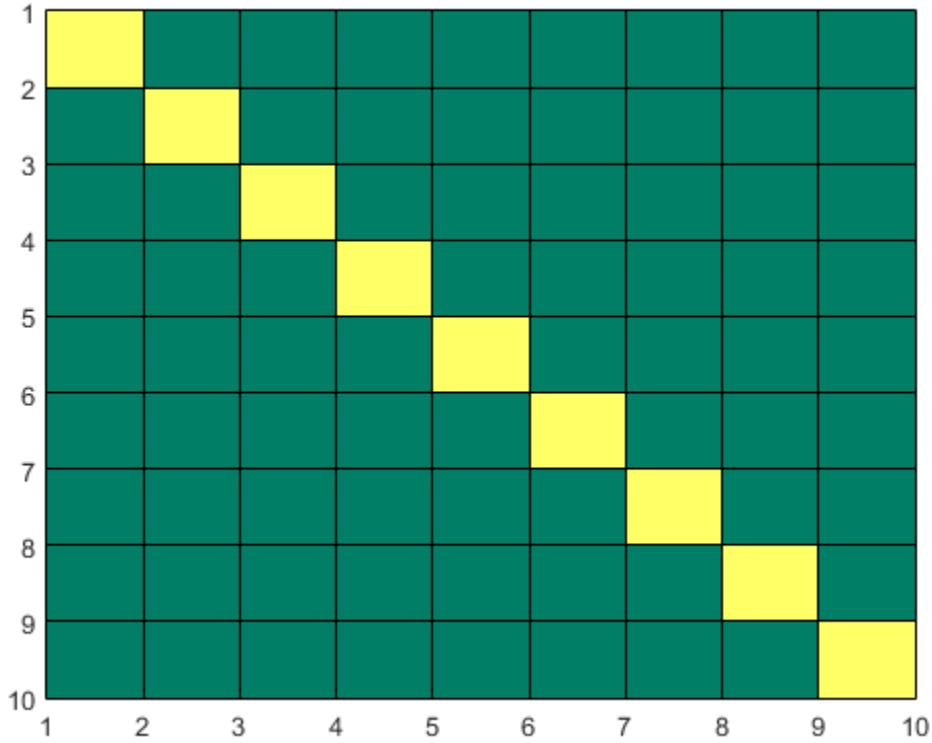
First, create the plot using the `summer` colormap. By default, the *x* values increase from left to right and the *y* values increase from bottom to top.

```
C = eye(10);
pcolor(C)
colormap summer
```



Reverse the coordinate system so that the *y* values increase from top to bottom.

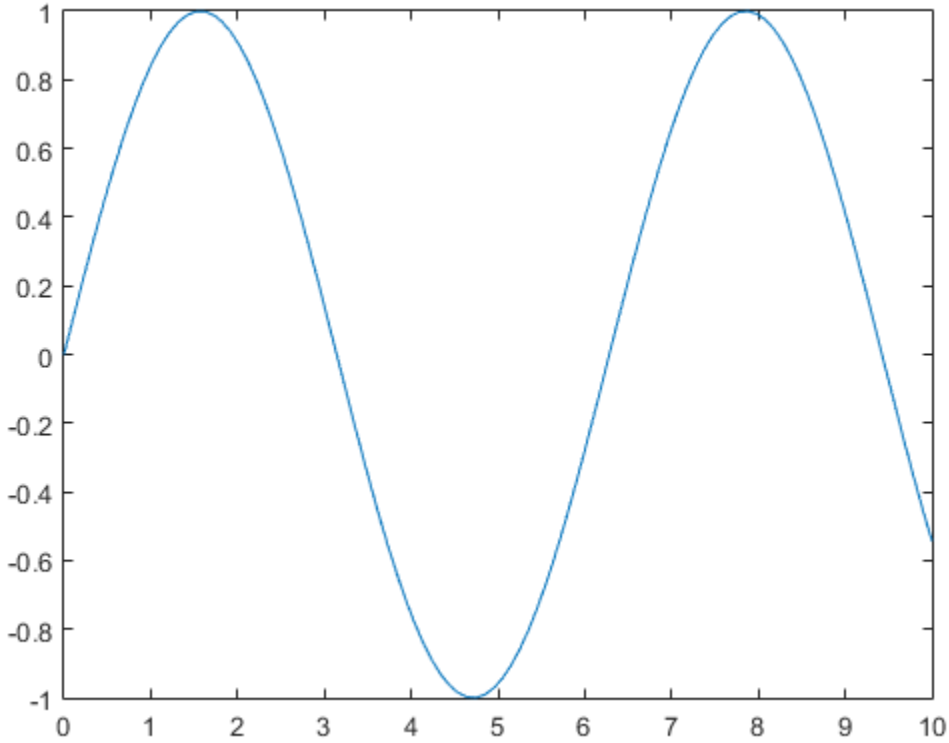
```
axis ij
```



Retain Current Axis Limits When Adding New Plots

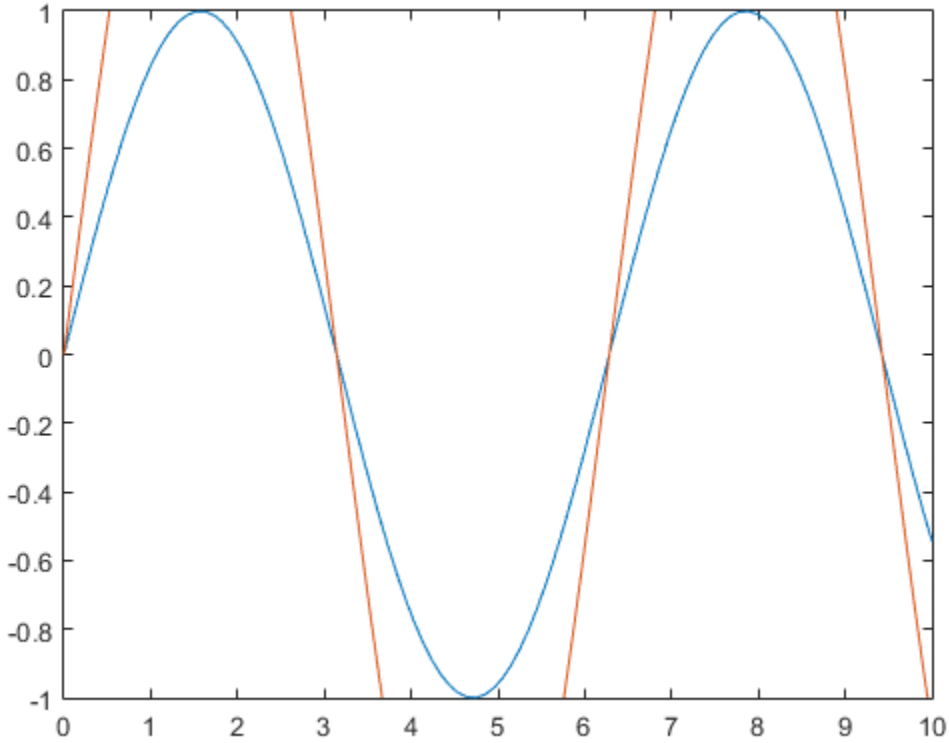
Plot a sine wave.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
```



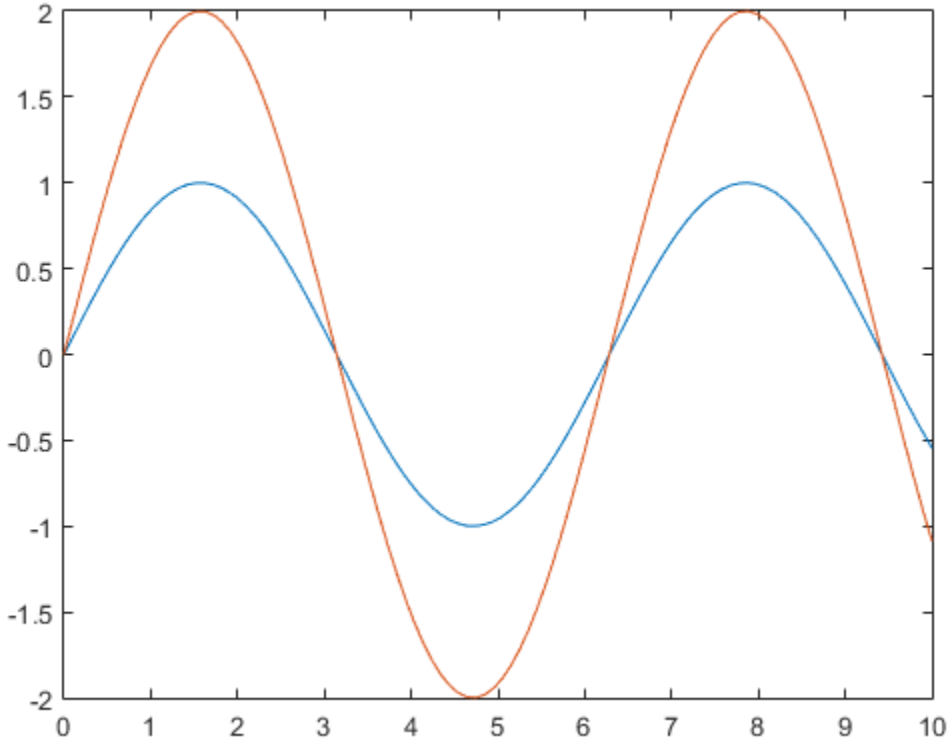
Add another sine wave to the axes using `hold on`. Keep the current axis limits by setting the limits mode to manual.

```
y2 = 2*sin(x);
hold on
axis manual
plot(x,y2)
hold off
```



If you want the axes to choose the appropriate limits, set the limits mode back to automatic.

```
axis auto
```



Input Arguments

`limits` — Axis limits

four-element vector | six-element vector | eight-element vector

Axis limits, specified as a vector of four, six, or eight elements in one of these forms:

- `[xmin xmax ymin ymax]` — Set the *x*-axis limits to range from `xmin` to `xmax`. Set the *y*-axis limits to range from `ymin` to `ymax`.
- `[xmin xmax ymin ymax zmin zmax]` — Additionally set the *z*-axis limits to range from `zmin` to `zmax`.

- `[xmin xmax ymin ymax zmin zmax cmin cmax]` — Additionally set the color limits. `cmin` is the data value that maps to the first color in the colormap. `cmax` is the data value that maps to the last color in the colormap.

Note: For partially automatic limits, use `inf` or `-inf` for the limits you want the axes to choose automatically. For example, `axis([-inf 10 0 inf])` lets the axes choose the appropriate minimum *x*-axis limit and maximum *y*-axis limit. It uses the specified values for the maximum *x*-axis limit and minimum *y*-axis limit.

If you specify the limits, then the `XLim`, `YLim`, `ZLim`, and `CLim` properties for the axes change to the specified values. Additionally, the associated mode properties change to `'manual'`.

Example: `[0 1 0 1]`

Example: `[0 1 0 1 0 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`mode` — Manual, automatic, or semiautomatic selection of axis limits

`manual` | `auto` | `'auto x'` | `'auto y'` | `'auto z'` | `'auto xy'` | `'auto xz'` | `'auto yz'`

Manual, automatic, or semiautomatic selection of axis limits, specified as one of the values in this table.

Value	Description	Axes Properties That Change
`manual`	Freeze all axis limits at their current values.	Sets `XLimMode`, `YLimMode`, and `ZLimMode` to `'manual'`.
`auto`	The axes automatically chooses all axis limits.	Sets `XLimMode`, `YLimMode`, and `ZLimMode` to `'auto'`.
`'auto x'`	The axes automatically chooses the *x*-axis limits.	Sets `XLimMode` to `'auto'`.
`'auto y'`	The axes automatically chooses the *y*-axis limits.	Sets `YLimMode` to `'auto'`.
`'auto z'`	The axes automatically chooses the *z*-axis limits.	Sets `ZLimMode` to `'auto'`.

Value	Description	Axes Properties That Change
`'auto xy'`	The axes automatically chooses the *x*-axis and *y*-axis limits.	Sets `XLimMode` and `YLimMode` to `'auto'`.
`'auto xz'`	The axes automatically chooses the *x*-axis and *z*-axis limits.	Sets `XLimMode` and `ZLimMode` to `'auto'`.
`'auto yz'`	The axes automatically chooses the *y*-axis and *z*-axis limits.	Sets `YLimMode` and `ZLimMode` to `'auto'`.

`style` — Axis limits and scaling

`tight | fill | equal | image | square | vis3d | normal`

Axis limits and scaling, specified as one of these values.

Value	Description	Axes Properties That Change
`tight`	Fit the axes box tightly around the data by setting the axis limits equal to the range of the data.	`XLimMode`, `YLimMode`, and `ZLimMode` change to `'auto'`. The limits automatically update to incorporate new data added to the axes. To keep the limits from changing when using `hold on`, use `axis tight manual`.
`equal`	Use the same length for the data units along each axis.	Sets `DataAspectRatio` to `[1 1 1]`, sets `PlotBoxAspectRatio` to `[3 4 4]`, and sets the associated mode properties to manual. Disables the "stretch-to-fill" behavior.
`image`	Use the same length for the data units along each axis and fit the axes box tightly around the data.	Sets `DataAspectRatio` to `[1 1 1]`and sets the associated mode property to manual. Disables the "stretch-to-fill" behavior.

Value	Description	Axes Properties That Change
`square`	Use axis lines with equal lengths. Adjust the increments between data units accordingly.	Sets `PlotBoxAspectRatio` to `[1 1 1]` and sets the associated mode property to manual. Disables the "stretch-to-fill" behavior.
`fill`	Enable the "stretch-to-fill" behavior (the default). The lengths of each axis line fill the position rectangle defined in the `Position` property of the axes.	Sets the plot box aspect ratio mode and data aspect ratio mode properties to auto.
`vis3d`	Freeze the aspect ratio properties.	Sets the plot box aspect ratio mode and data aspect ratio mode properties to manual.
`normal`	Restore the default behavior.	Sets the plot box aspect ratio mode and data aspect ratio mode properties to auto.

For more information on the plot box aspect ratio and the data aspect ratio, see the `PlotBoxAspectRatio` and `DataAspectRatio` properties for the axes.

You can combine style options. The options are evaluated from left to right, so subsequent options can overwrite properties set by prior ones.

`ydirection` — *y*-axis direction

`xy` (default) | `ij`

y-axis direction, specified as one of these values:

- `xy` — Default direction. For axes with a 2-D view, the *y*-axis is vertical with values increasing from bottom to top.
- `ij` — Reverse direction. For axes with a 2-D view, the *y*-axis is vertical with values increasing from top to bottom.

`visibility` — Axes lines and background visibility

`on` (default) | `off`

Axes lines and background visibility, specified as either `on` or `off`. Specifying the visibility sets the `Visible` property of the axes to the same value.

ax — Axes objects

scalar | vector

Axes objects, specified as a scalar or a vector. If you do not specify an axes, then `axis` sets the limits for the current axes (`gca`).

When you specify an axes, use single quotes around other input arguments that are character strings.

Example: `axis(ax,'tight')`

Example: `axis(ax,limits)`

Example: `axis(ax,'manual')`

Output Arguments

lim — Current limit values

four-element vector | six-element vector

Current limit values, returned as a four-element or six-element vector. For axes with a 2-D view, `lim` is of the form `[xmin xmax ymin ymax]`. For axes with a 3-D view, `lim` is of the form `[xmin xmax ymin ymax zmin zmax]`.

The `XLim`, `YLim`, and `ZLim` properties for the axes contain the limit values.

More About

Tips

- If an axes does not exist, the `axis` function creates one.
- Use `hold on` to keep plotting functions from overriding preset axis limits.

See Also

Functions

`caxis` | `grid` | `subplot` | `title` | `xlim` | `ylim` | `zlim`

Properties

Axes Properties

Introduced before R2006a

Using alphaShape Objects

Polygons and polyhedra from points in 2-D and 3-D

An `alphaShape` creates a bounding area or volume that envelops a set of 2-D or 3-D points. You can manipulate the `alphaShape` object to tighten or loosen the fit around the points to create a nonconvex region. Additionally, you can add or remove points or suppress holes or regions.

After you create an `alphaShape` object, you can perform geometric queries. For example, you can determine if a point is inside the shape or find the number of regions that make up the shape. You can also calculate useful quantities like area, perimeter, surface area, or volume as well as plot the shape for visual inspection.

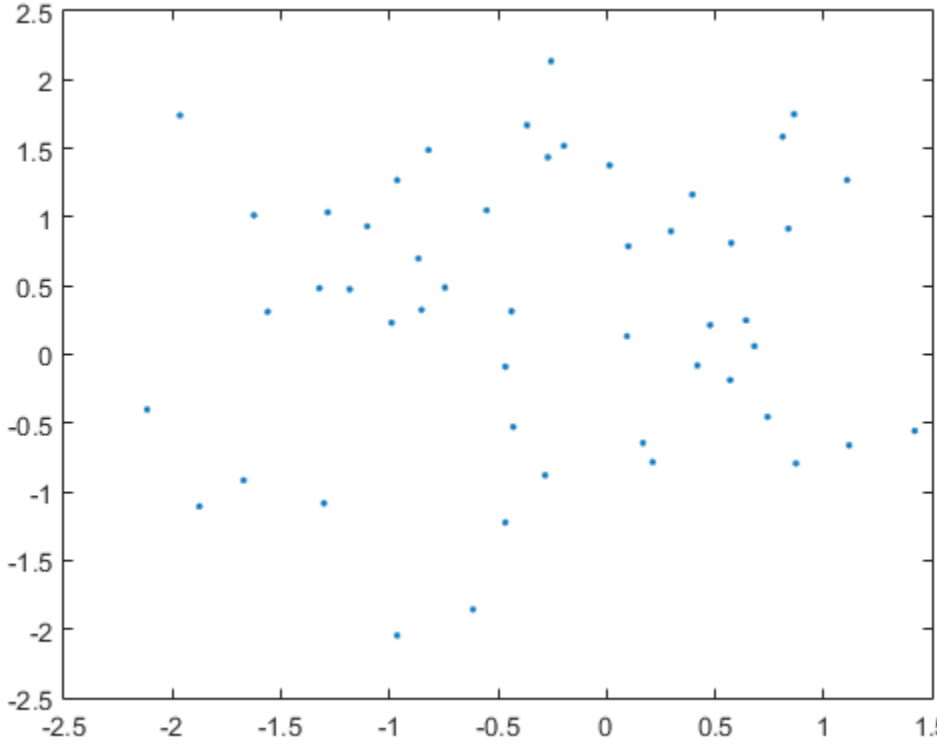
Examples

Using alphaShape Objects

This example shows how to change an alpha shape object and compute useful quantities that describe the alpha shape.

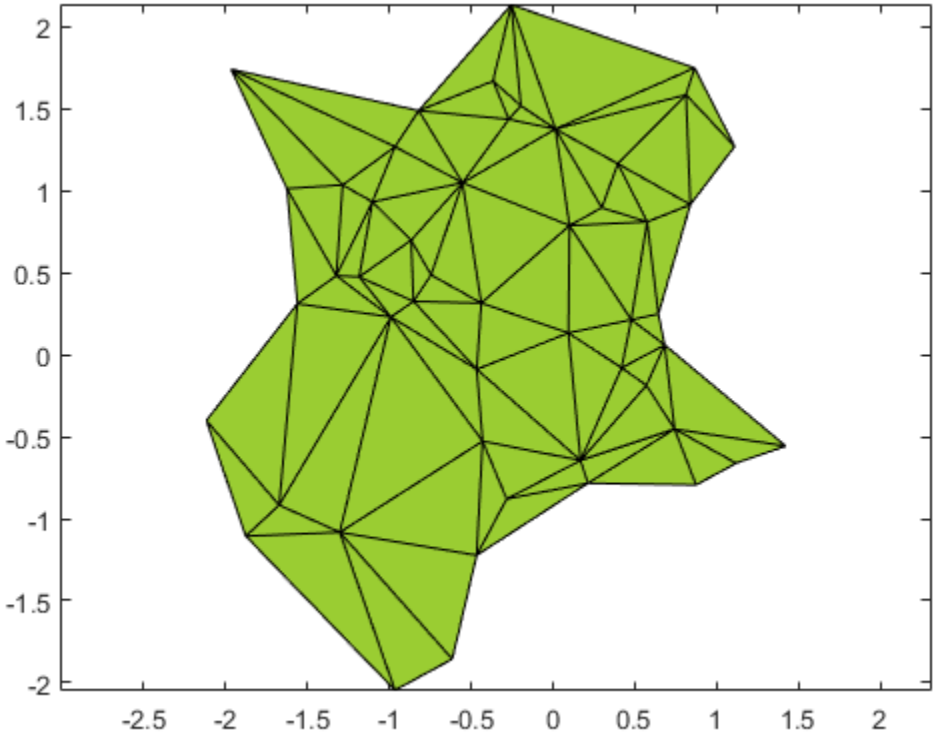
Create and plot a set of 2-D points.

```
x = gallery('normaldata',50,1,1);
y = gallery('normaldata',50,1,2);
plot(x,y,'.')
```



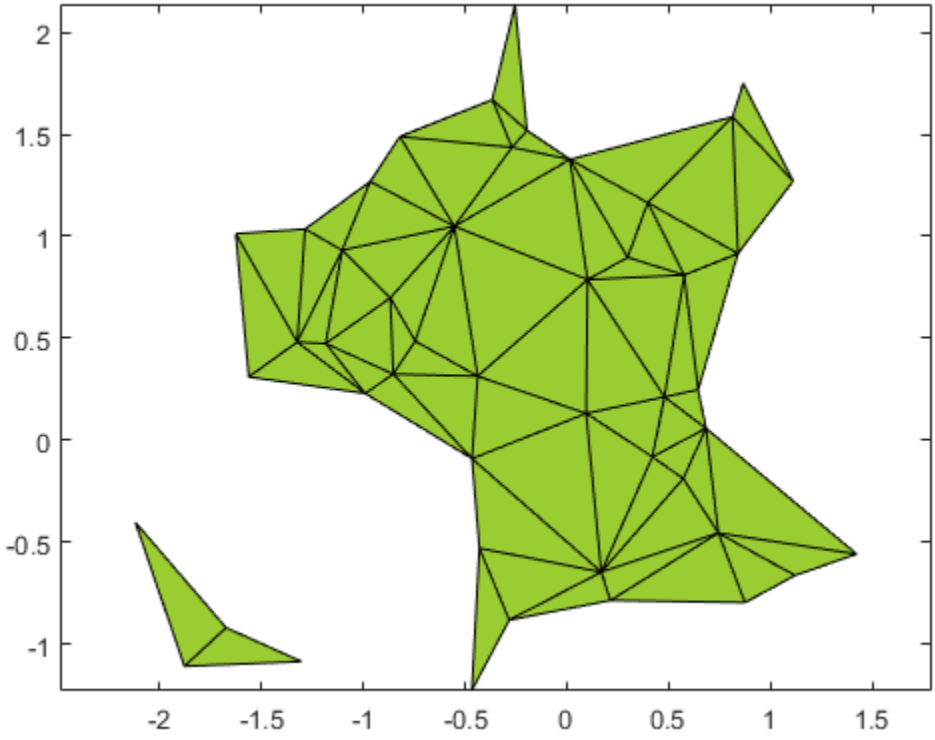
Create and plot an alpha shape using an alpha radius of 0.7.

```
shp = alphaShape(x,y,0.7);
plot(shp)
```



Change the alpha radius to 0.5 to tighten the boundary around the set of points.

```
shp.Alpha = 0.5;
plot(shp)
```



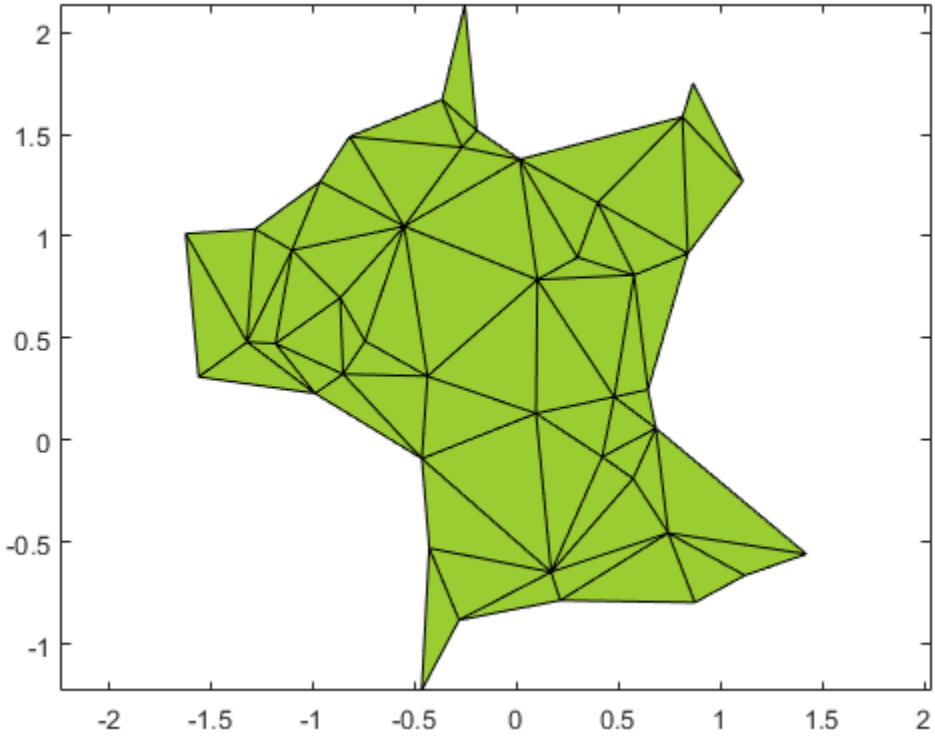
Query the number of regions that make up the new alpha shape.

```
numRegions(shp)
```

```
ans =

     2
```

Suppress the smaller of the two regions.

```
shp.RegionThreshold = 1;
plot(shp)
```



Compute the perimeter and area of the remaining region.

```
regionperims = perimeter(shp)
regionareas = area(shp)
```

```
regionperims =

   11.8543


regionareas =

    4.5407
```

Properties

`Points` — Coordinates of points
matrix

Coordinates of points, specified as a matrix with two or three columns (for 2-D or 3-D point sets). These points are initially used to create the alpha shape, excluding duplicates.

Data Types: `double`

`Alpha` — Alpha radius
nonnegative scalar

Alpha radius, specified as a nonnegative scalar. The alpha radius is the radius of the alpha disk or sphere which sweeps over the points to create the alpha shape.

The default alpha radius is `a = criticalAlpha(shp,'all-points')`, which is the smallest alpha radius that produces an alpha shape enclosing all points. Specify `a = criticalAlpha(shp,'one-region')` to use the smallest alpha radius that produces an alpha shape with only one region.

The extreme values of `Alpha`:

- If `Alpha` is `Inf`, then `alphaShape` produces the convex hull.
- If `Alpha` is `0`, then the resulting `alphaShape` is empty.

Data Types: `double`

`HoleThreshold` — Maximum area (2-D) or volume (3-D) of interior holes or voids to fill in
`0` (default) | finite nonnegative scalar

Maximum area or volume of interior holes or voids to fill in, specified as a finite nonnegative scalar.

- For 2-D, `HoleThreshold` specifies the maximum area of interior holes to fill in.
- For 3-D, `HoleThreshold` specifies the maximum volume of interior voids to fill in. Holes extending completely through the 3-D alpha shape cannot be filled in.

The default value is `0`, so that `alphaShape` does not suppress any holes or voids. The application of the `HoleThreshold` and `RegionThreshold` properties is order-dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

`RegionThreshold` — Maximum area (2-D) or volume (3-D) of regions to suppress

`0` (default) | finite nonnegative scalar

Maximum area (2-D) or volume (3-D) of regions to suppress, specified as a finite nonnegative scalar.

The default value is `0`, so that `alphaShape` does not suppress any regions. The application of the `HoleThreshold` and `RegionThreshold` properties is order-dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

Object Functions

`alphaSpectrum` `criticalAlpha` `numRegions` `inShape` `alphaTriangulation` `boundaryFacets` `perimeter` `area` `surfaceArea` `volume` `plot` `nearestNeighbor`

Create Object

Create an `alphaShape` using the `alphaShape` function.

See Also

`boundary`

balance

Diagonal scaling to improve eigenvalue accuracy

Syntax

```
[T,B] = balance(A)
[S,P,B] = balance(A)
B = balance(A)
B = balance(A,'noperm')
```

Description

`[T,B] = balance(A)` returns a similarity transformation `T` such that `B = T\A*T`, and `B` has, as nearly as possible, approximately equal row and column norms. `T` is a permutation of a diagonal matrix whose elements are integer powers of two to prevent the introduction of roundoff error. If `A` is symmetric, then `B == A` and `T` is the identity matrix.

`[S,P,B] = balance(A)` returns the scaling vector `S` and the permutation vector `P` separately. The transformation `T` and balanced matrix `B` are obtained from `A`, `S`, and `P` by `T(:,P) = diag(S)` and `B(P,P) = diag(1./S)*A*diag(S)`.

`B = balance(A)` returns just the balanced matrix `B`.

`B = balance(A,'noperm')` scales `A` without permuting its rows and columns.

Examples

This example shows the basic idea. The matrix `A` has large elements in the upper right and small elements in the lower left. It is far from being symmetric.

```
A = [1   100   10000; .01   1   100; .0001   .01   1]
A =
   1.0e+04 *
    0.0001    0.0100    1.0000
    0.0000    0.0001    0.0100
    0.0000    0.0000    0.0001
```

Balancing produces a diagonal matrix T with elements that are powers of two and a balanced matrix B that is closer to symmetric than A.

```
[T,B] = balance(A)
T =
   1.0e+03 *
    2.0480         0         0
         0    0.0320         0
         0         0    0.0003
B =
    1.0000    1.5625    1.2207
    0.6400    1.0000    0.7813
    0.8192    1.2800    1.0000
```

To see the effect on eigenvectors, first compute the eigenvectors of A, shown here as the columns of V.

```
[V,E] = eig(A); V
V =
0.9999         -0.9999             -0.9999
0.0100          0.0059 + 0.0085i    0.0059 - 0.0085i
0.0001          0.0000 - 0.0001i    0.0000 + 0.0001i
```

Note that all three vectors have the first component the largest. This indicates V is badly conditioned; in fact `cond(V)` is `8.7766e+003`. Next, look at the eigenvectors of B.

```
[V,E] = eig(B); V
V =
0.6933         -0.6993             -0.6993
0.4437          0.2619 + 0.3825i    0.2619 - 0.3825i
0.5679          0.2376 - 0.4896i    0.2376 + 0.4896i
```

Now the eigenvectors are well behaved and `cond(V)` is `1.4421`. The ill conditioning is concentrated in the scaling matrix; `cond(T)` is `8192`.

This example is small and not really badly scaled, so the computed eigenvalues of A and B agree within roundoff error; balancing has little effect on the computed results.

Limitations

Balancing can destroy the properties of certain matrices; use it with some care. If a matrix contains small elements that are due to roundoff error, balancing might scale them up to make them as significant as the other elements of the original matrix.

More About

Tips

- Nonsymmetric matrices can have poorly conditioned eigenvalues. Small perturbations in the matrix, such as roundoff errors, can lead to large perturbations in the eigenvalues. The condition number of the eigenvector matrix,

  ```
  cond(V) = norm(V)*norm(inv(V))
  ```

  where

  ```
  [V,T] = eig(A)
  ```

  relates the size of the matrix perturbation to the size of the eigenvalue perturbation. Note that the condition number of `A` itself is irrelevant to the eigenvalue problem.

  Balancing is an attempt to concentrate any ill conditioning of the eigenvector matrix into a diagonal scaling. Balancing usually cannot turn a nonsymmetric matrix into a symmetric matrix; it only attempts to make the norm of each row equal to the norm of the corresponding column.

  Note The MATLAB eigenvalue function, `eig(A)`, automatically balances `A` before computing its eigenvalues. Turn off the balancing with `eig(A,'nobalance')`.

See Also

`eig`

bandwidth

Lower and upper matrix bandwidth

Syntax

```
B = bandwidth(A,type)
```

```
[lower,upper] = bandwidth(A)
```

Description

`B = bandwidth(A,type)` returns the bandwidth of matrix `A` specified by `type`. Specify `type` as `'lower'` for the lower bandwidth, or `'upper'` for the upper bandwidth.

`[lower,upper] = bandwidth(A)` returns the lower bandwidth, `lower`, and upper bandwidth, `upper`, of matrix `A`.

Examples

Find Bandwidth of Triangular Matrix

Create a 6-by-6 lower triangular matrix.

```
A = tril(magic(6))
```

```
A =

    35     0     0     0     0     0
     3    32     0     0     0     0
    31     9     2     0     0     0
     8    28    33    17     0     0
    30     5    34    12    14     0
     4    36    29    13    18    11
```

Find the lower bandwidth of `A` by specifying `type` as `'lower'`.

```
B = bandwidth(A,'lower')
```

```
B =

     5
```

The result is 5 because every diagonal below the main diagonal has nonzero elements.

Find the upper bandwidth of A by specifying `type` as `'upper'`.

```
B = bandwidth(A,'upper')
```

```
B =

     0
```

The result is 0 because there are no nonzero elements above the main diagonal.

Find Bandwidth of Sparse Block Matrix

Create a 100-by-100 sparse block matrix.

```
B = kron(speye(25),ones(4));
```

View a 10-by-10 section of elements from the top left of B.

```
full(B(1:10,1:10))
```

```
ans =

     1     1     1     1     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     0     0     1     1
     0     0     0     0     0     0     0     0     1     1
```

B has 4-by-4 blocks of ones centered on the main diagonal.

Find both the lower and upper bandwidths of B by specifying two output arguments.

```
[lower,upper] = bandwidth(B)
```

```
lower =
```

```
     3

upper =

     3
```

The lower and upper bandwidths are both 3.

Input Arguments

A — Input matrix

2-D numeric matrix

Input matrix, specified as a 2-D numeric matrix. A can be either full or sparse.

Data Types: single | double
Complex Number Support: Yes

type — Bandwidth type

'lower' | 'upper'

Bandwidth type, specified as 'lower' or 'upper'.

- Specify 'lower' for the lower bandwidth (below the main diagonal).
- Specify 'upper' for the upper bandwidth (above the main diagonal).

Data Types: char

Output Arguments

B — Lower or upper bandwidth

nonnegative integer scalar

Lower or upper bandwidth, returned as a nonnegative integer scalar.

- If type is 'lower', then $0 \le B \le$ size(A,1)-1.
- If type is 'upper', then $0 \le B \le$ size(A,2)-1.

`lower` — Lower bandwidth

nonnegative integer scalar

Lower bandwidth, returned as a nonnegative integer scalar. `lower` is in the range `0 ≤ lower ≤ size(A,1)-1`.

`upper` — Upper bandwidth

nonnegative integer scalar

Upper bandwidth, returned as a nonnegative integer scalar. `upper` is in the range `0 ≤ upper ≤ size(A,2)-1`.

More About

Upper and Lower Bandwidth

The upper and lower bandwidths of a matrix are measured by finding the last diagonal (above or below the main diagonal, respectively) that contains nonzero values.

That is, for a matrix A with elements A_{ij}:

- The upper bandwidth B_1 is the smallest number such that $A_{ij} = 0$ whenever $j - i > B_1$.
- The lower bandwidth B_2 is the smallest number such that $A_{ij} = 0$ whenever $i - j > B_2$.

Note that this measurement does not disallow intermediate diagonals in a band from being all zero, but instead focuses on the location of the last diagonal containing nonzeros. By convention, the upper and lower bandwidths of an empty matrix are both zero.

Tips

- Use the `isbanded` function to test if a matrix is within a specific lower and upper bandwidth.

See Also

`diag` | `isbanded` | `isdiag` | `istril` | `istriu`

Introduced in R2014a

bar

Bar graph

Syntax

```
bar(y)
bar(x,y)

bar(___,width)
bar(___,style)
bar(___,color)
bar(___,Name,Value)

bar(ax,___)

b = bar(___)
```

Description

`bar(y)` creates a bar graph with one bar for each element in `y`. If `y` is a matrix, then `bar` groups the bars according to the rows in `y`.

`bar(x,y)` draws the bars at the locations specified by `x`.

`bar(___,width)` sets the relative bar width, which controls the separation of bars within a group. Specify `width` as a scalar value. Use this option with any of the input argument combinations in the previous syntaxes.

`bar(___,style)` specifies the style of the bar groups. For example, use `'stacked'` to display each group as one multicolored bar.

`bar(___,color)` sets the color for all the bars. For example, use `'r'` for red bars.

`bar(___,Name,Value)` sets bar series properties using one or more `Name,Value` pair arguments. The settings apply to all bars plotted. For example, use `'EdgeColor','black'` to outline all the bars in black.

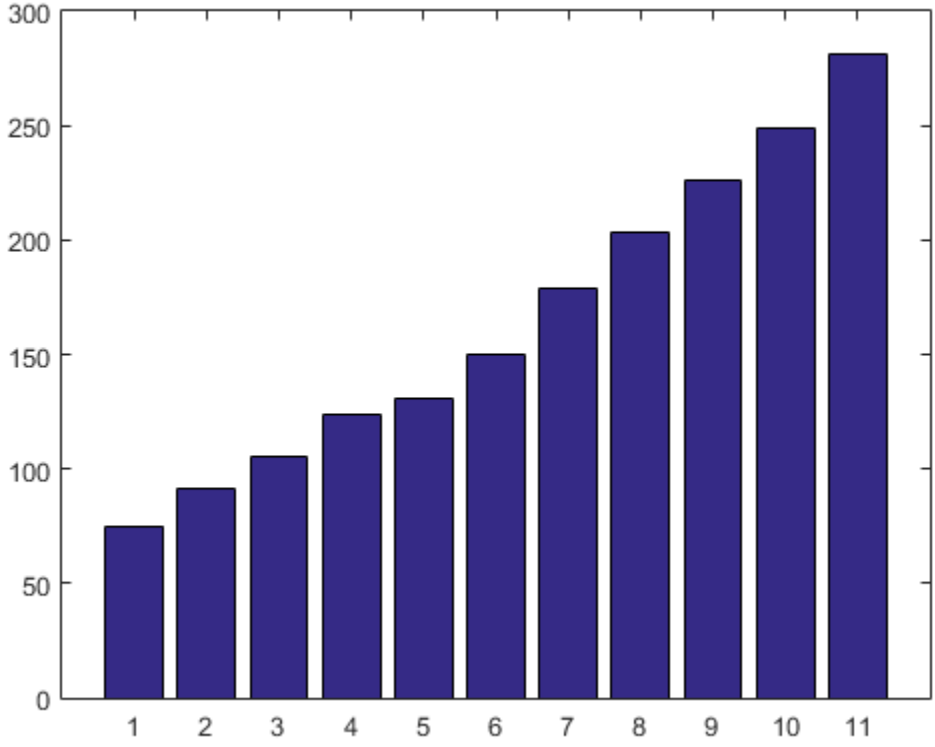
`bar(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`b = bar( ___ )` returns one or more bar series objects. If `y` is a vector, then `bar` creates one bar series object. If `y` is a matrix, then `bar` creates a bar series object for each column. Use `b` to make future modifications to the bars after they are created.

Examples

Create Bar Graph

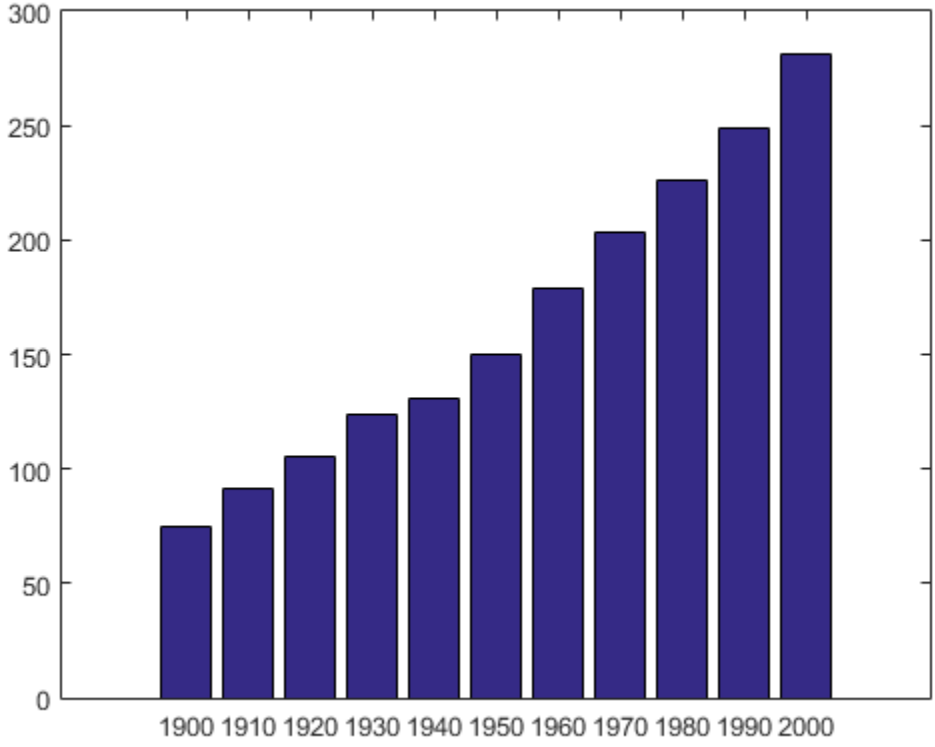
```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y)
```



Specify Bar Locations

Specify the bar locations along the *x*-axis.

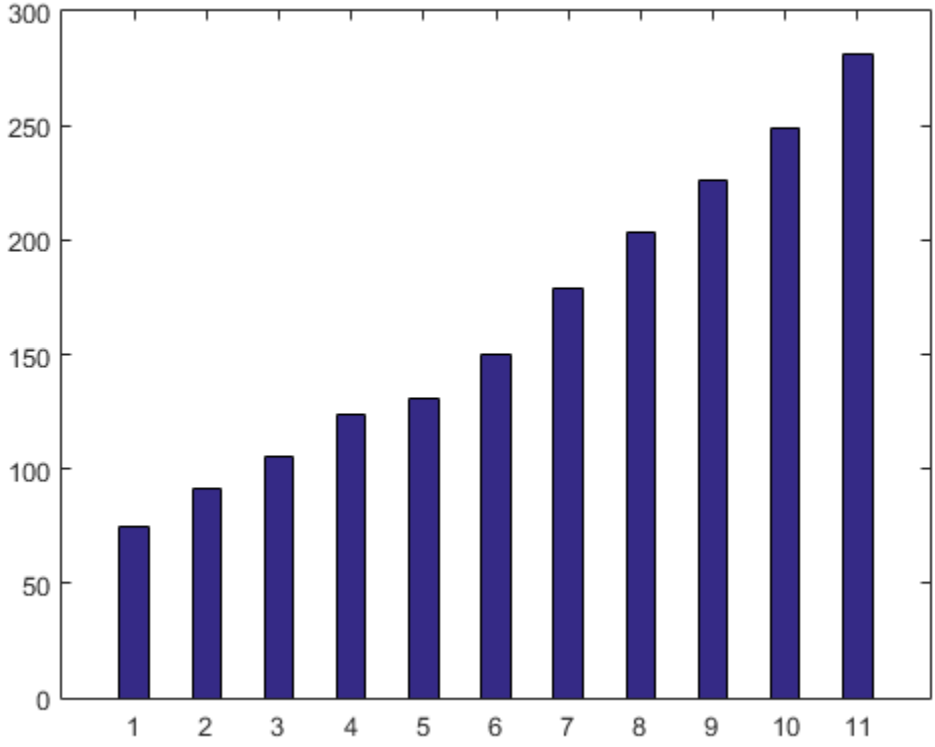
```
x = 1900:10:2000;
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(x,y)
```



Specify Bar Width

Set the width of each bar to 40 percent of the total space available for each bar.

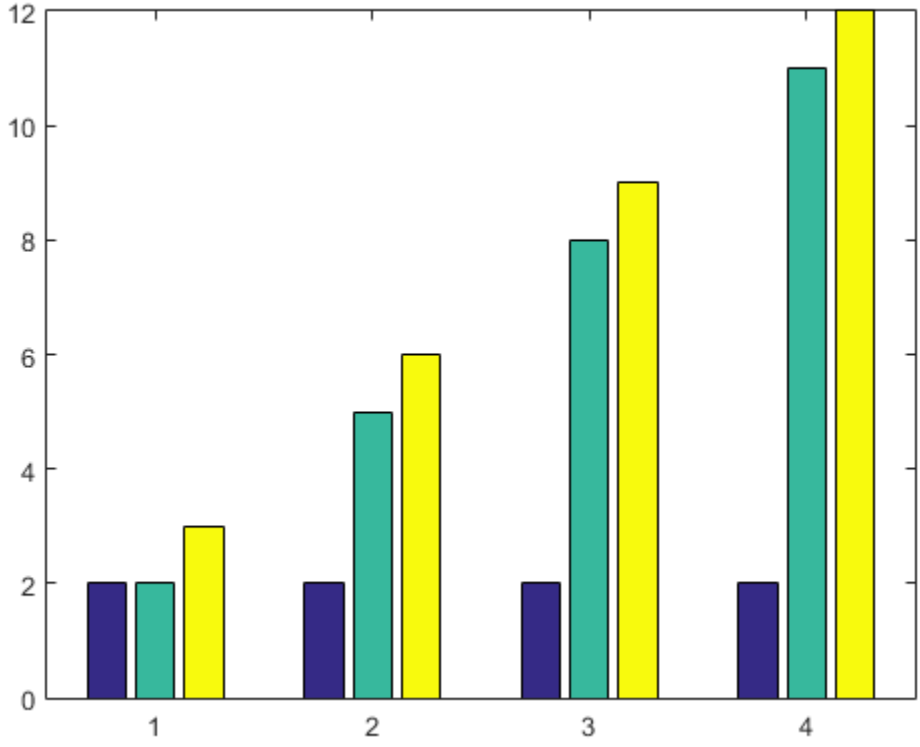
```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y,0.4)
```



Display Groups of Bars

Display four groups of three bars.

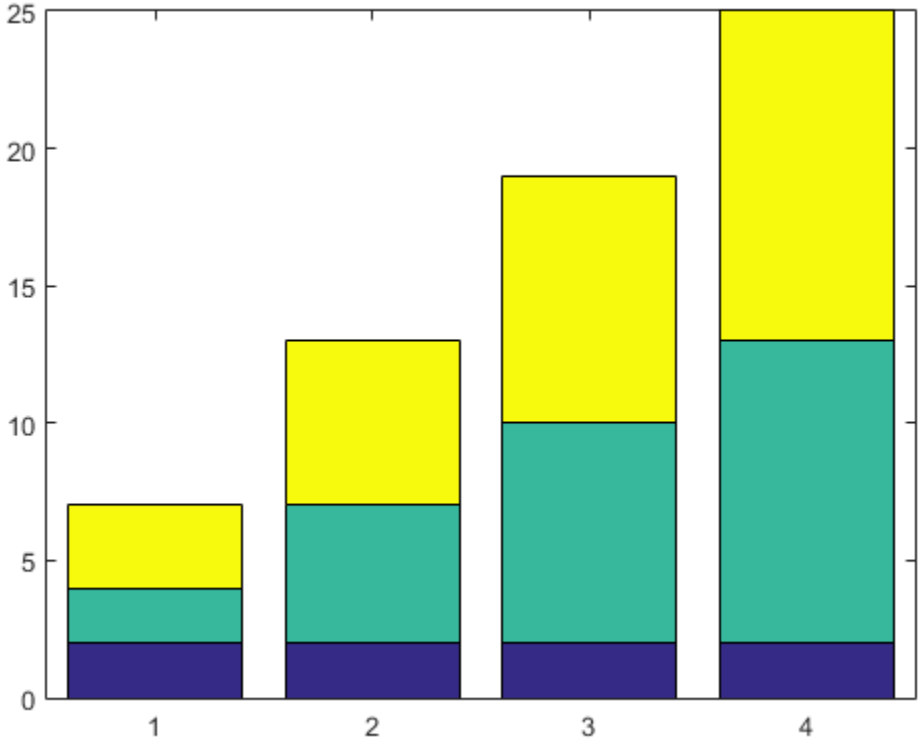
```
y = [2 2 3; 2 5 6; 2 8 9; 2 11 12];
bar(y)
```



Display Stacked Bars

Display one bar for each row of the matrix. The height of each bar is the sum of the elements in the row.

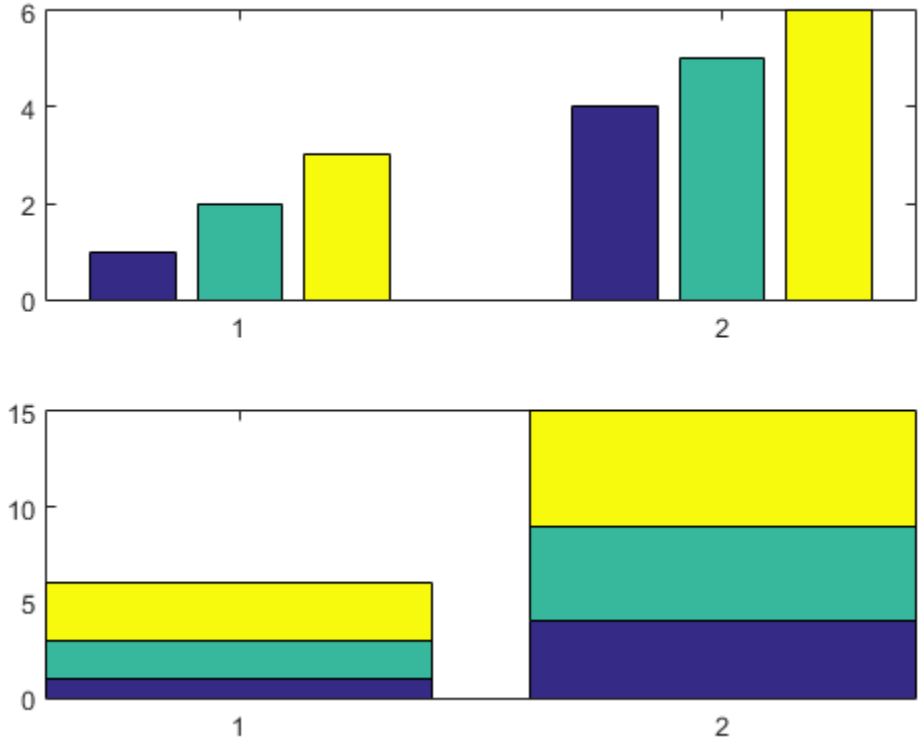
```
y = [2 2 3; 2 5 6; 2 8 9; 2 11 12];
bar(y,'stacked')
```



Specify Subplot for Bar Graph

Create a figure with two subplots. In the upper subplot, plot a bar graph. In the lower subplot, plot a stacked bar graph of the same data.

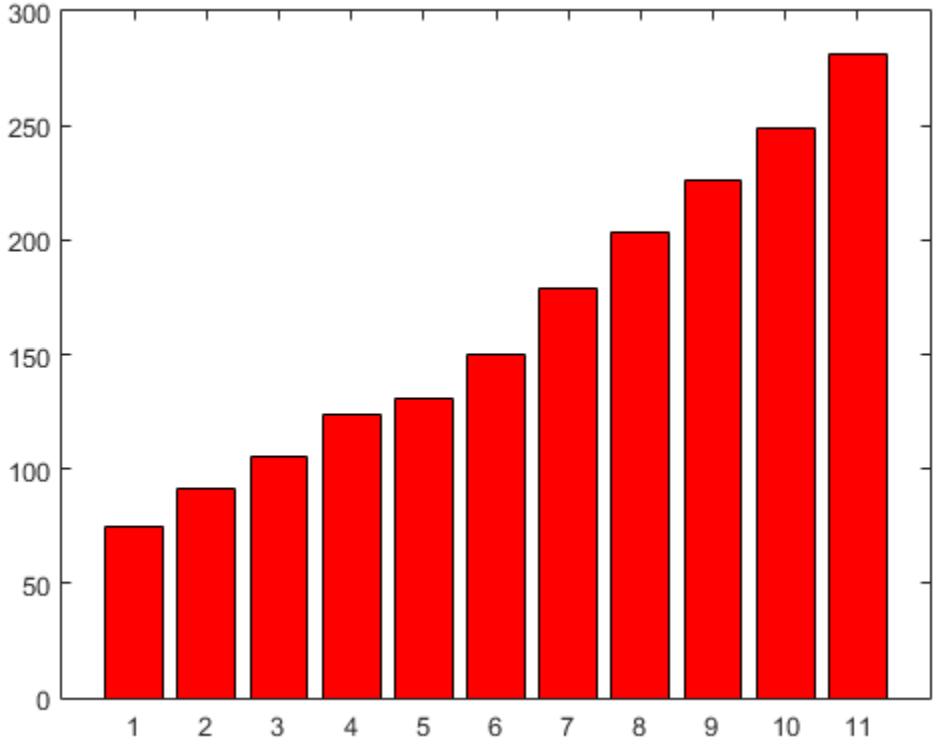
```
y = [1 2 3; 4 5 6];
ax1 = subplot(2,1,1);
bar(ax1,y)

ax2 = subplot(2,1,2);
bar(ax2,y,'stacked')
```



Specify Bar Color

Create a bar graph using red bars.

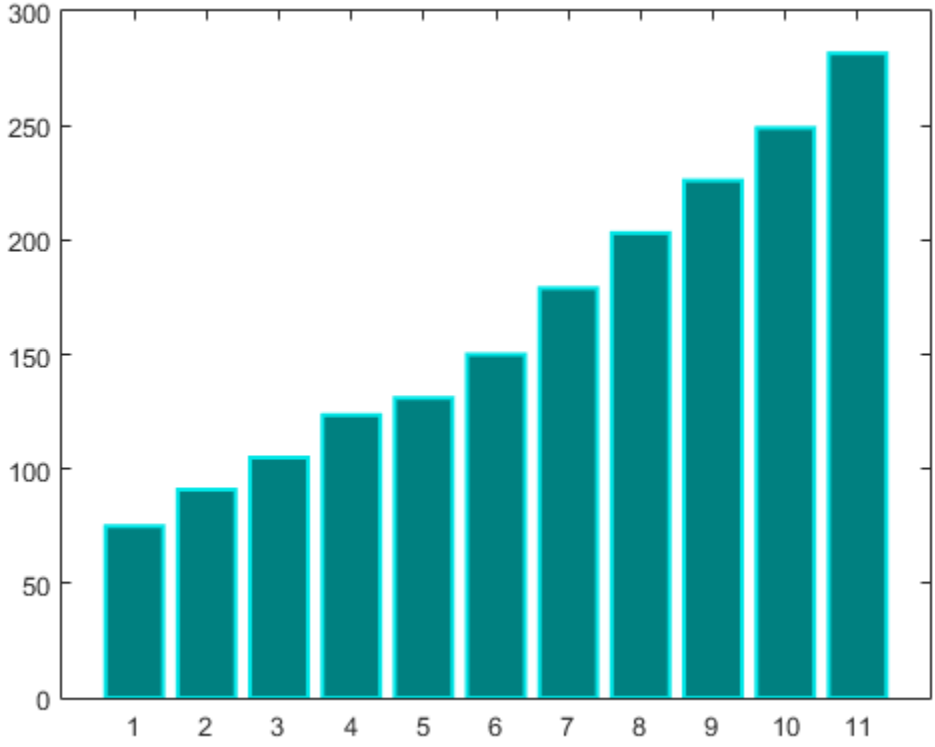
```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y,'r')
```



Specify Bar and Outline Colors

Set the bar interior color and outline color using RGB triplets. Set the width of the bar outline.

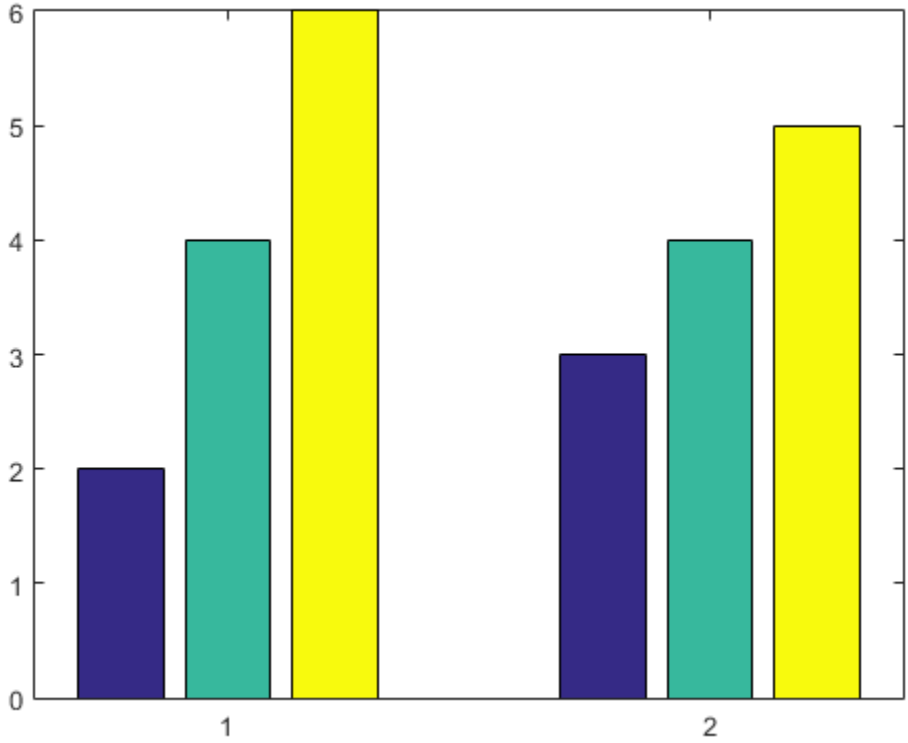
```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y,'FaceColor',[0 .5 .5],'EdgeColor',[0 .9 .9],'LineWidth',1.5)
```



Change Properties for Specific Bar Series

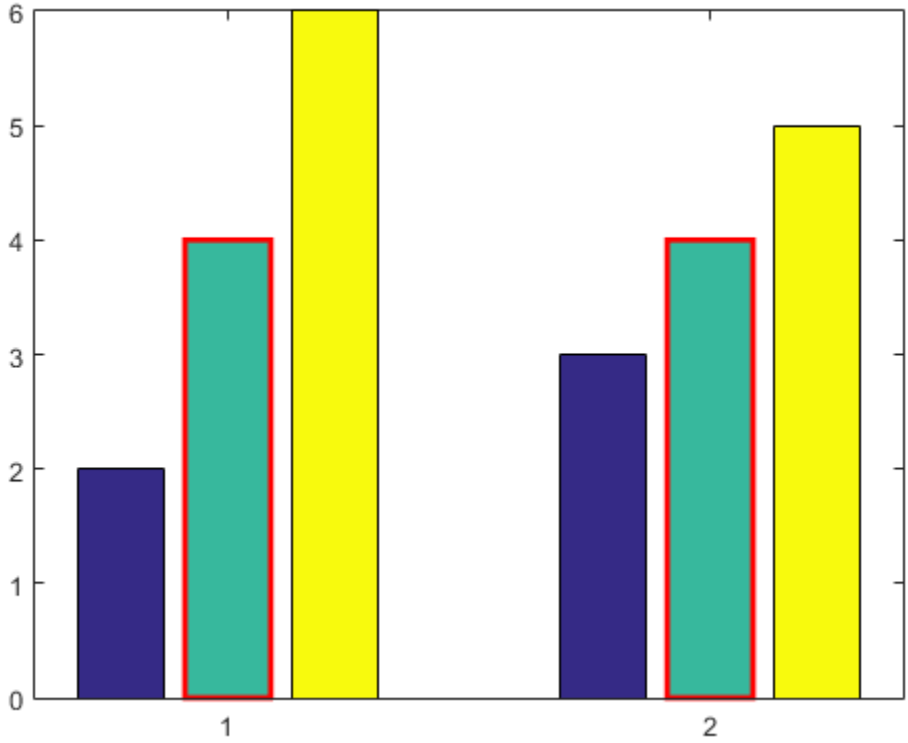
Create a bar graph with a three-column matrix input and return the three bar series objects. `bar` creates one bar series for each column in the matrix.

```
y = [2 4 6; 3 4 5];
b = bar(y);
```



Change properties for a specific bar series by indexing into the object array. For example, change properties of the bars representing the second column of y using `b(2)`. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
b(2).LineWidth = 2;
b(2).EdgeColor = 'red';
```



- “Modify Baseline of Bar Graph”
- “Overlay Bar Graphs”
- “Overlay Line Plot on Bar Graph Using Different y-Axes”

Input Arguments

x — x values

vector | matrix

x values, specified as a vector or a matrix. If `x` and `y` are both vectors, then they must be equal length. If `x` and `y` are both matrices, then they must be equal size. If `x` is a vector and `y` is a matrix, then the length of `x` must equal the number of rows in `y`.

The *x* values do not have to be in order, but they cannot contain duplicate values. If `x` is a matrix, then it cannot contain duplicate values across columns.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`y` — y values

vector | matrix

y values, specified as a vector or a matrix.

- If `y` is a vector, then `bar` draws one bar for each element. The `bar` function treats all vectors as column vectors.
- If `y` is a matrix, then `bar` groups the bars according to the rows in `y`.

Example: `[10 8 5 7 3 9 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`width` — Bar width

`0.8` (default) | scalar

Bar width, specified as a fraction of the total space available for each bar. The default of `0.8` means the bar width is 80% of the space from the previous bar to the next bar, with 10% of that space on each side.

If the width is `1`, then the bars within a group touch one another.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`style` — Bar group style

`'grouped'` (default) | `'stacked'` | `'hist'` | `'histc'`

Bar group style, specified by one of these values.

Style	Purpose
`'grouped'`	Display one group for each row in `y`.

Style	Purpose
	• If `y` is an *m*-by-*n* matrix, then `bar` displays *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in `y`. • If `y` is a vector of length *n*, then `bar` displays one group of *n* bars. The `bar` function treats all vectors as column vectors.
`'stacked'`	Display one bar for each row in `y`. • If `y` is an *m*-by-*n* matrix, then `bar` displays *m* bars where each bar height is the sum of the elements in the row. Each bar is multicolored. Colors correspond to distinct elements and show the relative contribution each row element makes to the total sum. • If `y` is a vector of length *n*, then `bar` displays *n* bars. The `bar` function treats all vectors as column vectors.
`'histc'`	Display the graph in histogram format, in which bars touch one another.
`'hist'`	Display the graph in histogram format, but center each bar over the *x*-ticks, rather than making bars span *x*-ticks as the `histc` option does.

`color` — Bar color

`'b' | 'r' | 'g' | 'c' | 'm' | 'y' | 'k' | 'w'`

Bar color, specified as one of the strings in this table.

String	Color
`'b'`	Blue
`'r'`	Red
`'g'`	Green

String	Color
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

ax — Axes object

axes object

Axes object. If you do not specify an axes, then `bar` uses the current axes for the bar graph.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The bar series properties listed here are only a subset. For a complete list, see Bar Series Properties.

Note: You cannot specify `Name,Value` pairs when using the `'hist'` or `'histc'` bar group style options.

Example: `'EdgeColor','g'` specifies a green outline around the bars.

'EdgeColor' — Bar outline color

`[0 0 0]` (default) | `'flat'` | `'none'` | RGB triplet | color string

Bar outline color, specified as one of these values:

- `'flat'` — Colors based on axes colormap. To change the colormap, use the `colormap` function.
- `'none'` — No color, which makes the outlines invisible.

- RGB triplet or color string — Specify a custom color. The default RGB triplet value of `[0 0 0]` corresponds to black.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`'FaceColor'` — Bar fill color

`'flat'` (default) | `'none'` | RGB triplet | color string

Bar fill color, specified as one of these values:

- `'flat'` — Colors based on the axes colormap.
- `'none'` — No color, which makes the fill invisible.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`

Long Name	Short Name	RGB Triplet
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

'BaseValue' — Baseline location

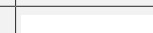
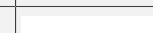
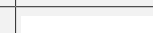
0 (default) | numeric scalar value

Baseline location, specified as a numeric scalar value.

'LineStyle' — Line style of bar outlines

'-' (default) | '--' | ':' | '-.' | 'none'

Line style of bar outlines, specified as one of the strings in this table.

String	Line Style	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

'LineWidth' — Width of bar outlines

0.5 (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: 1.5

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

b — Bar series objects

single object | array of objects

Bar series objects, returned as a single object or an array of objects. Each bar series object comprises a set of bars that have the same color. Use the elements in `b` to access and modify properties of a specific bar series object after it has been created.

More About

Tips

- The default bar colors are based on the axes colormap. You can change the axes colormap using the `colormap` function.

See Also

Functions

`bar3` | `bar3h` | `barh` | `histogram` | `hold` | `stairs`

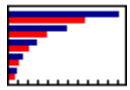
Properties

Bar Series Properties

Introduced before R2006a

barh

Plot bar graph horizontally



Syntax

```
barh(Y)
barh(X,Y)
barh(...,width)
barh(...,'style')
barh(...,'bar_color')
barh(...,'PropertyName',PropertyValue,...)
barh(axes_handle,...)
h = barh(...)
```

Description

A `barh` graph displays the values in a vector or matrix as horizontal bars.

`barh(Y)` draws one horizontal bar for each element in `Y`. If `Y` is a matrix, `barh` groups the bars produced by the elements in each row. The *x*-axis scale ranges from 1 up to `length(Y)` when `Y` is a vector, and `1` to `size(Y,1)`, which is the number of rows, when `Y` is a matrix. The default is to assign an appropriate progression of tick values according to the data. If you want the *x*-axis scale to end exactly at the last bar, set the axis limits as,

```
set(gca,'XLim',[1 length(Y)])
```

`barh(X,Y)` draws a bar for each element in `Y` at locations specified in `x`, where `X` is a vector defining the *x*-axis intervals for the vertical bars. The *x*-values can be

nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `barh` groups the elements of each row in `Y` at corresponding locations in `X`.

`barh(...,width)` sets the relative bar width and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `X`, the bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another. The value of `width` must be a scalar.

`barh(...,'style')` specifies the style of the bars. `'style'` is `'grouped'` or `'stacked'`. Default mode of display is `'grouped'`.

- `'grouped'` displays *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.
- `'histc'` displays the graph in histogram format, in which bars touch one another.
- `'hist'` also displays the graph in histogram format, but centers each bar over the *x*-ticks, rather than making bars span *x*-ticks as the `histc` option does.

Note: When you use either the `hist` or `histc` option, you cannot also use parameter/value syntax. These two options create patch objects rather than bar series.

`barh(...,'bar_color')` displays all bars using the color specified by the single-letter abbreviation `'r'`, `'g'`, `'b'`, `'c'`, `'m'`, `'y'`, `'k'`, or `'w'`.

`barh(...,'PropertyName',PropertyValue,...)` sets the named property or properties to the specified values. You cannot specify properties when `hist` or `histc` options are used. See Bar Series Properties for more information.

`barh(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = barh(...)` returns a vector of bar series handles. When `Y` is a matrix, `barh` creates one bar series per column in `Y`.

Bar Series

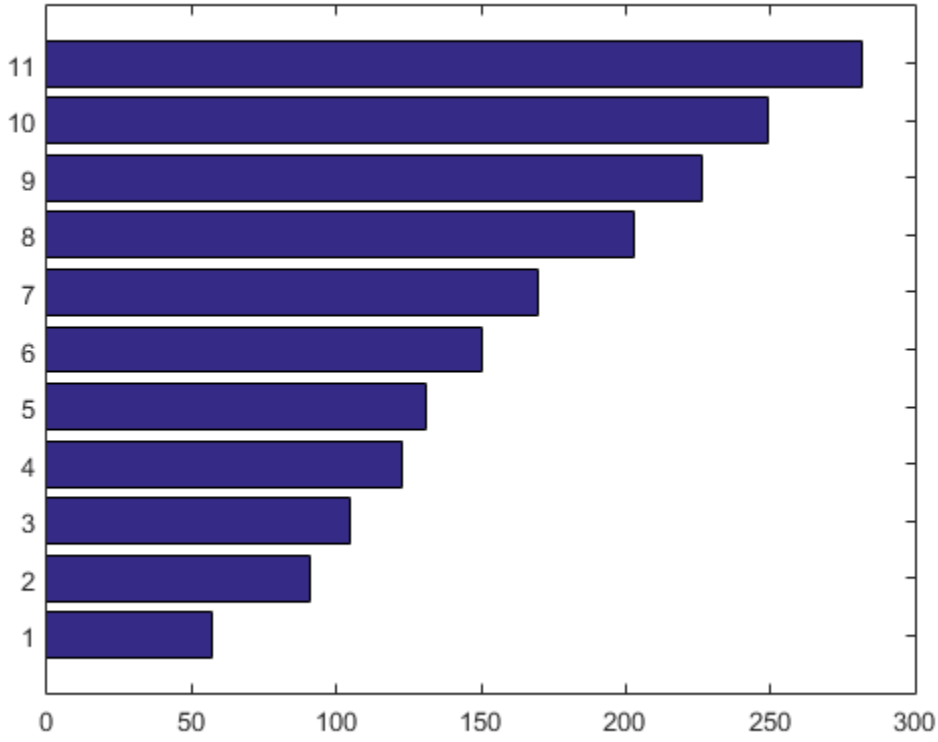
Creating a bar graph of an *m*-by-*n* matrix creates *m* groups of *n* bars. Each bar series comprises a set of bars that have the same color. Use the bar series handle to change properties for all bars in a series.

Examples

Horizontal Bar Graph of Single Data Series

Create a horizontal bar graph of vector data.

```
y = [57,91,105,123,131,150,...
     170,203,226.5,249,281.4];
figure
barh(y)
```

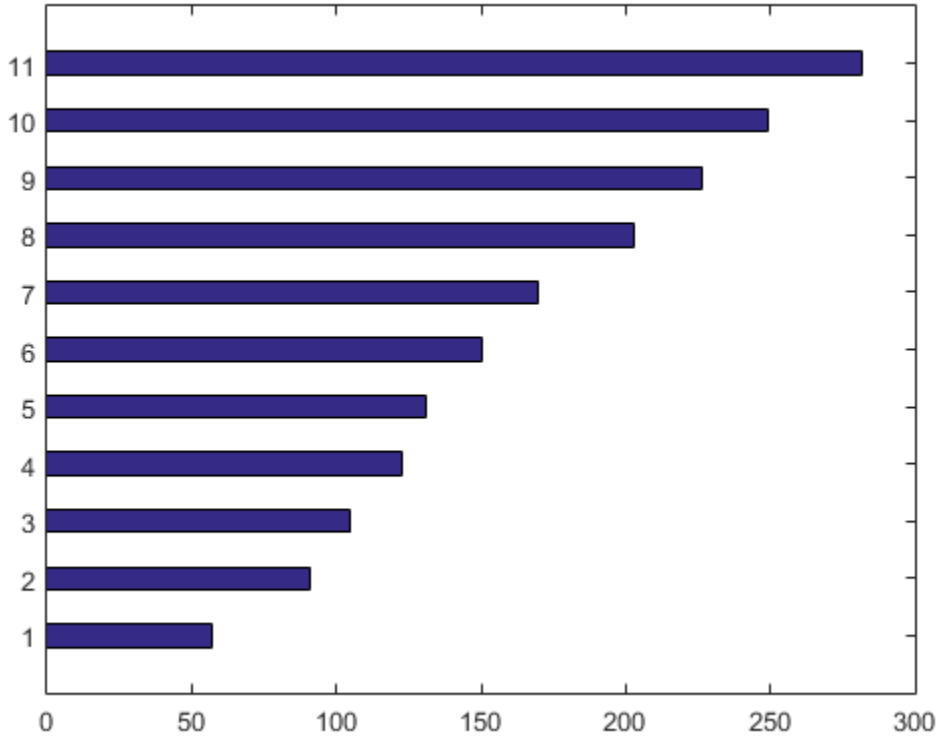


`barh` draws one horizontal bar for each element in `y`.

Specify Width for Horizontal Bar Graph

Specify the bar width to 0.4.

```
y = [57,91,105,123,131,150,...
     170,203,226.5,249,281.4];

figure;
width = 0.4;
barh(y,width);
```

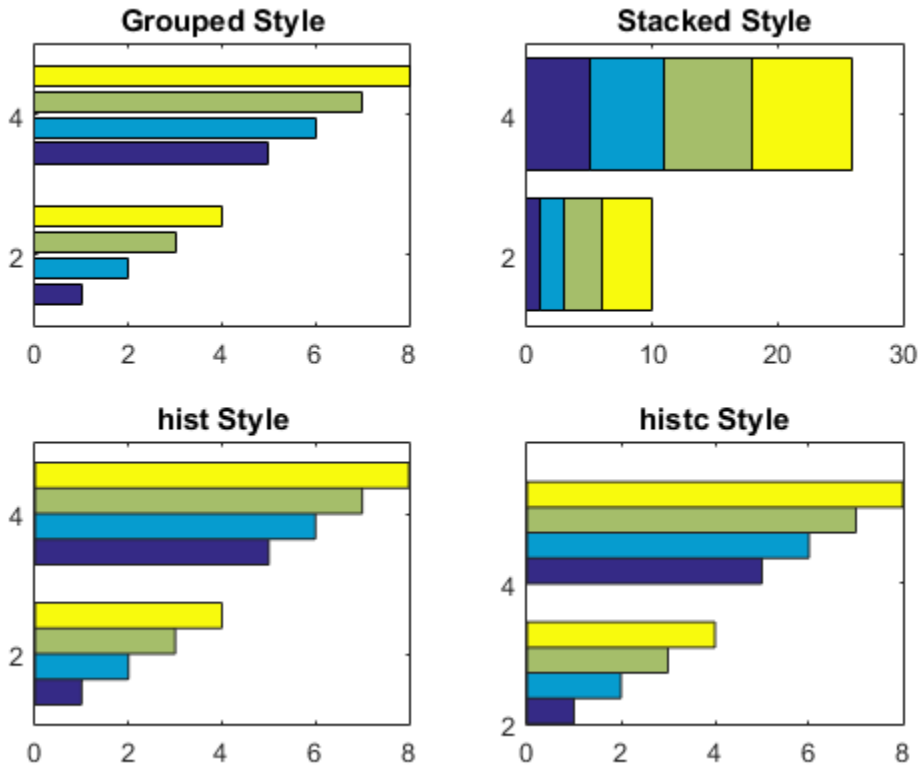


Specify Style for Horizontal Bar Graph

Create a figure with four subplots. In each subplot, create a horizontal bar graph using a different style option for each graph.

```
x = [2,4];
y = [1,2,3,4;...
     5,6,7,8];

figure;
subplot(2,2,1);
barh(x,y,'grouped'); % groups by row
title('Grouped Style')
```

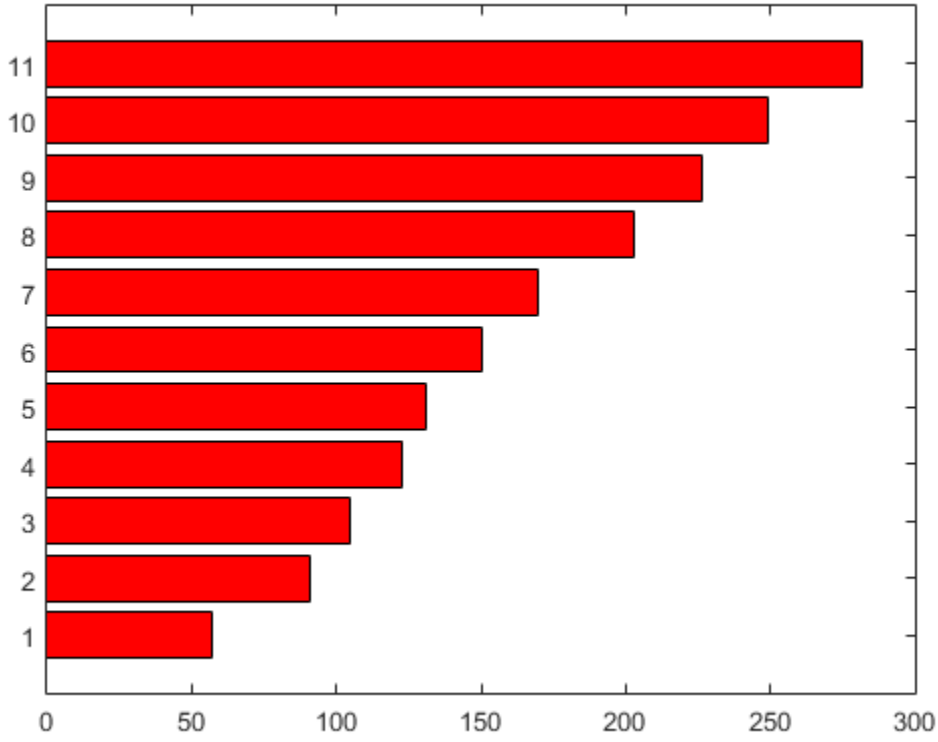
```
subplot(2,2,2);
barh(x,y,'stacked'); % stacks values in each row together
title('Stacked Style')

subplot(2,2,3);
barh(x,y,'hist'); % centers bars over x values
title('hist Style')

subplot(2,2,4);
barh(x,y,'histc'); % spans bars over x values
title('histc Style')
```



Specify Color for Horizontal Bar Graph

Create a horizontal bar graph and change the color of the bars to red.

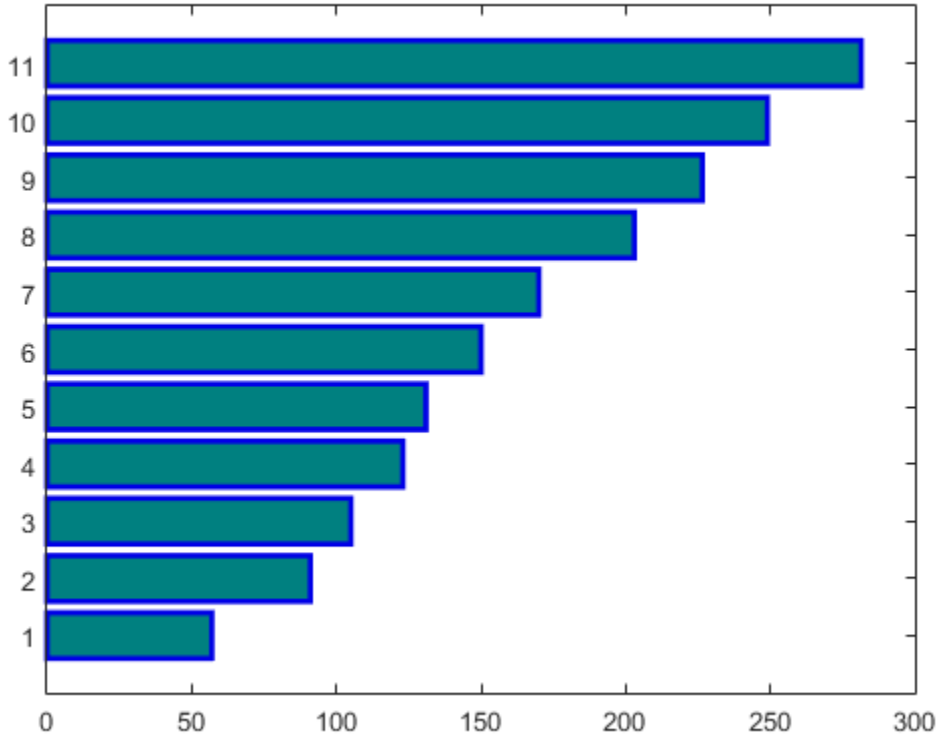
```
y = [57,91,105,123,131,150,...
     170,203,226.5,249,281.4];
figure
barh(y,'r')
```



Specify Bar Properties Using Name-Value Pairs

Create a horizontal bar graph and set the line width to 2. Use RGB triplets to set the face color and edge color for the bars.

```
y = [57,91,105,123,131,150,...
     170,203,226.5,249,281.4];

figure
barh(y,'FaceColor',[0,0.5,0.5],...
       'EdgeColor',[0,0,0.9],...
       'LineWidth',2)
```



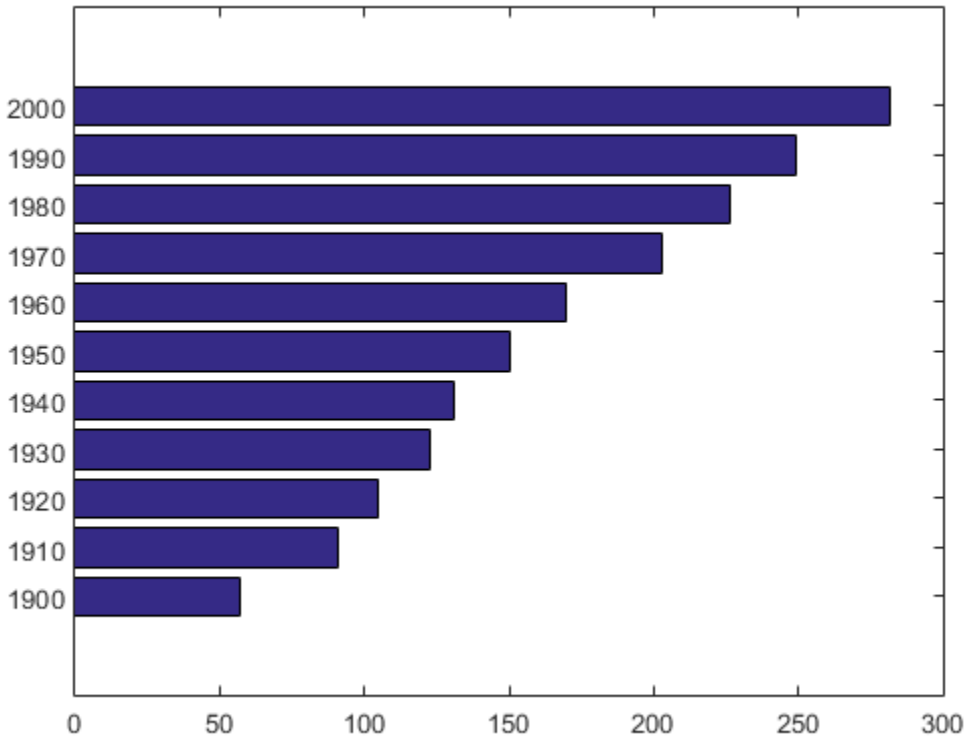
Specify Horizontal Bar Locations

Define `x` and `y` as vectors of data.

```
x = 1900:10:2000;
y = [57,91,105,123,131,150,...
     170,203,226.5,249,281.4];
```

Create a horizontal bar graph of the data in `y`. Use `x` to specify the bar locations along the *y*-axis.

```
figure
barh(x,y)
```



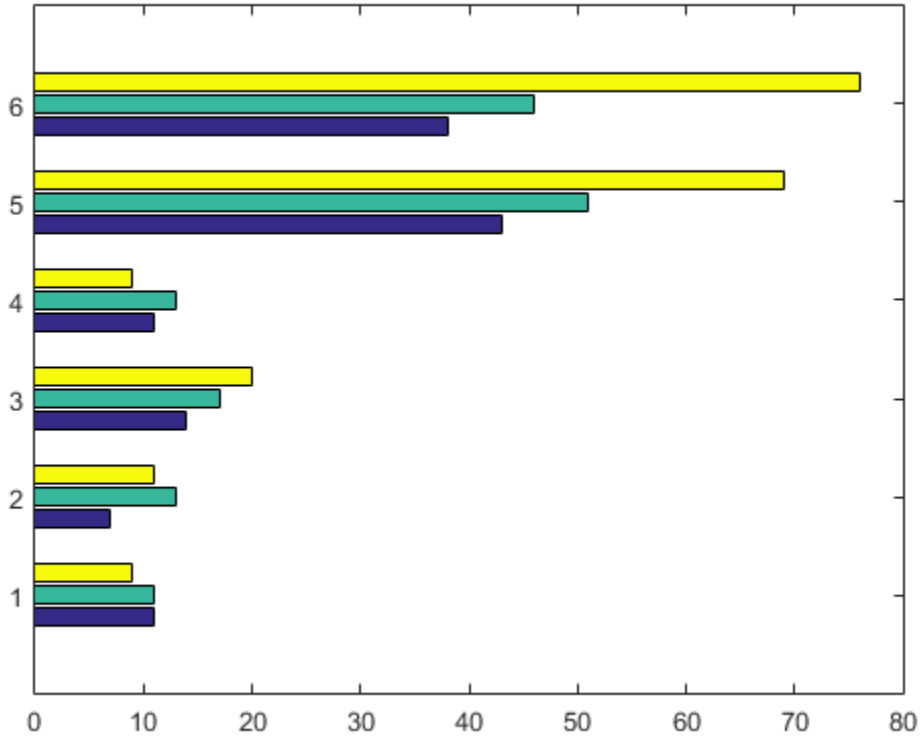
Horizontal Bar Graph of Matrix Data

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `y` as the first six rows of `count`.

```
load count.dat
y = count(1:6,:);
```

Create a horizontal bar graph of matrix `y`.

```
figure
barh(y)
```



By default, `barh` groups the bars by row.

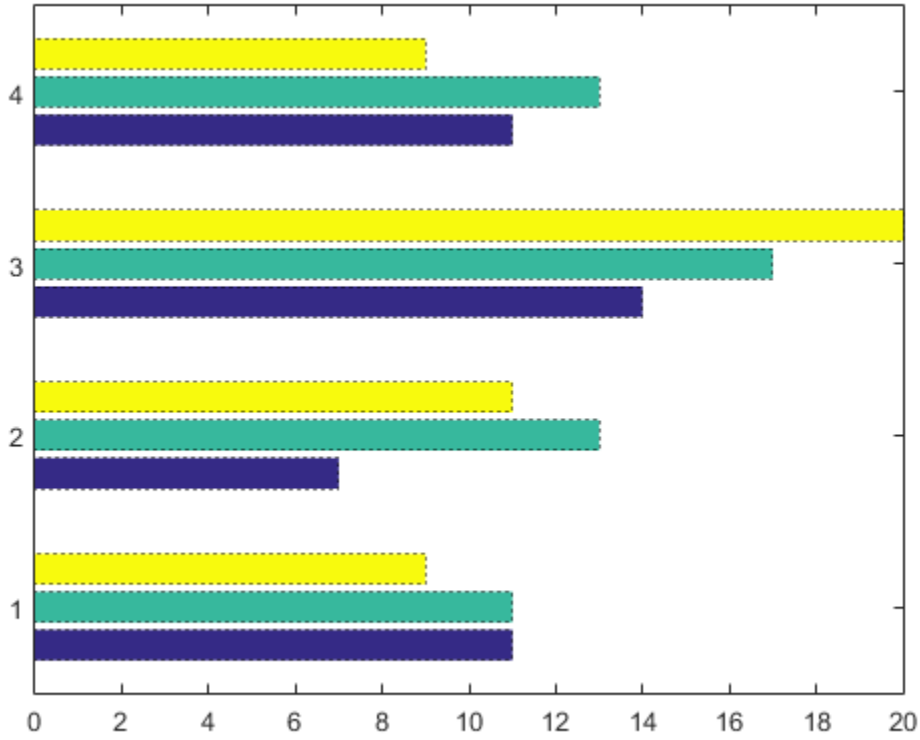
Specify Different Properties For Each Bar Series

Load the data set, `count.dat`, that returns a three-column matrix, `count`. Define `y` as the first four rows of count.

```
load count.dat
y = count(1:4,:);
```

Create a horizontal bar graph of `y` using a dotted line style. Return the three bar series handles. `barh` creates a bar series for each column in `y`.

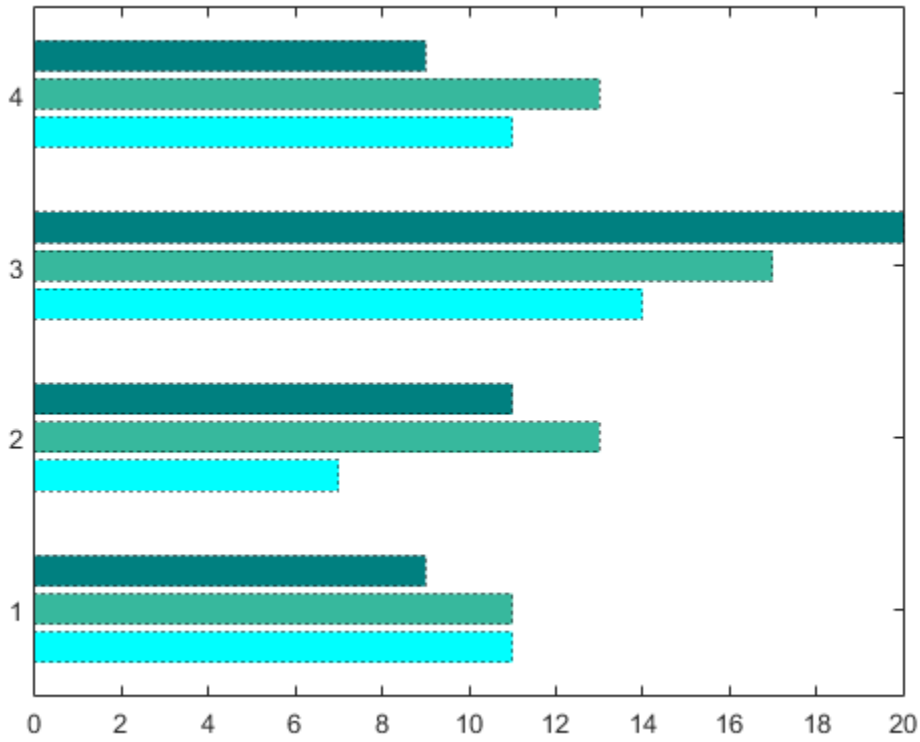
```
h = barh(y,'LineStyle',':');
```



Use the handles in `h` to set different property values for each bar series. Change the face color of the first bar series to cyan by setting the `FaceColor` property to `cyan`. Set the face color for the third bar series using an RGB triplet.

Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
h(1).FaceColor = 'cyan';
h(3).FaceColor = [0,0.5,0.5];
```



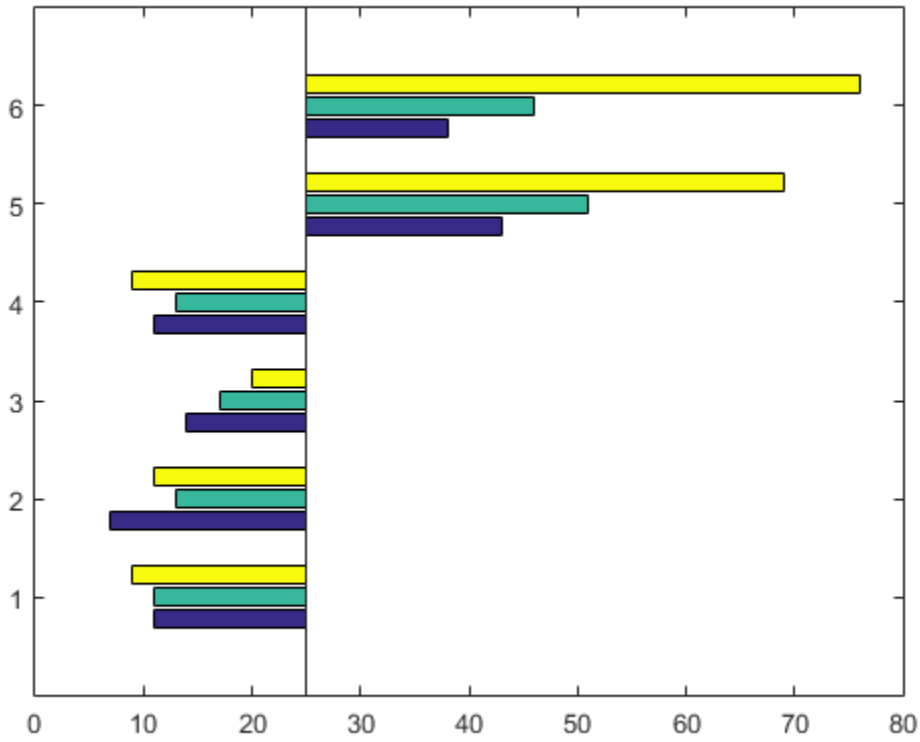
Change Baseline Value for Horizontal Bar Graph

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `y` as the first six rows of `count`.

```
load count.dat
y = count(1:6,:);
```

Create a horizontal bar graph of `y` and set the basevalue to 25.

```
figure
barh(y,'BaseValue',25)
```



More About

- “Modify Baseline of Bar Graph”
- “Overlay Bar Graphs”
- “Overlay Line Plot on Bar Graph Using Different y-Axes”

See Also

Functions

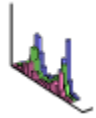
`bar` | `bar3` | `bar3h` | `ColorSpec` | `histogram` | `stairs`

Properties
Bar Series Properties

Introduced before R2006a

bar3

Plot 3-D bar graph



Syntax

```
bar3(Y)
bar3(x,Y)
bar3(...,width)
bar3(...,'style')
bar3(...,LineSpec)
bar3(axes_handle,...)
h = bar3(...)
```

Description

`bar3` draws a three-dimensional bar graph.

`bar3(Y)` draws a three-dimensional bar chart, where each element in `Y` corresponds to one bar. When `Y` is a vector, the *x*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *x*-axis scale ranges from `1` to `size(Y,1)` and the elements in each row are grouped together.

`bar3(x,Y)` draws a bar chart of the elements in `Y` at the locations specified in `x`, where `x` is a vector defining the *y*-axis intervals for vertical bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `bar3` clusters elements from the same row in `Y` at locations corresponding to an element in `x`. Values of elements in each row are grouped together.

`bar3(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar3(...,'style')` specifies the style of the bars. `'style'` is `'detached'`, `'grouped'`, or `'stacked'`. Default mode of display is `'detached'`.

- `'detached'` displays the elements of each row in `Y` as separate blocks behind one another in the *x* direction.
- `'grouped'` displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar3(...,LineSpec)` displays all bars using the color specified by `LineSpec`.

`bar3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = bar3(...)` returns a vector of handles to surface objects, one for each created. When `Y` is a matrix, `bar3` creates one surface object per column in `Y`.

Examples

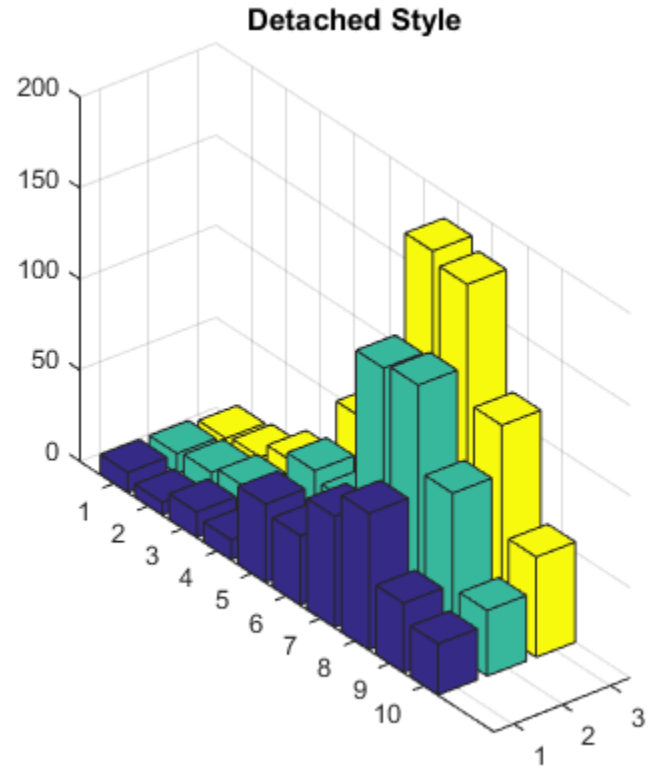
Create 3-D Bar Graph

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first 10 rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D bar graph of `Y`. By default, the style is `detached`.

```
figure
bar3(Y)
title('Detached Style')
```



Specify Bar Width for 3-D Bar Graph

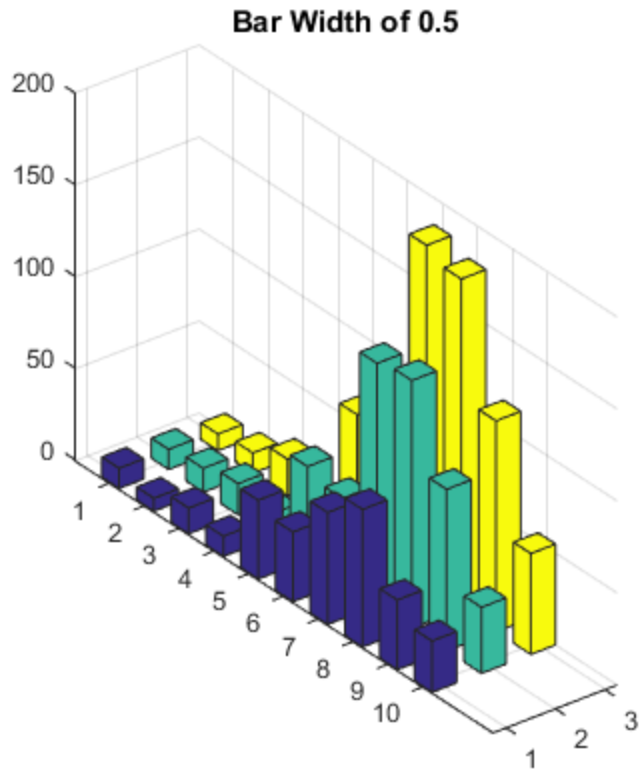
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first 10 rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D bar graph of Y and set the bar width to 0.5.

```
width = 0.5;

figure
bar3(Y,width)
```

```
title('Bar Width of 0.5')
```



3-D Bar Graph with Grouped Style

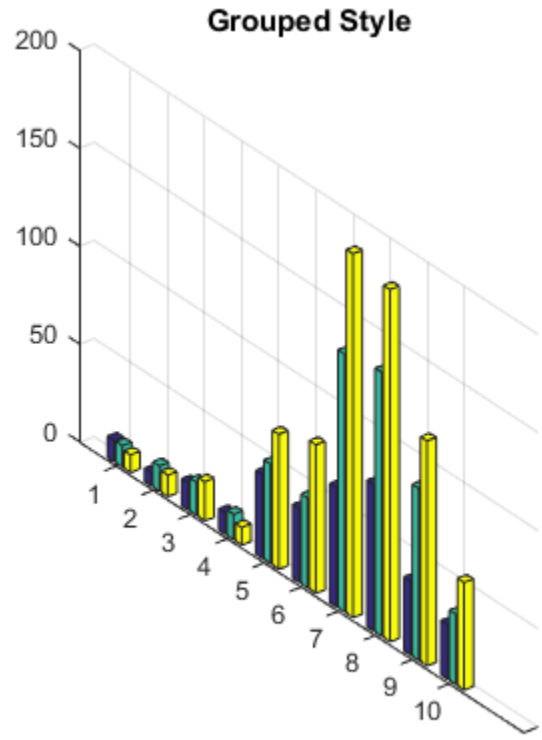
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first 10 rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D bar graph of `Y`. Group the elements in each row of `Y` by specifying the style option as `grouped`.

```
figure
```

```
bar3(Y,'grouped')
title('Grouped Style')
```



3-D Bar Graph with Stacked Style

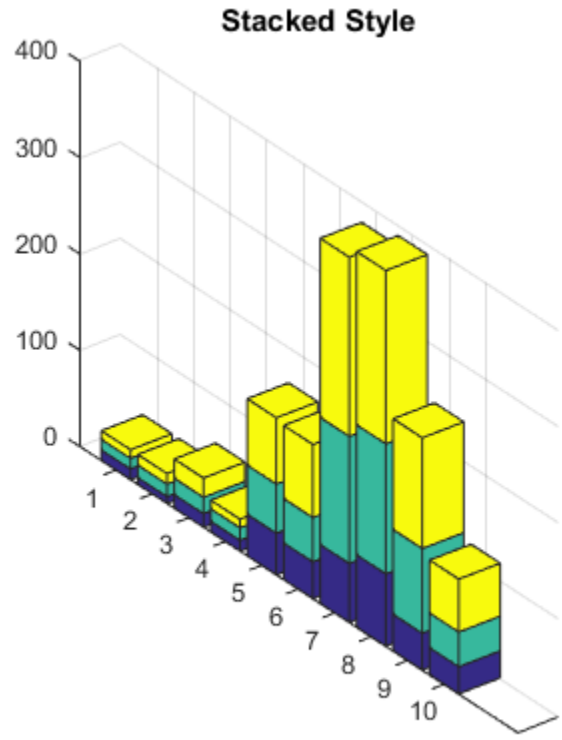
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first 10 rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D bar graph of `Y`. Stack the elements in each row of `Y` by specifying the style option as `stacked`.

```
figure
```

```
bar3(Y,'stacked')
title('Stacked Style')
```



More About

- “Color 3-D Bars by Height”

See Also

`bar` | `barh` | `bar3h` | `LineSpec`

Introduced before R2006a

bar3h

Plot horizontal 3-D bar graph

Syntax

```
bar3h(Y)
bar3h(x,Y)
bar3h(...,width)
bar3h(...,'style')
bar3h(...,LineSpec)
bar3h(axes_handle,...)
h = bar3h(...)
```

Description

`bar3h` draws three-dimensional horizontal bar charts.

`bar3h(Y)` draws a three-dimensional bar chart, where each element in `Y` corresponds to one bar. When `Y` is a vector, the *y*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *y*-axis scale ranges from `1` to `size(Y,1)` and the elements in each row are grouped together.

`bar3h(x,Y)` draws a bar chart of the elements in `Y` at the locations specified in `x`, where `x` is a vector defining the *y*-axis intervals for horizontal bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `bar3` clusters elements from the same row in `Y` at locations corresponding to an element in `x`. Values of elements in each row are grouped together.

`bar3h(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar3h(...,'style')` specifies the style of the bars. `'style'` is `'detached'`, `'grouped'`, or `'stacked'`. Default mode of display is `'detached'`.

- `'detached'` displays the elements of each row in `Y` as separate blocks behind one another in the *y* direction.
- `'grouped'` displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar length is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar3h(...,LineSpec)` displays all bars using the color specified by `LineSpec`.

`bar3h(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = bar3h(...)` returns a vector of handles to surface objects, one for each created. When `Y` is a matrix, `bar3h` creates one surface object per column in `Y`.

Examples

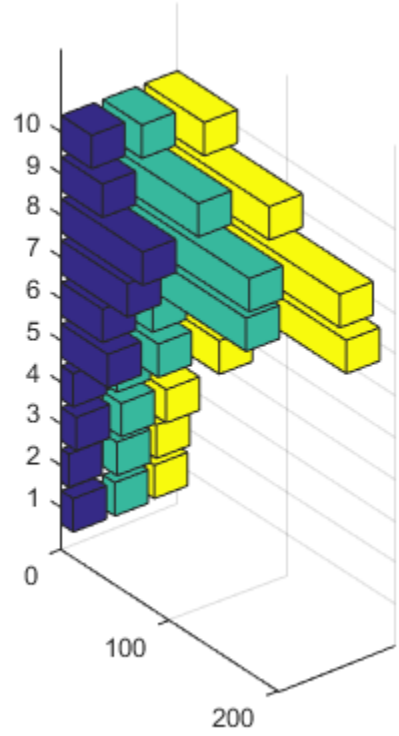
Create 3-D Horizontal Bar Graph

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first ten rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of `Y`. By default, the style is `detached`.

```
figure
bar3h(Y)
```



Specify Bar Width for 3-D Horizontal Bar Graph

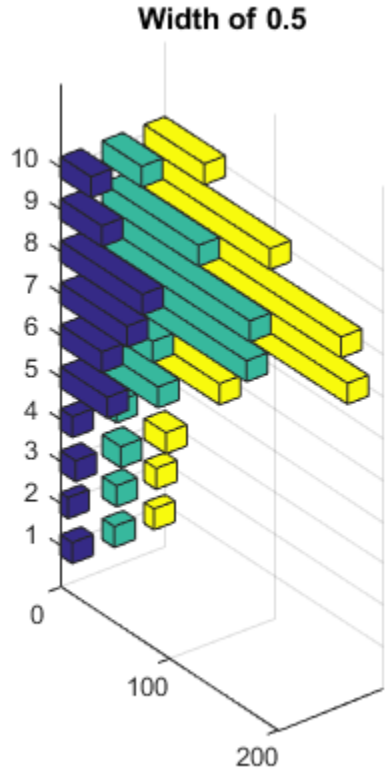
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first ten rows of `count`.

```
load count.dat;
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of `Y` and set the bar width to 0.5.

```
width = 0.5;

figure
bar3h(Y,width)
```

```
title('Width of 0.5')
```



3-D Horizontal Bar Graph with Grouped Style

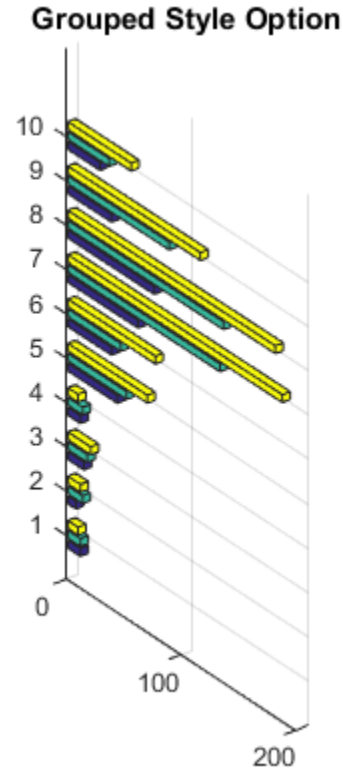
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first ten rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of Y and specify the style option as `grouped`.

```
figure
bar3h(Y,'grouped')
```

```
title('Grouped Style Option')
```



3-D Horizontal Bar Graph with Stacked Option

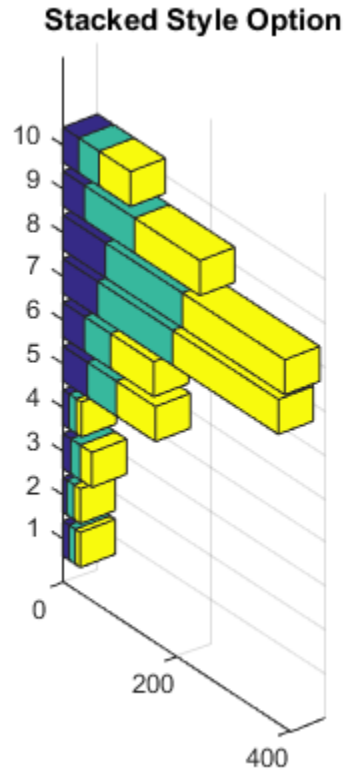
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first ten rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of Y and specify the style option as `stacked`.

```
figure
bar3h(Y,'stacked')
```

```
title('Stacked Style Option')
```



See Also

bar | barh | bar3 | LineSpec | patch

Introduced before R2006a

Bar Series Properties

Control bar series appearance and behavior

Bar series properties control the appearance and behavior of a bar series object. By changing property values, you can modify certain aspects of the bar series.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = bar(1:10);
w = h.BarWidth;
h.BarWidth = 1;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Bars

`BarWidth` — Relative width of individual bars

`0.8` (default) | scalar in range `[0,1]`

Relative width of individual bars, specified as a scalar value in the range `[0,1]`. Use this property to control the separation of bars within a group. The default value is `0.8`, which means that MATLAB separates the bars slightly. If you set this property to `1`, then adjacent bars touch.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`EdgeColor` — Bar outline color

`[0 0 0]` (default) | `'flat'` | `'none'` | RGB triplet | color string

Bar outline color, specified as one of these values:

- `'flat'` — Colors based on axes colormap. To change the colormap, use the `colormap` function.
- `'none'` — No color, which makes the outlines invisible.
- RGB triplet or color string — Specify a custom color. The default RGB triplet value of `[0 0 0]` corresponds to black.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`FaceColor` — Bar fill color

`'flat'` (default) | `'none'` | RGB triplet | color string

Bar fill color, specified as one of these values:

- `'flat'` — Colors based on the axes colormap.
- `'none'` — No color, which makes the fill invisible.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

`LineStyle` — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
'-'	Solid line	————
'--'	Dashed line	- - - - -
':'	Dotted line	.....
'-.'	Dash-dotted line	—·—·—·—
'none'	No line	No line

`LineWidth` — Width of bar outlines

0.5 (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: 1.5

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Bar Graph Type

`BarLayout` — Arrangement of bars

'grouped' (default) | 'stacked'

Arrangement of bars, specified as one of these values:

- `'grouped'` — Group bars by rows in `Y`, where `Y` is the input argument to the `bar` or `barh` function that created the bar chart.
- `'stacked'` — Display one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored. Colors correspond to distinct elements and show the relative contribution each row element makes to the total sum.

`Horizontal` — Horizontal bar chart

`'off'` (default) | `'on'`

Horizontal bar chart, specified as one of these values:

- `'on'` — Display bars horizontally. If you create a graph with `barh`, then the `Horizontal` property is set to `'on'`.
- `'off'` — Display bars vertically. If you create the chart with `bar`, then the `Horizontal` property is set to `'off'`.

Baseline

`BaseLine` — Baseline

baseline object

Baseline object. For a list of baseline properties, see Baseline Properties.

`BaseValue` — Baseline location

`0` (default) | numeric scalar value

Baseline location, specified as a numeric scalar value.

`ShowBaseLine` — Baseline visibility

`'on'` (default) | `'off'`

Baseline visibility, specified as one of these values:

- `'on'` — Show the baseline.
- `'off'` — Hide the baseline.

Data

XData — Bar locations

vector of real values

Bar locations, specified as a vector. For vertical bar charts, the values in `XData` are the bar locations along the *x*-axis. For horizontal bar charts, the values are the bar locations along the *y*-axis. You cannot specify the same value twice.

The input argument `X` to the `bar` and `barh` functions determines the bar locations. If you do not specify `X`, then the indices of the values in `YData` determine the bar locations. `XData` and `YData` must have equal lengths.

Setting this property sets the associated mode property to manual.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

YData — Bar lengths

vector

Bar lengths, specified as a vector. The input argument `Y` to the `bar` and `barh` functions determines the bar lengths.

`XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

XDataSource — Variable linked to XData

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`XDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData` (or `ZData` for 3-D plots).
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

Visibility

`Visible` — Visibility of bar series

`'on'` (default) | `'off'`

Visibility of bar series, specified as one of these values:

- `'on'` — Display the bar series.

- `'off'` — Hide the bar series without deleting it. You still can access the properties of an invisible bar series object.

`Clipping` — Clipping of bar series to axes limits

`'on'` (default) | `'off'`

Clipping of bar series to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the bar series that are outside the axes limits.
- `'off'` — Display the entire bar series, even if parts of it appear outside the axes limits. Parts of the bar series might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the bar series that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'bar'` (default)

Type of graphics object, returned as `'bar'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

`Tag` — Tag to associate with bar series

`''` (default) | string

Tag to associate with the bar series, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

Data Types: `char`

`UserData` — Data to associate with bar series

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the bar series object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the bar series.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the bar series object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the bar series from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the bar series object in the legend as one entry (default).
   - `'off'` — Do not include the bar series object in the legend.
   - `'children'` — Include only children of the bar series object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of bar series

axes object | group object | transform object

Parent of bar series, specified as an axes, group, or transform object.

`Children` — Children of bar series

empty `GraphicsPlaceholder` array

The bar series has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of bar series object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The bar series object handle is always visible.
- `'off'` — The bar series object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The bar series object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the bar series at the command-line, but allows callback functions to access it.

If the bar series object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the bar series. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The bar series object — You can access properties of the bar series object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the bar series. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the bar series when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the bar series.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the bar series that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The HitTest property determines if the bar series responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the bar series passes the click to the object below it in the current view of the figure window. The HitTest property of the bar series has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the bar series. If you have defined the UIContextMenu property, then invoke the context menu.

- `'off'` — Trigger the callbacks for the nearest ancestor of the bar series that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the bar series object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for bar series

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for bar series, specified as one of these values:

- `'off'` — Click the bar series plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the bar series plot to select it, that is, anywhere within the rectangle that encloses the bar series plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the bar series is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the bar series tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.

- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the bar series. Setting the `CreateFcn` property on an existing bar series has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during bar series creation. MATLAB executes the callback after creating the bar series and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The bar series object — You can access properties of the bar series object from within the callback function. You also can access the bar series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the bar series. MATLAB executes the callback before destroying the bar series so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The bar series object — You can access properties of the bar series object from within the callback function. You also can access the bar series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of bar series

`'off'` (default) | `'on'`

Deletion status of bar series, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the bar series begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the bar series no longer exists.

Check the value of the BeingDeleted property to verify that the bar series is not about to be deleted before querying or modifying it.

See Also

`bar` | `barh` | `pareto`

More About

- "Access Property Values"

- “Graphics Object Properties”

baryToCart

Class: TriRep

(Will be removed) Convert point coordinates from barycentric to Cartesian

---

Note: `baryToCart(TriRep)` will be removed in a future release. Use `barycentricToCartesian(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
XC = baryToCart(TR, SI, B)
```

Description

`XC = baryToCart(TR, SI, B)` returns the Cartesian coordinates `XC` of each point in `B` that represents the barycentric coordinates with respect to its associated simplex `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`
`B`	`B` is a matrix that represents the barycentric coordinates of the points to convert with respect to the simplices `SI`. `B` is of size `m`-by-`k`, where `m = length(SI)`, the number of points to convert, and `k` is the number of vertices per simplex.

Output Arguments

`XC`	Matrix of cartesian coordinates of the converted points. `XC` is of size `m`-by-`n`, where `n` is the dimension of the space where the triangulation resides. That is, the Cartesian coordinates of the point `B(j)` with respect to simplex `SI(j)` is `XC(j)`.

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Examples

Compute the Delaunay triangulation of a set of points.

```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
dt = DelaunayTri(x,y)
```

Compute the barycentric coordinates of the incenters.

```
cc = incenters(dt);
tri = dt(:,:);
```

Plot the original triangulation and reference points.

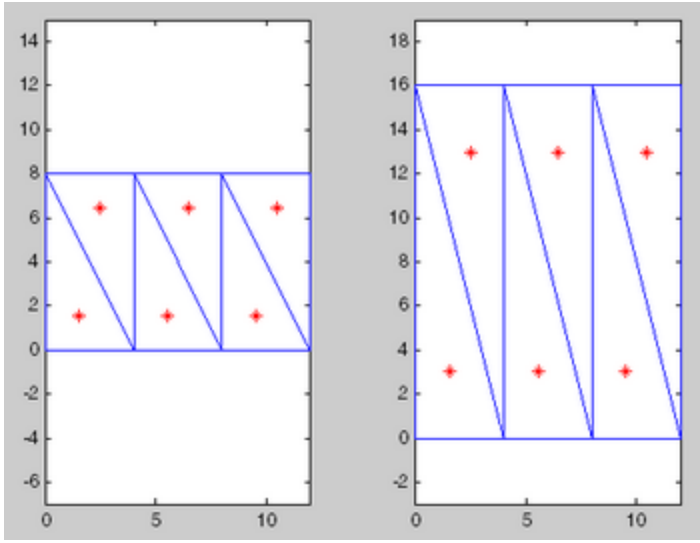
```
figure
subplot(1,2,1);
triplot(dt); hold on;
plot(cc(:,1), cc(:,2), '*r'); hold off;
axis equal;
```

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

```
b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(tr); hold on;
plot(xc(:,1), xc(:,2), '*r'); hold off;
axis equal;
```



See Also

`delaunayTriangulation` | `cartesianToBarycentric` | `pointLocation` | `triangulation`

base2dec

Convert base N number string to decimal number

Syntax

```
d = base2dec('strn', base)
```

Description

`d = base2dec('strn', base)` converts the string number *`strn`* of the specified `base` into its decimal (base 10) equivalent. `base` must be an integer between 2 and 36. *`strn`* must represent a nonnegative integer value. If *`strn`* represents an integer value greater than 2^52, `base2dec` might not return an exact conversion.

If *`strn`* is a character array, each row is interpreted as a string in the specified `base`.

Examples

The expression `base2dec('212',3)` converts 212_3 to decimal, returning 23.

See Also

`dec2base`

Baseline Properties

Control baseline appearance and behavior

Baseline objects are created as part of bar charts, area charts, and stem charts. Baseline properties control the appearance and behavior of a baseline object. By changing property values, you can modify certain aspects of the baseline. Use dot notation to refer to a particular object and property:

```
b = bar(1:10);
bl = b.BaseLine;
c = bl.Color;
bl.Color = 'red';
```

Appearance

`Color` — Line color

RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string, or `'none'`. If you specify the color as `'none'`, then the baseline is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

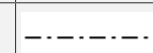
Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — Line style

`'-' | '--' | ':' | '-.' | 'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

positive value

Line width, specified as a positive value in point units.

Example: `0.75`

Location and Visibility

`BaseValue` — Value of baseline

scalar

Value of the baseline, specified as a scalar.

Typically, baselines are associated with bar series, stem series, or area objects. The `BaseValue` property for the associated object and the `BaseValue` property for the baseline object always have the same value. Setting one property also sets the other property. The `BaseLine` property for the associated object contains the baseline object.

Example: `0.75`

`Visible` — Baseline visibility

`'on' | 'off'`

Baseline visibility, specified as one of these values:

- `'on'` — Display the baseline.
- `'off'` — Hide the baseline without deleting it. You can access the properties of an invisible baseline.

Typically, baselines are associated with bar series, stem series, or area objects. The `ShowBaseline` property for the associated object and the `Visible` property for the baseline object always have the same value. Setting one property also sets the other property. The `BaseLine` property for the associated object contains the baseline object.

See Also

`area` | `bar` | `barh` | `stem`

More About

- "Access Property Values"
- "Graphics Object Properties"

beep

Produce operating system beep sound

Syntax

```
beep
beep on
beep off

status = beep
```

Description

`beep` produces your computer's default beep sound, if it is enabled.

`beep on` enables the beep sound.

`beep off` disables the beep sound.

`status = beep` returns the current beep mode (`on` or `off`).

Examples

Produce Beep Sound

Produce your system's default beep sound after a period of silence.

Pause for 5 seconds of silence, and then produce your system's default beep sound.

```
pause(5)
beep
```

More About

Tips

- If you have configured your system not to produce any sound, then `beep` is silent.

- `beep` produces the operating system's default beep sound. To produce a sound and specify its pitch and duration in MATLAB, use the `sound` function.

See Also

sound

Introduced before R2006a

BeginInvoke

Initiate asynchronous .NET delegate call

Syntax

```
result = BeginInvoke(arg1,...,argN,callback,object)
```

Description

`result = BeginInvoke(arg1,...,argN,callback,object)` initiates asynchronous call to a .NET delegate. You must call `EndInvoke` to complete the asynchronous call.

Input Arguments

`arg1,...,argN`

Input arguments for delegate. The type and number of arguments must agree with the delegate signature.

`callback`

.NET System.AsyncCallback delegate, or `[]` null value.

`object`

User-defined object, or `[]` null value.

Output Arguments

`result`

.NET System.IAsyncResult object. Used to monitor the progress of the asynchronous call. Input argument to `EndInvoke`.

More About

- “Calling .NET Methods Asynchronously”
- MSDN Calling Synchronous Methods Asynchronously

See Also

`EndInvoke`

Introduced in R2011a

bench

MATLAB benchmark

Syntax

```
bench
bench(N)
bench(O)
t = bench(N)
```

Description

`bench` times six different MATLAB tasks and compares the execution speed with the speed of several other computers. The six tasks are:

Test	Description	Performance Factors
LU	Perform LU of a full matrix	Floating-point, regular memory access
FFT	Perform FFT of a full vector	Floating-point, irregular memory access
ODE	Solve van der Pol equation with ODE45	Data structures and MATLAB function files
Sparse	Solve a symmetric sparse linear system	Mixed integer and floating-point
2-D	Plot Lissajous curves	2-D line drawing graphics
3-D	Display colormapped `peaks` with clipping and transforms	3-D animated OpenGL graphics

A final bar chart shows speed, which is inversely proportional to time. The longer bars represent faster machines, and the shorter bars represent the slower ones.

`bench(N)` runs each of the six tasks N times.

`bench(O)` just displays the results from other machines.

`t = bench(N)` returns an N-by-6 array with the execution times.

More About

Tips

Note: A benchmark is intended to compare performance of one particular version of MATLAB on different machines. It does not offer direct comparisons between different versions of MATLAB because tasks and problem sizes change from version to version.

The LU and FFT tasks involve large matrices and long vectors. The 2-D and 3-D tasks measure graphics performance, including software or hardware support for OpenGL®. The command

```
opengl info
```

describes the OpenGL support available on a particular machine.

Fluctuations of five or ten percent in the measured times of repeated runs on a single machine are normal.

See Also

`profile` | `mlint` | `profsave` | `mlintrpt` | `memory` | `pack` | `tic` | `cputime` | `rehash`

besselh

Bessel function of third kind (Hankel function)

Syntax

```
H = besselh(nu,K,Z)
H = besselh(nu,Z)
H = besselh(nu,K,Z,1)
```

Definitions

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} + \left(z^2 - \nu^2\right) y = 0,$$

where ν is a real constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*. $J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν. $Y_\nu(z)$ is a second solution of Bessel's equation—linearly independent of $J_\nu(z)$—defined by

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}.$$

The relationship between the Hankel and Bessel functions is

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z),$$

where $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`.

Description

`H = besselh(nu,K,Z)` computes the Hankel function $H_\nu^{(K)}(z)$ where `K` = 1 or 2, for each element of the complex array `Z`. If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, `besselh` expands it to the other input's size.

`H = besselh(nu,Z)` uses `K` = 1.

`H = besselh(nu,K,Z,1)` scales $H_\nu^{(K)}(z)$ by `exp(-i*Z)` if `K` = 1, and by `exp(+i*Z)` if `K` = 2.

Examples

Modulus and Phase of Hankel Function

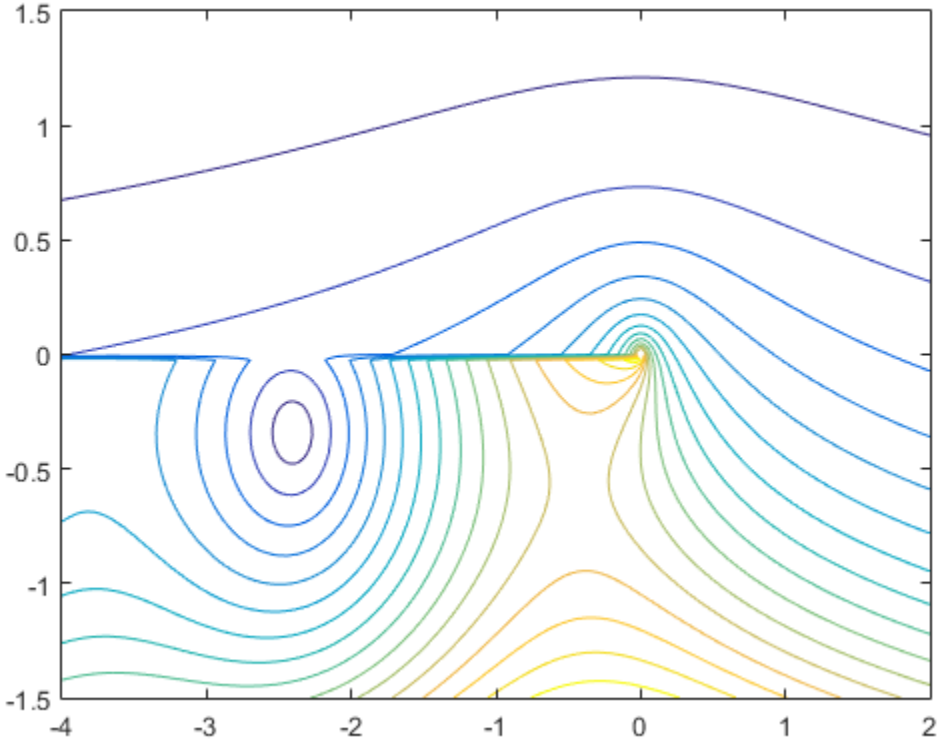
This example generates the contour plots of the modulus and phase of the Hankel function $H_0^{(1)}(z)$ shown on page 359 of Abramowitz and Stegun, *Handbook of Mathematical Functions* [1].

Create a grid of values for the domain.

```
[X,Y] = meshgrid(-4:0.025:2,-1.5:0.025:1.5);
```

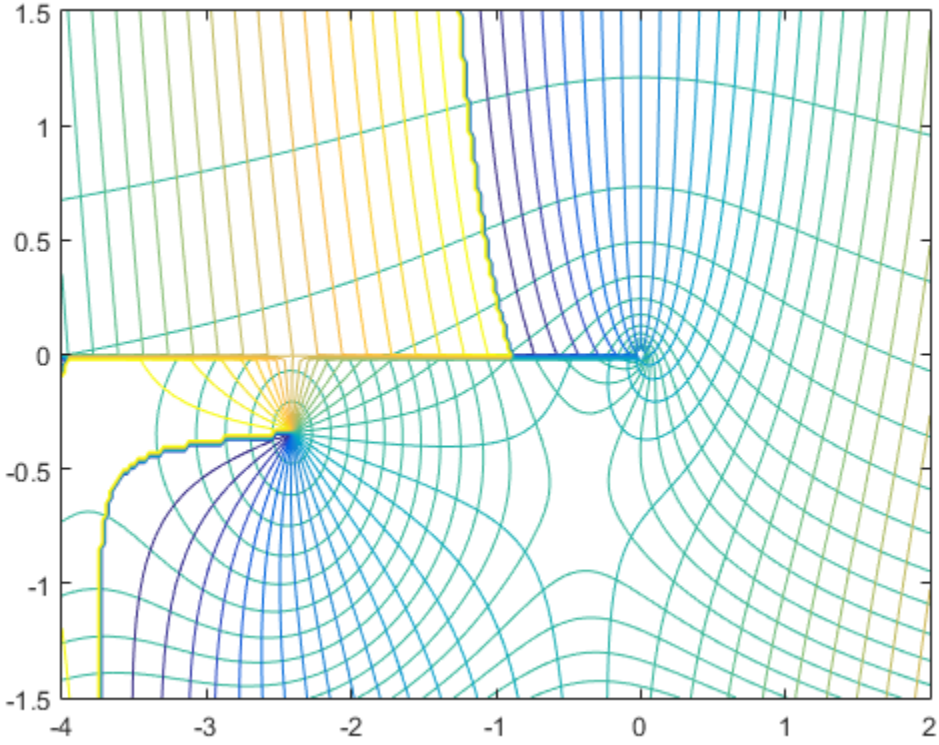
Calculate the Hankel function over this domain and generate the modulus contour plot.

```
H = besselh(0,1,X+1i*Y);
contour(X,Y,abs(H),0:0.2:3.2)
hold on
```



In the same figure, add the contour plot of the phase.

```
contour(X,Y,(180/pi)*angle(H),-180:10:180)
hold off
```



References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965.

See Also

`besselj` | `bessely` | `besseli` | `besselk`

besseli

Modified Bessel function of first kind

Syntax

```
I = besseli(nu,Z)
I = besseli(nu,Z,1)
```

Definitions

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} - \left(z^2 + \nu^2\right)y = 0,$$

where ν is a real constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

$I_\nu(z)$ and $I_{-\nu}(z)$ form a fundamental set of solutions of the modified Bessel's equation. $I_\nu(z)$ is defined by

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{(k=0)}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\Gamma(\nu + k + 1)},$$

where $\Gamma(a)$ is the gamma function.

$K_\nu(z)$ is a second solution, independent of $I_\nu(z)$. It can be computed using `besselk`.

Description

`I = besseli(nu,Z)` computes the modified Bessel function of the first kind, $I_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size.

`I = besseli(nu,Z,1)` computes `besseli(nu,Z).*exp(-abs(real(Z)))`.

Examples

Vector of Function Values

Create a column vector of domain values.

```
z = (0:0.2:1)';
```

Calculate the function values using `besseli` with `nu = 1`.

```
format long
besseli(1,z)
```

```
ans =

                   0
   0.100500834028125
   0.204026755733571
   0.313704025604922
   0.432864802620640
   0.565159103992485
```

Plot Modified Bessel Functions of First Kind

Define the domain.

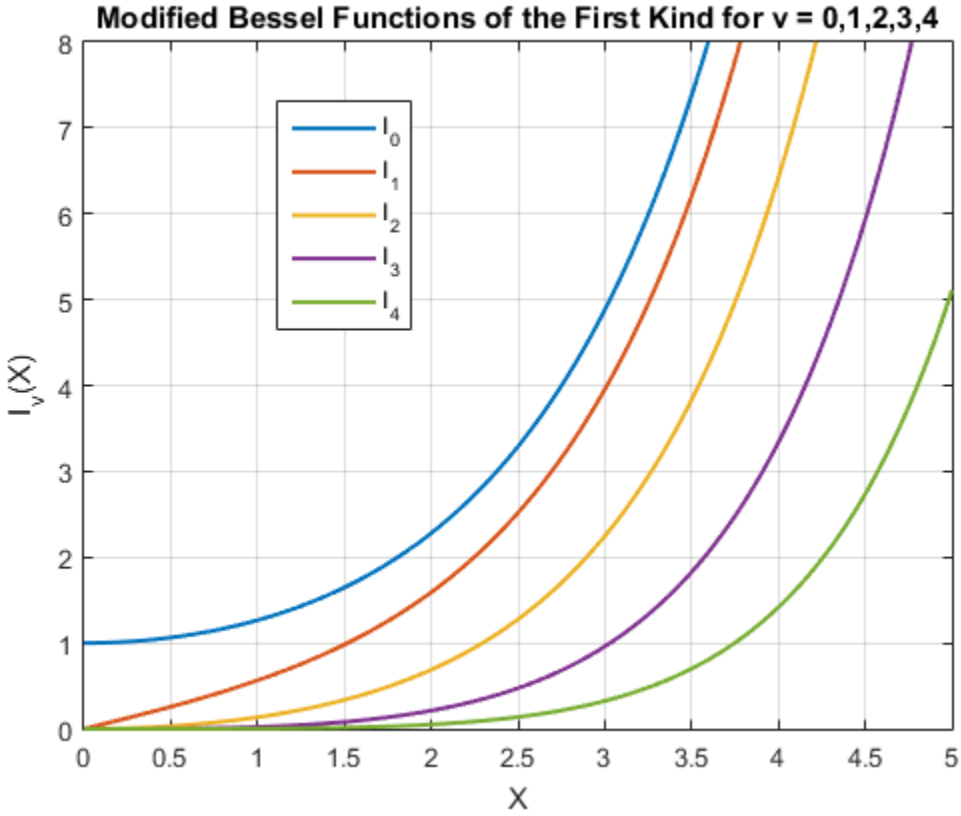
```
X = 0:0.01:5;
```

Calculate the first five modified Bessel functions of the first kind.

```
I = zeros(5,501);
for i = 0:4
    I(i+1,:) = besseli(i,X);
end
```

Plot the results.

```
plot(X,I,'LineWidth',1.5)
axis([0 5 0 8])
grid on
legend('I_0','I_1','I_2','I_3','I_4','Location','Best')
title('Modified Bessel Functions of the First Kind for v = 0,1,2,3,4')
xlabel('X')
ylabel('I_v(X)')
```



See Also

airy | besselh | besselj | besselk | bessely

besselj

Bessel function of first kind

Syntax

```
J = besselj(nu,Z)
J = besselj(nu,Z,1)
```

Definitions

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} + \left(z^2 - v^2\right)y = 0,$$

where v is a real constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*.

$J_v(z)$ and $J_{-v}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger v. $J_v(z)$ is defined by

$$J_v(z) = \left(\frac{z}{2}\right)^v \sum_{(k=0)}^{\infty} \frac{\left(\frac{-z^2}{4}\right)^k}{k!\Gamma(v+k+1)}$$

where $\Gamma(a)$ is the gamma function.

$Y_v(z)$ is a second solution of Bessel's equation that is linearly independent of $J_v(z)$. It can be computed using `bessely`.

Description

`J = besselj(nu,Z)` computes the Bessel function of the first kind, $J_v(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size.

`J = besselj(nu,Z,1)` computes `besselj(nu,Z).*exp(-abs(imag(Z)))`.

Examples

Vector of Function Values

Create a column vector of domain values.

```
z = (0:0.2:1)';
```

Calculate the function values using `besselj` with `nu = 1`.

```
format long
besselj(1,z)
```

```
ans =

                   0
   0.099500832639236
   0.196026577955319
   0.286700988063916
   0.368842046094170
   0.440050585744934
```

Plot Bessel Functions of First Kind

Define the domain.

```
X = 0:0.1:20;
```

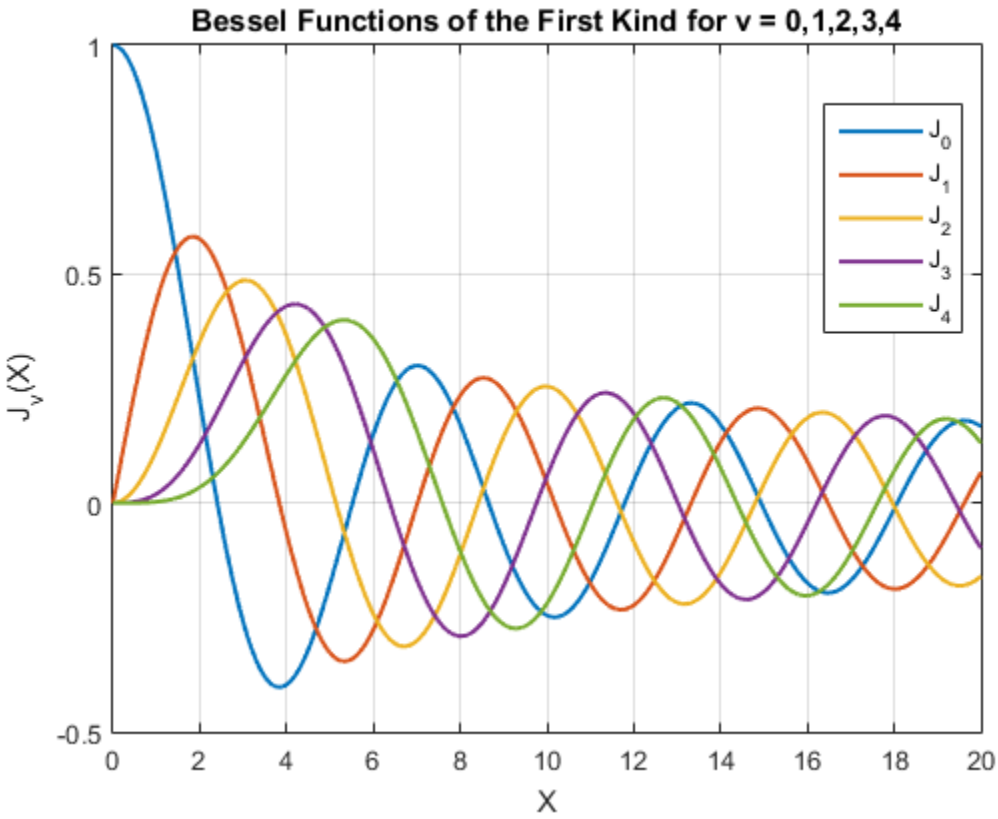
Calculate the first five Bessel functions of the first kind.

```
J = zeros(5,201);
for i = 0:4
    J(i+1,:) = besselj(i,X);
end
```

Plot the results.

```
plot(X,J,'LineWidth',1.5)
axis([0 20 -.5 1])
```

```
grid on
legend('J_0','J_1','J_2','J_3','J_4','Location','Best')
title('Bessel Functions of the First Kind for v = 0,1,2,3,4')
xlabel('X')
ylabel('J_v(X)')
```



More About

Tips

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind, by the formula

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z)$$

where $H_\nu^{(K)}(z)$ is `besselh`, $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

See Also

`besselh` | `besseli` | `besselk` | `bessely`

besselk

Modified Bessel function of second kind

Syntax

```
K = besselk(nu,Z)
K = besselk(nu,Z,1)
```

Definitions

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} - \left(z^2 + \nu^2\right) y = 0,$$

where ν is a real constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

A solution $K_\nu(z)$ of the second kind can be expressed as:

$$K_\nu(z) = \left(\frac{\pi}{2}\right)\frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)},$$

where $I_\nu(z)$ and $I_{-\nu}(z)$ form a fundamental set of solutions of the modified Bessel's equation,

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\Gamma(\nu + k + 1)}$$

and $\Gamma(a)$ is the gamma function. $K_\nu(z)$ is independent of $I_\nu(z)$.

$I_\nu(z)$ can be computed using `besseli`.

Description

`K = besselk(nu,Z)` computes the modified Bessel function of the second kind, $K_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size.

`K = besselk(nu,Z,1)` computes `besselk(nu,Z).*exp(Z)`.

Examples

Column Vector of Function Values

Create a column vector of domain values.

```
z = (0:0.2:1)';
```

Calculate the function values using `besselk` with `nu = 1`.

```
format long
besselk(1,z)
```

```
ans =

                  Inf
   4.775972543220472
   2.184354424732687
   1.302834939763502
   0.861781634472180
   0.601907230197235
```

Plot Modified Bessel Functions of Second Kind

Define the domain.

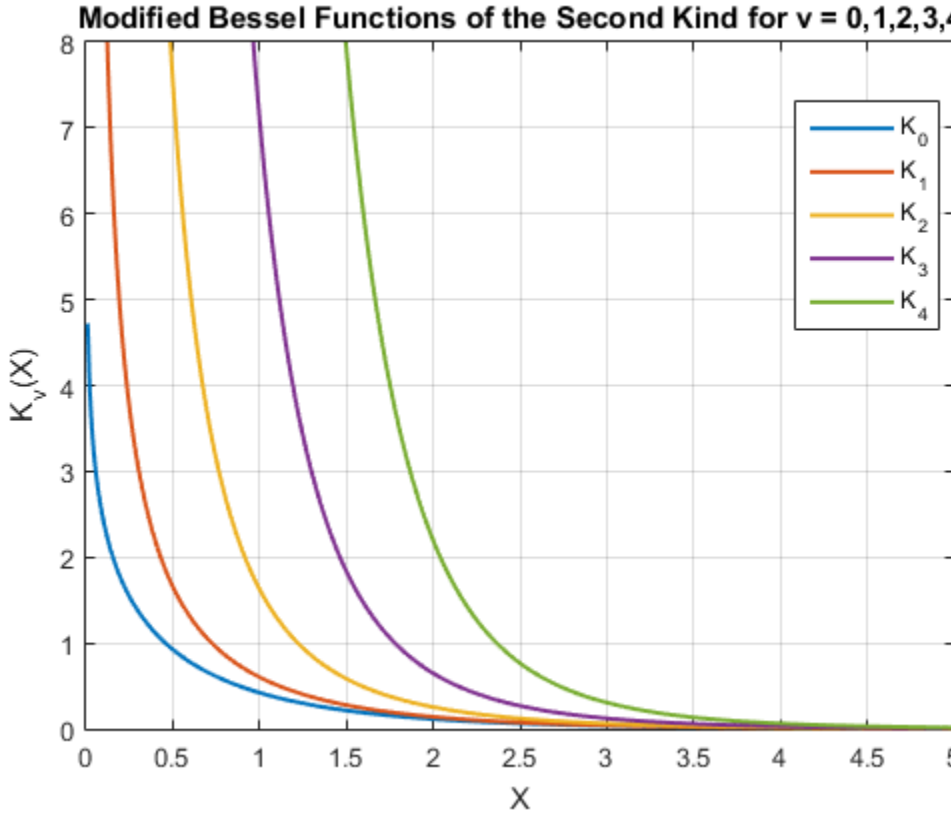
```
X = 0:0.01:5;
```

Calculate the first five modified Bessel functions of the second kind.

```
K = zeros(5,501);
for i = 0:4
    K(i+1,:) = besselk(i,X);
end
```

Plot the results.

```
plot(X,K,'LineWidth',1.5)
axis([0 5 0 8])
grid on
legend('K_0','K_1','K_2','K_3','K_4','Location','Best')
title('Modified Bessel Functions of the Second Kind for v = 0,1,2,3,4')
xlabel('X')
ylabel('K_v(X)')
```



See Also

`airy` | `besselh` | `besseli` | `besselj` | `bessely`

bessely

Bessel function of second kind

Syntax

```
Y = bessely(nu,Z)
Y = bessely(nu,Z,1)
```

Definitions

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} + \left(z^2 - \nu^2\right)y = 0,$$

where ν is a real constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*.

A solution $Y_\nu(z)$ of the second kind can be expressed as

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

where $J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{z^2}{4}\right)^k}{k!\Gamma(\nu + k + 1)},$$

and $\Gamma(a)$ is the gamma function. $Y_\nu(z)$ is linearly independent of $J_\nu(z)$.

$J_v(z)$ can be computed using `besselj`.

Description

`Y = bessely(nu,Z)` computes Bessel functions of the second kind, $Y_v(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size.

`Y = bessely(nu,Z,1)` computes `bessely(nu,Z).*exp(-abs(imag(Z)))`.

Examples

Vector of Function Values

Create a column vector of domain values.

```
z = (0:0.2:1)';
```

Calculate the function values using `bessely` with `nu = 1`.

```
format long
bessely(1,z)
```

```
ans =

                -Inf
  -3.323824988111848
  -1.780872044270052
  -1.260391347177388
  -0.978144176683359
  -0.781212821300289
```

Plot Bessel Functions of Second Kind

Define the domain.

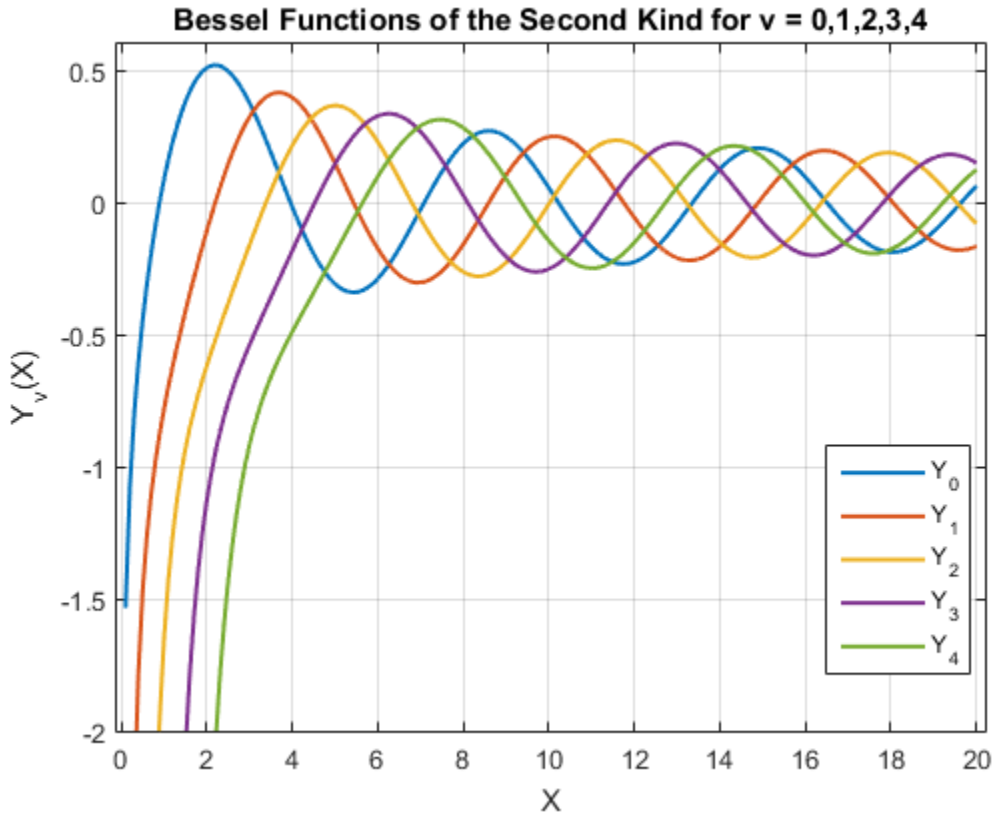
```
X = 0:0.1:20;
```

Calculate the first five Bessel functions of the second kind.

```
Y = zeros(5,201);
for i = 0:4
    Y(i+1,:) = bessely(i,X);
end
```

Plot the results.

```
plot(X,Y,'LineWidth',1.5)
axis([-0.1 20.2 -2 0.6])
grid on
legend('Y_0','Y_1','Y_2','Y_3','Y_4','Location','Best')
title('Bessel Functions of the Second Kind for v = 0,1,2,3,4')
xlabel('X')
ylabel('Y_v(X)')
```



More About

Tips

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind,

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z),$$

where $H_\nu^{(K)}(z)$ is `besselh`, $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

See Also

`besselh` | `besseli` | `besselj` | `besselk`

beta

Beta function

Syntax

```
B = beta(Z,W)
```

Definitions

The beta function is

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1}dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

where $\Gamma(z)$ is the gamma function.

Description

`B = beta(Z,W)` computes the beta function for corresponding elements of arrays `Z` and `W`. The arrays must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

In this example, which uses integer arguments,

```
beta(n,3)
   = (n-1)!*2!/(n+2)!
   = 2/(n*(n+1)*(n+2))
```

is the ratio of fairly small integers, and the rational format is able to recover the exact result.

```
format rat
```

```
beta((0:10)',3)

ans =

     1/0
     1/3
     1/12
     1/30
     1/60
     1/105
     1/168
     1/252
     1/360
     1/495
     1/660
```

More About

Algorithms

```
beta(z,w) = exp(gammaln(z)+gammaln(w)-gammaln(z+w))
```

See Also

`betainc` | `betaln` | `gammaln`

Introduced before R2006a

betainc

Incomplete beta function

Syntax

```
I = betainc(X,Z,W)
I = betainc(X,Z,W,tail)
```

Definitions

The incomplete beta function is

$$I_x(z,w) = \frac{1}{B(z,w)} \int_0^x t^{z-1}(1-t)^{w-1} dt$$

where $B(z,w)$, the beta function, is defined as

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1} dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

and $\Gamma(z)$ is the gamma function.

Description

`I = betainc(X,Z,W)` computes the incomplete beta function for corresponding elements of the arrays `X`, `Z`, and `W`. The elements of `X` must be in the closed interval [0,1]. The arrays `Z` and `W` must be nonnegative and real. All arrays must be the same size, or any of them can be scalar.

`I = betainc(X,Z,W,tail)` specifies the tail of the incomplete beta function. Choices are:

`'lower'` (the default)	Computes the integral from `0` to `x`

`'upper'`	Computes the integral from `x` to `1`

These functions are related as follows:

`1-betainc(X,Z,W) = betainc(X,Z,W,'upper')`
Note that especially when the upper tail value is close to `0`, it is more accurate to use the `'upper'` option than to subtract the `'lower'` value from `1`.

Examples

```
format long
betainc(.5,(0:10)',3)

ans =
   1.00000000000000
   0.87500000000000
   0.68750000000000
   0.50000000000000
   0.34375000000000
   0.22656250000000
   0.14453125000000
   0.08984375000000
   0.05468750000000
   0.03271484375000
   0.01928710937500
```

See Also

`beta` | `betaln`

Introduced before R2006a

betaincinv

Beta inverse cumulative distribution function

Syntax

```
x = betaincinv(y,z,w)
x = betaincinv(y,z,w,tail)
```

Description

`x = betaincinv(y,z,w)` computes the inverse incomplete beta function for corresponding elements of `y`, `z`, and `w`, such that `y = betainc(x,z,w)`. The elements of `y` must be in the closed interval [0,1], and those of `z` and `w` must be nonnegative. `y`, `z`, and `w` must all be real and the same size (or any of them can be scalar).

`x = betaincinv(y,z,w,tail)` specifies the tail of the incomplete beta function. Choices are `'lower'` (the default) to use the integral from 0 to `x`, or `'upper'` to use the integral from `x` to 1. These two choices are related as follows: `betaincinv(y,z,w,'upper') = betaincinv(1-y,z,w,'lower')`. When `y` is close to 0, the `'upper'` option provides a way to compute `x` more accurately than by subtracting `y` from 1.

Definitions

The incomplete beta function is defined as

$$I_x(z,w) = \frac{1}{\beta(z,w)} \int_0^x t^{(z-1)}(1-t)^{(w-1)}\,dt$$

`betaincinv` computes the inverse of the incomplete beta function with respect to the integration limit `x` using Newton's method.

See Also

`betainc` | `beta` | `betaln`

betaln

Logarithm of beta function

Syntax

```
L = betaln(Z,W)
```

Description

`L = betaln(Z,W)` computes the natural logarithm of the `beta` function `log(beta(Z,W))`, for corresponding elements of arrays `Z` and `W`, without computing `beta(Z,W)`. Since the beta function can range over very large or very small values, its logarithm is sometimes more useful.

`Z` and `W` must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

```
x = 510
betaln(x,x)

ans =
      -708.8616
```

`-708.8616` is slightly less than `log(realmin)`. Computing `beta(x,x)` directly would underflow (or be denormal).

More About

Algorithms

```
betaln(z,w) = gammaln(z)+gammaln(w)-gammaln(z+w)
```

See Also

`beta` | `betainc` | `gammaln`

Introduced before R2006a

between

Calendar math differences

Syntax

```
dt = between(t1,t2)
dt = between(t1,t2,components)
```

Description

`dt = between(t1,t2)` returns the differences between the datetime values in `t1` and `t2`. The `dt` output is a `calendarDuration` array in terms of the calendar components years, months, days, and time, such that `t2 = t1+dt`.

`dt = between(t1,t2,components)` returns the differences between datetime values in terms of the specified calendar or time components.

Examples

Differences Between Two Datetime Arrays

Create two `datetime` arrays.

```
t1 = datetime('now')
```

```
t1 =

   23-Feb-2015 10:06:51
```

```
t2 = datetime('tomorrow','Format','dd-MMM-yyyy HH:mm:ss') + caldays(0:2)
```

```
t2 =

   24-Feb-2015 00:00:00   25-Feb-2015 00:00:00   26-Feb-2015 00:00:00
```

Find the difference between the two arrays.

```
dt = between(t1,t2)
```

```
dt =

      13h 53m 8.551s   1d 13h 53m 8.551s   2d 13h 53m 8.551s
```

`between` returns a `calendarDuration` array containing differences in terms of days, hours, minutes, and seconds.

Difference Between Datetime Values in Calendar Days

Create a sequence of datetimes over a 6-month period. Then, find the number of days between the first date and each of the dates in the sequence.

```
t1 = datetime(2013,1,1);
t2 = dateshift(t1,'end','month',0:4)
```

```
t2 =

   31-Jan-2013   28-Feb-2013   31-Mar-2013   30-Apr-2013   31-May-2013
```

```
dt = between(t1,t2,'Days')
```

```
dt =

    30d    58d    89d   119d   150d
```

Input Arguments

`t1` — Input date and time

`datetime` array | datetime string | cell array of datetime strings

Input date and time, specified as a `datetime` array, datetime string, or cell array of datetime strings. At least one of inputs `t1` and `t2` must be a `datetime` array. `t1` and `t2` must be the same size unless one is a scalar.

t2 — Input date and time

datetime array | datetime string | cell array of datetime strings

Input date and time, specified as a datetime array, datetime string, or cell array of datetime strings. At least one of inputs t1 and t2 must be a datetime array. t1 and t2 must be the same size unless one is a scalar.

components — Calendar or time components

'years' | 'quarters' | 'months' | 'weeks' | 'days' | 'time' | cell array of strings

Calendar or time components, specified as one of the following strings, or a cell array containing one ore more of these strings:

- 'years'
- 'quarters'
- 'months'
- 'weeks'
- 'days'
- 'time'

Except for 'time', the above components are flexible lengths of time. For example, 1 month represents a different length of time when added to a datetime in January than when added to a datetime in February.

between operates on the calendar or time components in decreasing order, starting with the largest component.

In general, t2 is not equal to t1 + dt, unless you include 'time' in components.

Example: {'years','quarters'}

Data Types: char | cell

Output Arguments

dt — Difference array

calendarDuration array

Difference array, returned as a calendarDuration array.

More About

Tips

- To compute differences between datetime values in `t1` and `t2` as exact, fixed-length durations, use `t2-t1`.

See Also

`caldiff` | `calendarDuration` | `diff` | `minus`

Introduced in R2014b

bicg

Biconjugate gradients method

Syntax

```
x = bicg(A,b)
bicg(A,b,tol)
bicg(A,b,tol,maxit)
bicg(A,b,tol,maxit,M)
bicg(A,b,tol,maxit,M1,M2)
bicg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicg(A,b,...)
[x,flag,relres] = bicg(A,b,...)
[x,flag,relres,iter] = bicg(A,b,...)
[x,flag,relres,iter,resvec] = bicg(A,b,...)
```

Description

`x = bicg(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicg` converges, it displays a message to that effect. If `bicg` fails to converge after the maximum number of iterations or halts for any reason, it prints a warning message that includes the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicg` uses the default, `1e-6`.

`bicg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicg` uses the default, `min(n,20)`.

`bicg(A,b,tol,maxit,M)` and `bicg(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicg` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`bicg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `bicg` uses the default, an all-zero vector.

`[x,flag] = bicg(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`bicg` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicg` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicg` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicg` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = bicg(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicg(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = bicg(A,b,...)` also returns a vector of the residual norms at each iteration including `norm(b-A*x0)`.

Examples

Using bicg with a Matrix Input

This example shows how to use `bicg` with a matrix input. `bicg`. The following code:

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = bicg(A,b,tol,maxit,M1,M2);
```

displays this message:

```
bicg converged at iteration 9 to a solution with relative
residual 5.3e-009
```

Using bicg with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_bicg` that

- Calls `bicg` with the `@afun` function handle as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_bicg` are available to `afun`.

Place the following into a file called `run_bicg`:

```
function x1 = run_bicg
n = 100;
on = ones(n,1);
b = afun(on,'notransp');
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = bicg(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
```

```
        elseif strcmp(transp_flag,'notransp') % y = A*x
           y = 4 * x;
           y(2:n) = y(2:n) - 2 * x(1:n-1);
           y(1:n-1) = y(1:n-1) - x(2:n);
        end
    end
end
```

When you enter

```
x1 = run_bicg;
```

MATLAB software displays the message

```
bicg converged at iteration 9 to a solution with ...
relative residual
5.3e-009
```

Using bicg with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

Use `bicg` to find a solution at the requested tolerance and number of iterations.

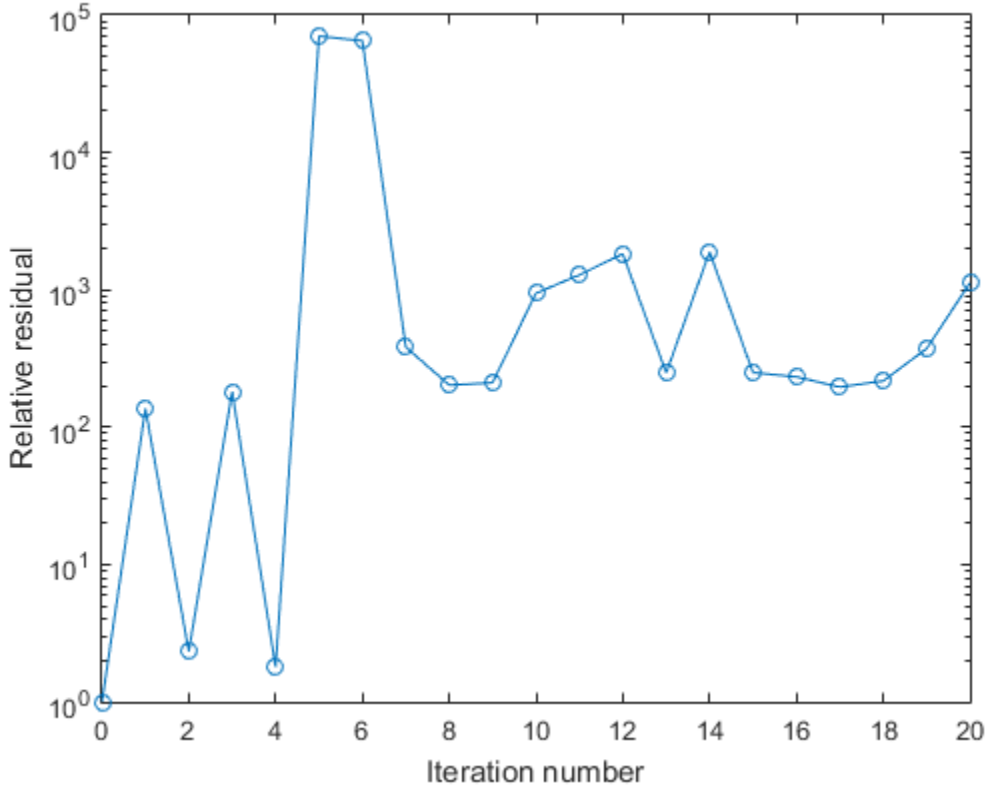
```
[x0,fl0,rr0,it0,rv0] = bicg(A,b,tol,maxit);
```

`fl0` is 1 because `bicg` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicg` is so poor that the initial guess (`x0`

`= zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `bicg`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));

Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

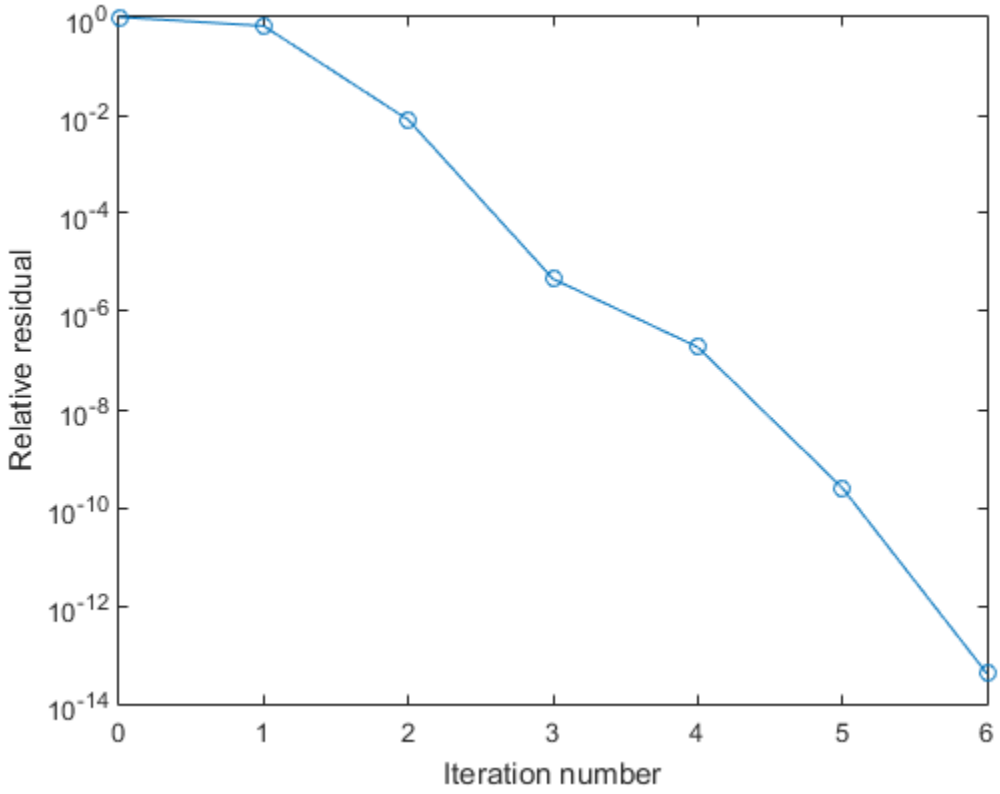
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicg(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicg` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

You can follow the progress of `bicg` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

See Also

`bicgstab` | `cgs` | `function_handle` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Introduced before R2006a

bicgstab

Biconjugate gradients stabilized method

Syntax

```
x = bicgstab(A,b)
bicgstab(A,b,tol)
bicgstab(A,b,tol,maxit)
bicgstab(A,b,tol,maxit,M)
bicgstab(A,b,tol,maxit,M1,M2)
bicgstab(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstab(A,b,...)
[x,flag,relres] = bicgstab(A,b,...)
[x,flag,relres,iter] = bicgstab(A,b,...)
[x,flag,relres,iter,resvec] = bicgstab(A,b,...)
```

Description

`x = bicgstab(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicgstab` converges, a message to that effect is displayed. If `bicgstab` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicgstab(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicgstab` uses the default, `1e-6`.

`bicgstab(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicgstab` uses the default, `min(n,20)`.

`bicgstab(A,b,tol,maxit,M)` and `bicgstab(A,b,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicgstab` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x)` returns `M\x`.

`bicgstab(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `bicgstab` uses the default, an all zero vector.

`[x,flag] = bicgstab(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`bicgstab` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicgstab` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicgstab` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicgstab` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = bicgstab(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicgstab(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`. `iter` can be an integer + 0.5, indicating convergence halfway through an iteration.

`[x,flag,relres,iter,resvec] = bicgstab(A,b,...)` also returns a vector of the residual norms at each half iteration, including `norm(b-A*x0)`.

Examples

Using bicgstab with a Matrix Input

This example first solves `Ax = b` by providing `A` and the preconditioner `M1` directly as arguments.

The code:

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = bicgstab(A,b,tol,maxit,M1);
```

displays the message:

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 2e-014.
```

Using bicgstab with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in a file `run_bicgstab` that

- Calls `bicgstab` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_bicgstab` are available to `afun` and `mfun`.

The following shows the code for `run_bicgstab`:

```
function x1 = run_bicgstab
n = 21;
b = afun(ones(n,1));
tol = 1e-12;
maxit = 15;
x1 = bicgstab(@afun,b,tol,maxit,@mfun);

    function y = afun(x)
       y = [0; x(1:n-1)] + ...
           [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
           [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
```

```
    end
end
```

When you enter

```
x1 = run_bicgstab;
```

MATLAB software displays the message

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 2e-014.
```

Using bicgstab with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

Use `bicgstab` to find a solution at the requested tolerance and number of iterations.

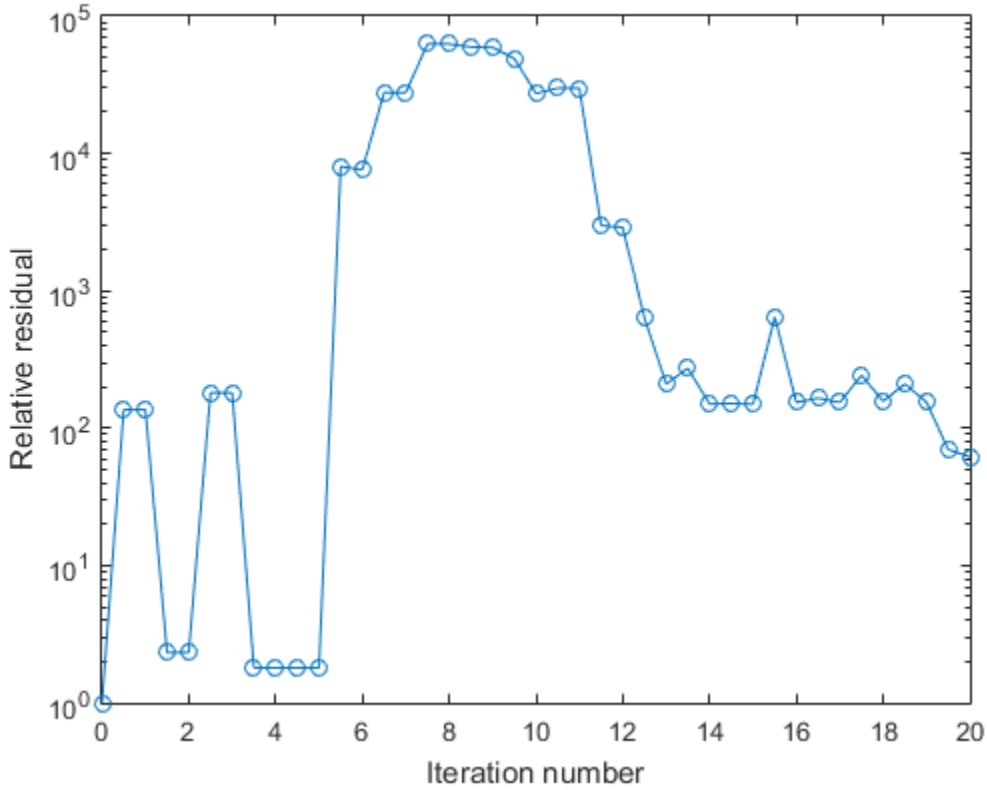
```
[x0,fl0,rr0,it0,rv0] = bicgstab(A,b,tol,maxit);
```

`fl0` is 1 because `bicgstab` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicgstab` is so bad that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `bicgstab`.

```
semilogy(0:0.5:maxit,rv0/norm(b),'-o');
```

```
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create a preconditioner with `ilu`, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

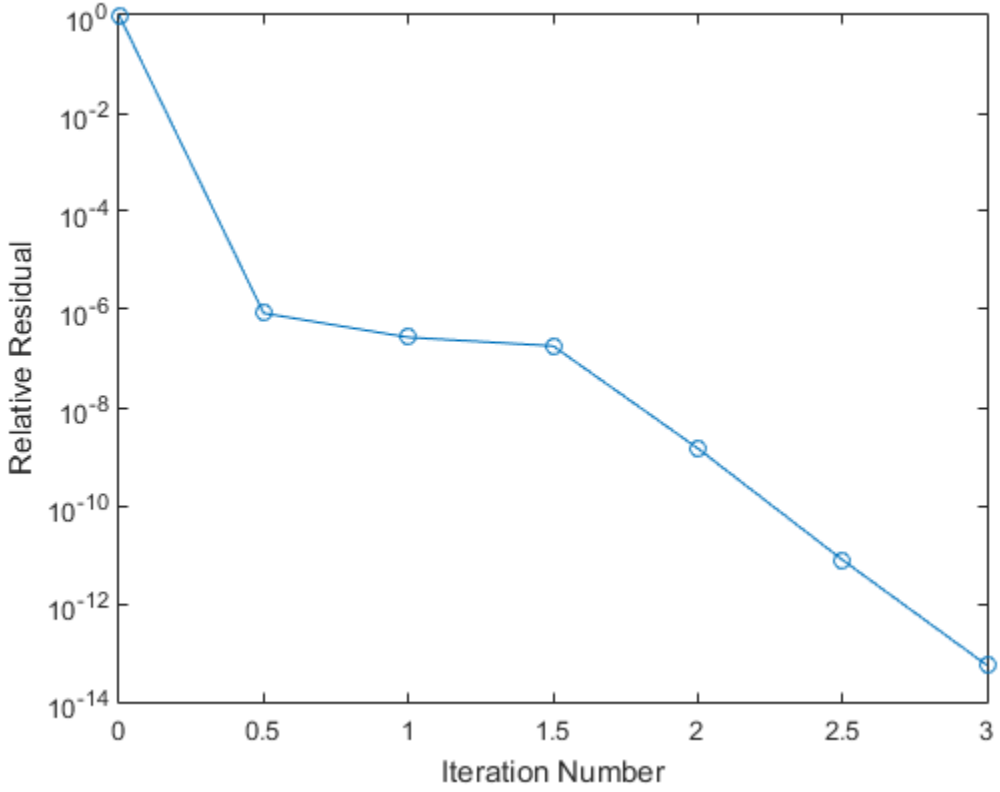
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicgstab(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicgstab` drives the relative residual to `5.9829e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the third iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)` and the output `rv1(7)` is `norm(b-A*x2)` since `bicgstab` uses half iterations.

You can follow the progress of `bicgstab` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:0.5:it1,rv1/norm(b),'-o');
xlabel('Iteration Number');
ylabel('Relative Residual');
```



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] van der Vorst, H.A., "BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, March 1992, Vol. 13, No. 2, pp. 631–644.

See Also

`bicg` | `cgs` | `function_handle` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Introduced before R2006a

bicgstabl

Biconjugate gradients stabilized (l) method

Syntax

```
x = bicgstabl(A,b)
x = bicgstabl(afun,b)
x = bicgstabl(A,b,tol)
x = bicgstabl(A,b,tol,maxit)
x = bicgstabl(A,b,tol,maxit,M)
x = bicgstabl(A,b,tol,maxit,M1,M2)
x = bicgstabl(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstabl(A,b,...)
[x,flag,relres] = bicgstabl(A,b,...)
[x,flag,relres,iter] = bicgstabl(A,b,...)
[x,flag,relres,iter,resvec] = bicgstabl(A,b,...)
```

Description

`x = bicgstabl(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the right-hand side column vector `b` must have length `n`.

`x = bicgstabl(afun,b)` accepts a function handle `afun` instead of the matrix `A`. `afun(x)` accepts a vector input `x` and returns the matrix-vector product `A*x`. In all of the following syntaxes, you can replace `A` by `afun`.

`x = bicgstabl(A,b,tol)` specifies the tolerance of the method. If `tol` is [] then `bicgstabl` uses the default, 1e-6.

`x = bicgstabl(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is [] then `bicgstabl` uses the default, `min(N,20)`.

`x = bicgstabl(A,b,tol,maxit,M)` and `x = bicgstabl(A,b,tol,maxit,M1,M2)` use preconditioner `M` or `M=M1*M2` and effectively solve the system `A*inv(M)*x = b` for

x. If `M` is [] then a preconditioner is not applied. `M` may be a function handle returning `M\x`.

`x = bicgstabl(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is [] then `bicgstabl` uses the default, an all zero vector.

`[x,flag] = bicgstabl(A,b,...)` also returns a convergence `flag`:

Flag	Convergence
0	`bicgstabl` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicgstabl` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicgstabl` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicgstabl` became too small or too large to continue computing.

`[x,flag,relres] = bicgstabl(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicgstabl(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`. `iter` can be `k/4` where `k` is some integer, indicating convergence at a given quarter iteration.

`[x,flag,relres,iter,resvec] = bicgstabl(A,b,...)` also returns a vector of the residual norms at each quarter iteration, including `norm(b-A*x0)`.

Examples

Using bicgstabl with Inputs or with a Function

You can pass inputs directly to `bicgstabl`:

```
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
```

```
tol = 1e-12;
maxit = 15;
M = diag([10:-1:1 1 1:10]);
x = bicgstabl(A,b,tol,maxit,M);
```

You can also use a matrix-vector product function:

```
function y = afun(x,n)
y = [0; x(1:n-1)] + [((n-1)/2:-1:0)';
(1:(n-1)/2)'].*x+[x(2:n); 0];
```

and a preconditioner backsolve function:

```
function y = mfun(r,n)
y = r ./ [((n-1)/2:-1:1)';
1;
(1:(n-1)/2)'];
```

as inputs to `bicgstabl`:

```
x1 = bicgstabl(@(x)afun(x,n),b,tol,maxit,@(x)mfun(x,n));
```

Using bicgstabl with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

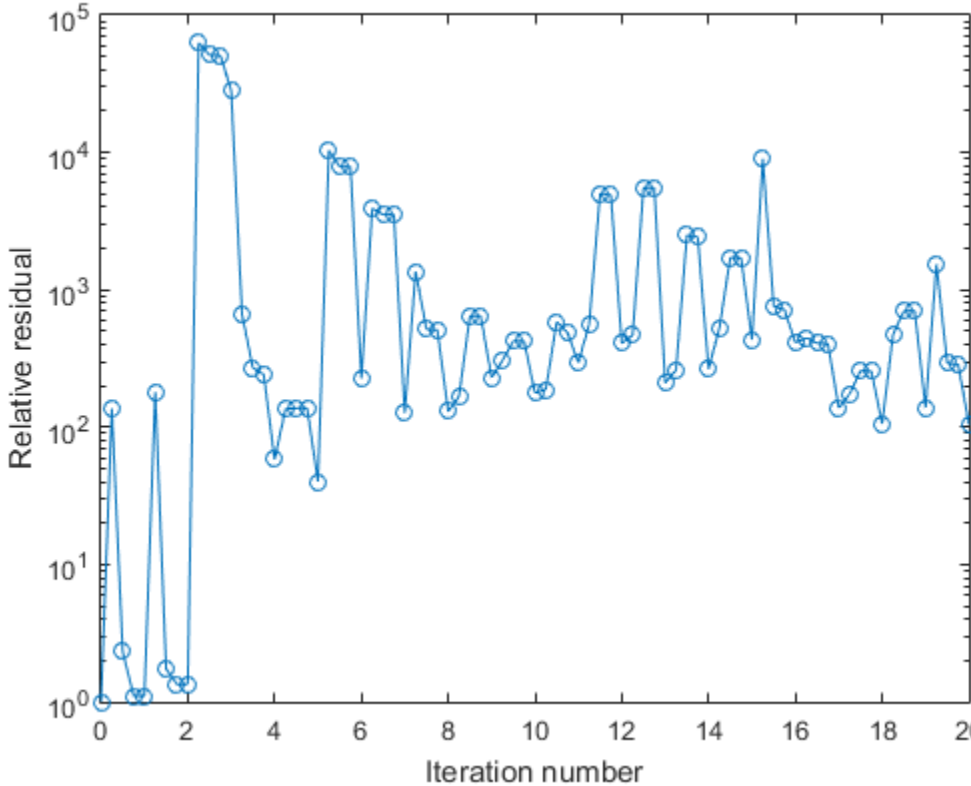
Use `bicgstabl` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = bicgstabl(A,b,tol,maxit);
```

`fl0` is 1 because `bicgstabl` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicgstabl` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `bicgstabl`.

```
semilogy(0:0.25:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create a preconditioner with `ilu`, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));

Error using ilu
There is a pivot equal to zero.  Consider decreasing the
drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

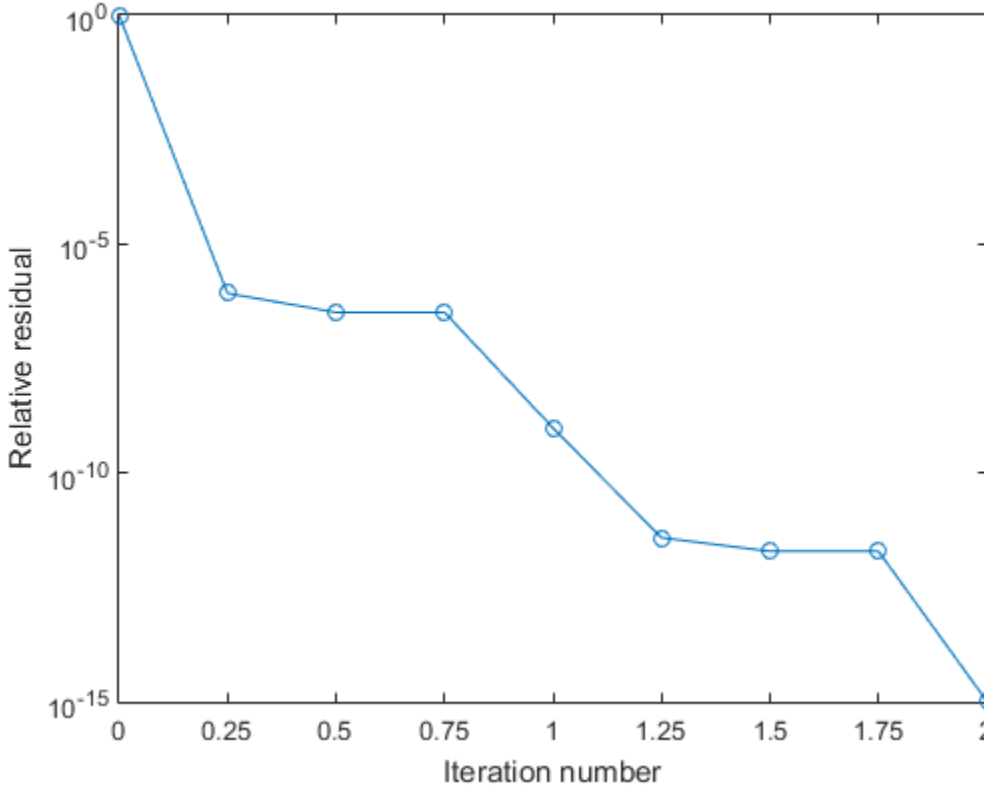
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicgstabl(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicgstabl` drives the relative residual to `1.0257e-015` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(9)` is `norm(b-A*x2)` since `bicgstabl` uses quarter iterations.

You can follow the progress of `bicgstabl` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:0.25:it1,rv1/norm(b),'-o');

h = gca;
h.XTick = 0:0.25:it1;

xlabel('Iteration number');
ylabel('Relative residual');
```



See Also

`bicg` | `bicgstab` | `cgs` | `function_handle` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

bin2dec

Convert binary number string to decimal number

Syntax

```
bin2dec(binarystr)
```

Description

`bin2dec(binarystr)` interprets the binary string `binarystr` and returns the equivalent decimal number. `binarystr` must represent a nonnegative integer value smaller than or equal to 2^52.

`bin2dec` ignores any space (`' '`) characters in the input string.

Examples

Binary `010111` converts to decimal `23`:

```
bin2dec('010111')
ans =
    23
```

Because space characters are ignored, this string yields the same result:

```
bin2dec(' 010   111 ')
ans =
    23
```

See Also

`dec2bin`

binary

Class: FTP

Set FTP transfer type to binary

Syntax

```
binary(ftpobj)
```

Description

`binary(ftpobj)` sets the FTP download and upload mode to binary, which does not convert new line characters. Binary mode is the default for FTP objects. If you previously called the `ascii` method, use this method before transferring a nontext file, such as an executable or ZIP archive.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

Examples

Connect to the MathWorks FTP server, and set the transfer mode to binary:

```
mw=ftp('ftp.mathworks.com');
binary(mw)
```

See Also

`ascii` | `ftp`

Introduced before R2006a

bitand

Bit-wise AND

Syntax

```
intout = bitand(integ1,integ2)
intout = bitand(integ1,integ2,assumedtype)

objout = bitand(netobj1,netobj2)
```

Description

`intout = bitand(integ1,integ2)` returns the bit-wise AND of values integ1 and integ2.

`intout = bitand(integ1,integ2,assumedtype)` assumes that integ1 and integ2 are of assumedtype.

`objout = bitand(netobj1,netobj2)` returns the bit-wise AND of the .NET enumeration objects netobj1 and netobj2.

Examples

Truth Table

Create a truth table for the logical AND operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitand(A, B)
```

```
TTable =

    0    0
    0    1
```

`bitand` returns 1 only if both bit-wise inputs are 1.

Negative Values

Explore how `bitand` handles negative values

MATLAB encodes signed integers using two's complement. Thus, the bit-wise AND of –5 (11111011) and 6 (00000110) is 2 (00000010).

```
C = -5;
D = 6;
bitand(C,D,'int8')
```

```
ans =

     2
```

- "Creating .NET Enumeration Bit Flags"

Input Arguments

`integ1, integ2` — Input values

signed integer arrays | unsigned integer arrays | double arrays

Input values, specified as signed integer arrays, unsigned integer arrays, or double arrays. integ1 and integ2 must be the same data type, or one must be a scalar double value.

- If integ1 and integ2 are double arrays, and assumedtype is not specified, then MATLAB treats integ1 and integ2 as unsigned 64-bit integers.
- If assumedtype is specified, then all elements in integ1 and integ2 must have integer values within the range of assumedtype.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`assumedtype` — Assumed data type of `integ1` and `integ2`

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of integ1 and integ2, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If integ1 and integ2 are double arrays, then assumedtype can specify any valid integer type, but defaults to `'uint64'`.
- If integ1 and integ2 are integer type arrays, then assumedtype must specify that same integer type.

Data Types: `char`

`netobj1`, `netobj2` — Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

`bitand` is an instance method for MATLAB enumeration objects created from a .NET enumeration.

Output Arguments

`intout` — Bit-wise AND result

signed integer array | unsigned integer array | double array

Bit-wise AND result, returned as a signed integer array, unsigned integer array, or double array. intout is the same data type and size as integ1 and integ2.

- If either integ1 or integ2 is a scalar double, and the other is a non-double integer type, intout is the non-double integer type.

`objout` — Bit-wise AND result

.NET enumeration object

Bit-wise AND result, returned as a .NET enumeration objects.

See Also

`bitcmp` | `bitget` | `bitnot` | `bitor` | `bitset` | `bitshift` | `bitxor` | `intmax`

Introduced before R2006a

bitcmp

Bit-wise complement

Compatibility

`bitcmp(A,N)` has been removed. Use `bitcmp(A)` or `bitcmp(A,assumedtype)` instead.

Syntax

```
cmp = bitcmp(A)
cmp = bitcmp(A,assumedtype)

cmp = bitcmp(A,N)
```

Description

`cmp = bitcmp(A)` returns the bit-wise complement of A.

`cmp = bitcmp(A,assumedtype)` assumes that A is of assumedtype.

`cmp = bitcmp(A,N)` returns an N-bit complement of A. Elements of A cannot exceed 2^N-1.

Examples

Complement of a Negative Integer

```
A = int8(-11);
cmp = bitcmp(A)

cmp =

   10
```

You can see the complement operation when the numbers are shown in binary.

```
original = bitget(A,8:-1:1)
complement = bitget(bitcmp(A),8:-1:1)
```

```
original =

    1    1    1    1    0    1    0    1


complement =

    0    0    0    0    1    0    1    0
```

Complement of Unsigned Integers

```
cmp = bitcmp(64,'uint8')
maxint = intmax('uint8') - 64
```

```
cmp =

   191


maxint =

  191
```

The complement of an unsigned integer is equal to itself subtracted from the maximum integer of its data type.

Input Arguments

A — Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If A is a double array, and assumedtype is not specified, then MATLAB treats A as an unsigned 64-bit integer.
- If assumedtype is specified, then all elements in A must have integer values within the range of assumedtype.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`assumedtype` — Assumed data type of A

`'uint64' | 'uint32' | 'uint16' | 'uint8' | 'int64' | 'int32' | 'int16' | 'int8'`

Assumed data type of A, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If A is a double array, then assumedtype can specify any valid integer type, but defaults to `'uint64'`.
- If A is an integer type array, then assumedtype must specify that same integer type.

Data Types: `char`

`N` — number of returned bits

integer

Number of returned bits, specified as an integer. N cannot exceed the number of bits in the integer type of A.

Data Types: `double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Output Arguments

`cmp` — Bit-wise complement

signed integer array | unsigned integer array | double array

Bit-wise complement, returned as a signed integer array, unsigned integer array, or double array. cmp is the same size and type as A.

See Also

`bitand | bitget | bitor | bitset | bitshift | bitxor | intmax`

Introduced before R2006a

bitget

Get bit at specified position

Syntax

```
b = bitget(A,bit)
b = bitget(A,bit,assumedtype)
```

Description

`b = bitget(A,bit)` returns the bit value at position bit in integer array A.

`b = bitget(A,bit,assumedtype)` assumes that A is of assumedtype.

Examples

Maximum Integer

Find the difference in the binary representation between the maximum integer of signed and unsigned integers.

```
a1 = intmax('int8');
a2 = intmax('uint8');
b1 = bitget(a1,8:-1:1)
b2 = bitget(a2,8:-1:1)
```

```
b1 =

    0     1     1     1     1     1     1     1


b2 =

    1     1     1     1     1     1     1     1
```

The signed integers require a bit to accommodate negative integers.

Negative Numbers Using Two's Complement

Find the 8-bit representation of a negative number.

```
A = -29;
b = bitget(A,8:-1:1,'int8')
```

```
b =

     1     1     1     0     0     0     1     1
```

Input Arguments

A — Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If A is a double array, and assumedtype is not specified, then MATLAB treats A as an unsigned 64-bit integer.
- If assumedtype is specified, then all elements in A must have integer values within the range of assumedtype.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

bit — Bit position

integer | integer array

Bit position, specified as an integer or integer array. bit must be between 1 (the least-significant bit) and the number of bits in the integer class of A.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype — Assumed data type of A

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of A, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If A is a double array, then assumedtype can specify any valid integer type, but defaults to `'uint64'`.
- If A is an integer type array, then assumedtype must specify that same integer type.

Data Types: `char`

Output Arguments

b — Bit value at `bit`

0 | 1

Bit value at bit, returned as `0` or `1`. b is the same data type as A.

See Also

`bitand` | `bitcmp` | `bitor` | `bitset` | `bitshift` | `bitxor` | `intmax`

Introduced before R2006a

bitmax

Maximum double-precision floating-point integer

Compatibility

`bitmax` has been removed. Use `flintmax` instead.

Syntax

```
bitmax
```

Description

`bitmax` returns the maximum unsigned double-precision floating-point integer for your computer. It is the value when all bits are set, namely the value $2^{53} - 1$.

Note Instead of integer-valued double-precision variables, use unsigned integers for bit manipulations and replace `bitmax` with `intmax`.

Examples

Display in different formats the largest floating point integer and the largest 32 bit unsigned integer:

```
format long e
bitmax
ans =
   9.007199254740991e+015

intmax('uint32')
ans =
   4294967295
```

```
format hex
bitmax
ans =
   433fffffffffffff

intmax('uint32')
ans =
   ffffffff
```

In the second bitmax statement, the last 13 hex digits of `bitmax` are `f`, corresponding to 52 1's (all 1's) in the mantissa of the binary representation. The first 3 hex digits correspond to the sign bit 0 and the 11 bit biased exponent `10000110011` in binary (`1075` in decimal), and the actual exponent is (`1075-1023`) = 52. Thus the binary value of `bitmax` is `1.111...111 x 2^52` with 52 trailing 1's, or `2^53-1`.

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitshift` | `bitxor`

Introduced before R2006a

bitnot

.NET enumeration object bit-wise NOT instance method

Syntax

```
objout = bitnot(netobj)
```

Description

`objout = bitnot(netobj)` reverses all bits of the .NET enumeration objects netobj.

Examples

- "Creating .NET Enumeration Bit Flags"

Input Arguments

`netobj` — Input value

.NET enumeration objects

Input value, specified as .NET enumeration object. You must be running a version of Windows to use .NET enumeration objects as input arguments.

Output Arguments

`objout` — Bit-wise NOT result

.NET enumeration object

Bit-wise NOT result, returned as a .NET enumeration object.

Limitations

- The method is an instance method for MATLAB enumeration objects created from a .NET enumeration. This method does not have an equivalent MATLAB function.

See Also

`bitand` | `bitor` | `bitxor`

bitor

Bit-wise OR

Syntax

```
intout = bitor(integ1,integ2)
intout = bitor(integ1,integ2,assumedtype)

objout = bitor(netobj1,netobj2)
```

Description

`intout = bitor(integ1,integ2)` returns the bit-wise OR of integ1 and integ2.

`intout = bitor(integ1,integ2,assumedtype)` assumes that integ1 and integ2 are of assumedtype.

`objout = bitor(netobj1,netobj2)` returns the bit-wise OR of the .NET enumeration objects netobj1 and netobj2.

Examples

Truth Table

Create a truth table for the logical OR operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitor(A, B)
```

```
TTable =

    0    1
    1    1
```

`bitor` returns 1 if either bit-wise input is 1.

Negative Values

Explore how `bitor` handles negative values.

MATLAB encodes negative integers using two's complement. Thus, the bit-wise OR of -5 (11111010) and 6 (00000110) is -1 (11111110).

```
C = -5;
D = 6;
bitor(C,D,'int8')
```

```
ans =

    -1
```

- "Creating .NET Enumeration Bit Flags"

Input Arguments

`integ1, integ2` — Input values

signed integer arrays | unsigned integer arrays | double arrays

Input values, specified as signed integer arrays, unsigned integer arrays, or double arrays. integ1 and integ2 must be the same data type, or one must be a scalar double value.

- If integ1 and integ2 are double arrays, and assumedtype is not specified, then MATLAB treats integ1 and integ2 as unsigned 64-bit integers.
- If assumedtype is specified, then all elements in integ1 and integ2 must have integer values within the range of assumedtype.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`assumedtype` — Assumed data type of `integ1` and `integ2`

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of integ1 and integ2, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If integ1 and integ2 are double arrays, then assumedtype can specify any valid integer type, but defaults to `'uint64'`.
- If integ1 and integ2 are integer type arrays, then assumedtype must specify that same integer type.

Data Types: `char`

`netobj1`, `netobj2` — Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

`bitor` is an instance method for MATLAB enumeration objects created from a .NET enumeration.

Output Arguments

`intout` — Bit-wise OR result

signed integer array | unsigned integer array | double array

Bit-wise OR result, returned as a signed integer array, unsigned integer array, or double array. intout is the same data type and size as integ1 and integ2.

- If either integ1 or integ2 is a scalar double, and the other is a non-double integer type, intout is the non-double integer type.

`objout` — Bit-wise OR result

.NET enumeration object

Bit-wise OR result, returned as a .NET enumeration objects.

See Also

`bitand` | `bitcmp` | `bitget` | `bitnot` | `bitset` | `bitshift` | `bitxor` | `intmax`

Introduced before R2006a

bitset

Set bit at specific location

Syntax

```
intout = bitset(A,bit)
intout = bitset(A,bit,assumedtype)

intout = bitset(A,bit,V)
intout = bitset(A,bit,V,assumedtype)
```

Description

`intout = bitset(A,bit)` returns the value of A with position bit set to 1 (on).

`intout = bitset(A,bit,assumedtype)` assumes A is of type assumedtype.

`intout = bitset(A,bit,V)` returns A with position bit set to the value of V.

- If V is zero, then the bit position bit is set to 0 (off).
- If V is nonzero, then the bit position bit is set to 1 (on).

`intout = bitset(A,bit,V,assumedtype)` assumes A is of type assumedtype.

Examples

Set Bits to On

Add powers of 2 onto a number.

```
A = 4;
intout = bitset(A,4:6)
```

```
intout =

    12    20    36
```

You can see that `bitset` sequentially turns on bits 4 through 6.

```
c = dec2bin(intout)
```

```
c =

001100
010100
100100
```

Out of Range of Integer Type

MATLAB throws an error if you specify an integer outside the range of assumedtype.

```
intout = bitset(300,5,'int8')
```

```
Error using bitset
Double inputs must have integer values in the range of ASSUMEDTYPE.
```

You can avoid this error by limiting your input to the range of the specified datatype.

Set Bits to Off

Repeatedly subtract powers of 2 from a number.

```
a = intmax('uint8')
for k = 0:7
   a = bitset(a, 8-k, 0);
   b(1,k+1) = a;
end
b
```

```
a =

  255


b =

   127    63    31    15     7     3     1     0
```

Set Multiple Bits

Set multiple bits to different values

```
bits = 2:6;
```

```
val = [1 0 0 1 1];
intout = bitset(0,bits,val,'int8')

intout =

      2     0     0    16    32
```

Input Arguments

A — Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If A is a double array, and assumedtype is not specified, then MATLAB treats A as an unsigned 64-bit integer.
- If assumedtype is specified, then all elements in A must have integer values within the range of assumedtype.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`bit` — Bit position

integer | integer array

Bit position, specified as an integer or integer array. bit must be between 1 (the least significant bit) and the number of bits in the integer class of A.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`assumedtype` — Assumed data type of A

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of A, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If A is a double array, then assumedtype can specify any valid integer type, but defaults to `'uint64'`.
- If A is an integer type array, then assumedtype must specify that same integer type.

Data Types: `char`

`V` — bit value

scalar | numeric array

Bit value, specified as a scalar or a numeric array. If V and bit are arrays, they must be the same size.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

`intout` — Adjusted integer

signed integer array | unsigned integer array | double array

Adjusted integer, returned as a signed integer array, unsigned integer array, or double array. intout is the same size and type as A.

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitshift` | `bitxor` | `intmax`

Introduced before R2006a

bitshift

Shift bits specified number of places

Compatibility

`bitshift(A,k,N)` has been removed. Use `bitshift(A,k)` or `bitshift(A,k,assumedtype)` instead.

`bitshift` no longer interprets double values as 53-bit unsigned integers by default, but as `uint64` values.

Syntax

```
intout = bitshift(A,k)
intout = bitshift(A,k,assumedtype)

intout = bitshift(A,k,N)
```

Description

`intout = bitshift(A,k)` returns A shifted to the left by k bits, equivalent to multiplying by 2^k. Negative values of k correspond to shifting bits right or dividing by $2^{|k|}$ and rounding to the nearest integer towards negative infinity. Any overflow bits are truncated.

- If A is an array of signed integers, then `bitshift` returns the arithmetic shift results, preserving the signed bit when k is negative, and not preserving the signed bit when k is positive.
- If k is positive, MATLAB shifts the bits to the left and inserts k 0-bits on the right.
- If k is negative and A is nonnegative, then MATLAB shifts the bits to the right and inserts |k| 0-bits on the left.
- If k is negative and A is negative, then MATLAB shifts the bits to the right and inserts |k| 1-bits on the left.

`intout = bitshift(A,k,assumedtype)` assumes A is of type assumedtype.

`intout = bitshift(A,k,N)` truncates any bits that overflow N bits.

Examples

Shifted 8-bit Integer

Repeatedly shift the bits of an unsigned 8-bit value to the left until all the nonzero bits overflow.

```
a = intmax('uint8');
s1 = 'Initial uint8 value %5d is %08s in binary\n';
s2 = 'Shifted uint8 value %5d is %08s in binary\n';
fprintf(s1,a,dec2bin(a))
 for i = 1:8
    a = bitshift(a,1);
    fprintf(s2,a,dec2bin(a))
 end
```

```
Initial uint8 value   255 is 11111111 in binary
Shifted uint8 value   254 is 11111110 in binary
Shifted uint8 value   252 is 11111100 in binary
Shifted uint8 value   248 is 11111000 in binary
Shifted uint8 value   240 is 11110000 in binary
Shifted uint8 value   224 is 11100000 in binary
Shifted uint8 value   192 is 11000000 in binary
Shifted uint8 value   128 is 10000000 in binary
Shifted uint8 value     0 is 00000000 in binary
```

Different Results for Different Integer Types

Find the shift for a number using different assumed integer types.

```
uintout = bitshift(6,5:7,'uint8')
intout = bitshift(6,5:7,'int8')
```

```
uintout =

   192   128     0


intout =
```

```
   -64  -128     0
```

Input Arguments

A — Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If A is a double array, and assumedtype is not specified, then MATLAB treats A as an unsigned 64-bit integer.
- If assumedtype is specified, then all elements in A must have integer values within the range of assumedtype.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

k — Number of switched bits

integer | integer array

Number of switched bits, specified as an integer or integer array.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype — Assumed data type of A

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of A, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If A is an integer type array, then assumedtype must specify that same integer type.
- If A is a double array, then assumedtype can specify any valid integer type.

Data Types: `char`

N — Number of bits kept

nonnegative integer | nonnegative integer array

Number of bits kept, specified as a nonnegative integer or integer array. N must be less than or equal to the number of bits in the unsigned integer class of A.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`intout` — Shifted value

signed integer array | unsigned integer array | double array

Shifted value, returned as signed integer array, unsigned integer array, or double array. intout is the same size and type as A.

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitxor` | `intmax`

Introduced before R2006a

bitxor

Bit-wise XOR

Syntax

```
intout = bitxor(integ1,integ2)
intout = bitxor(integ1,integ2,assumedtype)

objout = bitxor(netobj1,netobj2)
```

Description

`intout = bitxor(integ1,integ2)` returns the bit-wise XOR of integ1 and integ2.

`intout = bitxor(integ1,integ2,assumedtype)` assumes that integ1 and integ2 are of assumedtype.

`objout = bitxor(netobj1,netobj2)` returns the bit-wise XOR of the .NET enumeration objects netobj1 and netobj2.

Examples

Truth Table

Create a truth table for the logical XOR operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitxor(A, B)
```

```
TTable =

    0    1
    1    0
```

`bitxor` returns 0 if both bit-wise inputs are equal.

Negative Values

Explore how `bitxor` handles negative values.

MATLAB encodes negative integers using two's complement. Thus, the bit-wise XOR of -5 (11111010) and 6 (00000110) is -3 (11111100).

```
C = -5;
D = 6;
bitxor(C,D,'int8')
```

```
ans =

    -3
```

- "Creating .NET Enumeration Bit Flags"

Input Arguments

`integ1, integ2` — Input values

signed integer arrays | unsigned integer arrays | double arrays

Input values, specified as signed integer arrays, unsigned integer arrays, or double arrays. integ1 and integ2 must be the same data type, or one must be a scalar double value.

- If integ1 and integ2 are double arrays, and assumedtype is not specified, then MATLAB treats integ1 and integ2 as unsigned 64-bit integers.
- If assumedtype is specified, then all elements in integ1 and integ2 must have integer values within the range of assumedtype.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`assumedtype` — Assumed data type of `integ1` and `integ2`

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of integ1 and integ2, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If integ1 and integ2 are double arrays, then assumedtype can specify any valid integer type, but defaults to `'uint64'`.
- If integ1 and integ2 are integer type arrays, then assumedtype must specify that same integer type.

Data Types: `char`

`netobj1`, `netobj2` — Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

`bitxor` is an instance method for MATLAB enumeration objects created from a .NET enumeration.

Output Arguments

`intout` — Bit-wise XOR result

signed integer array | unsigned integer array | double array

Bit-wise XOR result, returned as a signed integer array, unsigned integer array, or double array. intout is the same data type and size as integ1 and integ2.

- If either integ1 or integ2 is a scalar double, and the other is a non-double integer type, intout is the non-double integer type.

`objout` — Bit-wise XOR result

.NET enumeration object

Bit-wise XOR result, returned as a .NET enumeration objects.

See Also

`bitand` | `bitcmp` | `bitget` | `bitnot` | `bitor` | `bitset` | `bitshift` | `intmax`

Introduced before R2006a

blanks

Create string of blank characters

Syntax

```
blanks(n)
```

Description

`blanks(n)` is a string of `n` blanks.

Examples

`blanks` is useful with the `display` function. For example,

```
disp(['xxx' blanks(20) 'yyy'])
```

displays twenty blanks between the strings `'xxx'` and `'yyy'`.

`disp(blanks(n)')` moves the cursor down `n` lines.

See Also

`clc` | `format` | `home`

Introduced before R2006a

blkdiag

Construct block diagonal matrix from input arguments

Syntax

```
out = blkdiag(a,b,c,d,...)
```

Description

`out = blkdiag(a,b,c,d,...)`, where `a`, `b`, `c`, `d`, `...` are matrices, outputs a block diagonal matrix of the form

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & \ldots \end{bmatrix}$$

The input matrices do not have to be square, nor do they have to be of equal size.

See Also

`diag` | `horzcat` | `vertcat`

Introduced before R2006a

boundary

Boundary of a set of points in 2-D or 3-D

Syntax

```
k = boundary(x,y)
k = boundary(x,y,z)
k = boundary(P)
k = boundary( ___ ,s)
[k,v] = boundary( ___ )
```

Description

`k = boundary(x,y)` returns a vector of point indices representing a single conforming 2-D boundary around the points `(x,y)`. The points `(x(k),y(k))` form the boundary. Unlike the convex hull, the boundary can shrink towards the interior of the hull to envelop the points.

`k = boundary(x,y,z)` returns a triangulation representing a single conforming 3-D boundary around the points `(x,y,z)`. Each row of `k` is a triangle defined in terms of the point indices.

`k = boundary(P)` specifies points `(x,y)` or `(x,y,z)` in the columns of matrix `P`.

`k = boundary( ___ ,s)` specifies shrink factor `s` using any of the previous syntaxes. `s` is a scalar between `0` and `1`. Setting `s` to `0` gives the convex hull, and setting `s` to `1` gives a compact boundary that envelops the points. The default shrink factor is `0.5`.

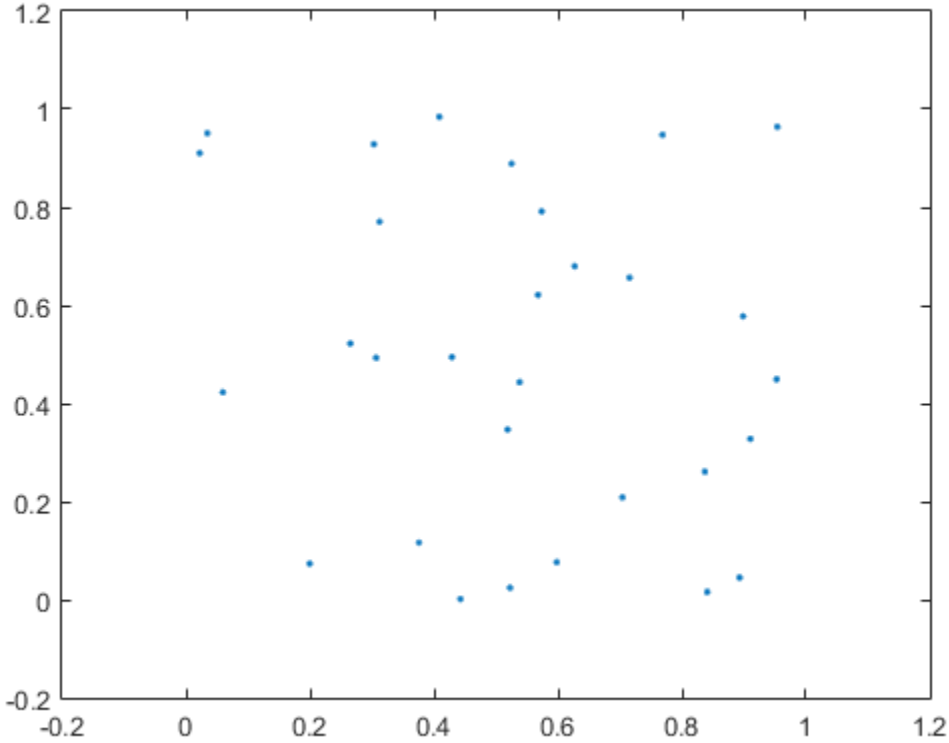
`[k,v] = boundary( ___ )` also returns a scalar `v`, which is the area (2-D) or volume (3-D) which boundary `k` encloses.

Examples

Boundary of 2-D Point Cloud

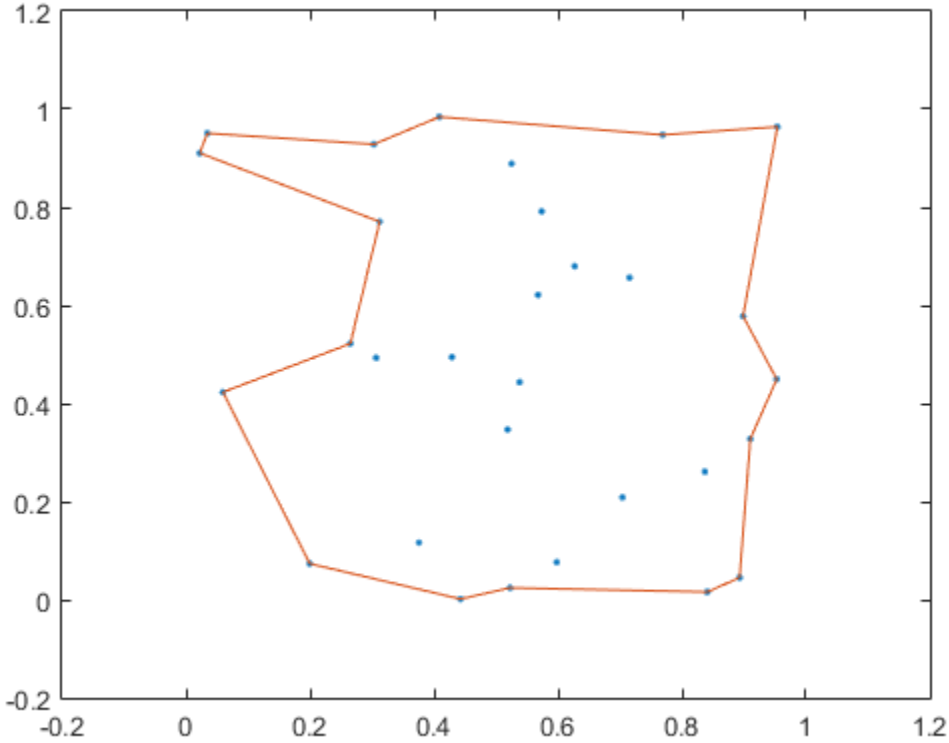
Create and plot a set of random 2-D points.

```
x = gallery('uniformdata',30,1,1);
y = gallery('uniformdata',30,1,10);
plot(x,y,'.')
xlim([-0.2 1.2])
ylim([-0.2 1.2])
```



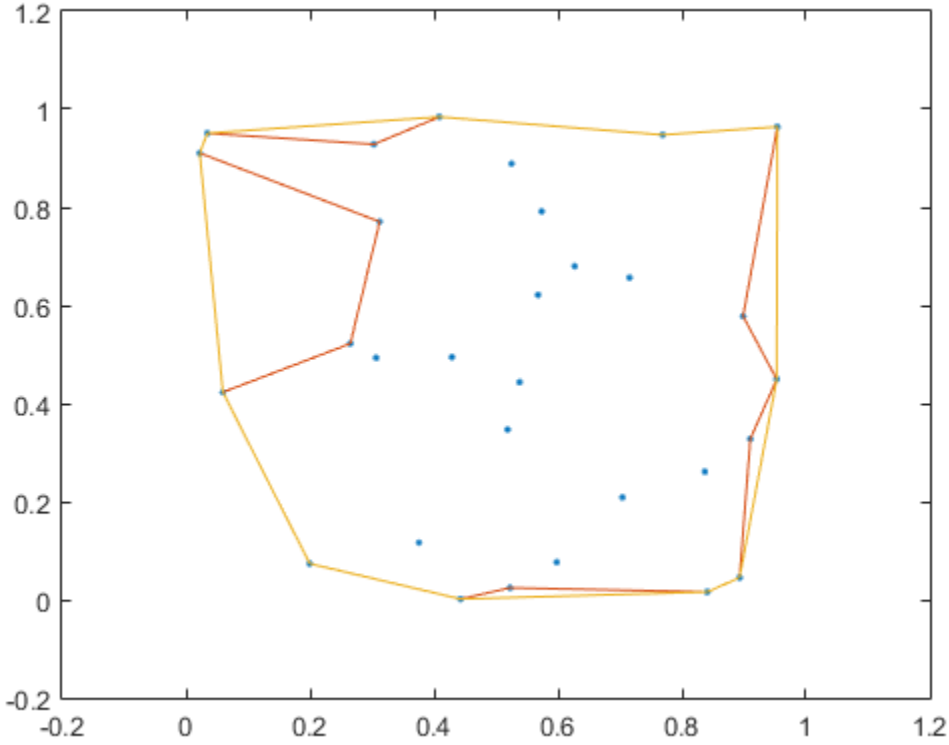
Compute a boundary around the points using the default shrink factor.

```
k = boundary(x,y);
hold on;
plot(x(k),y(k));
```



Create a new boundary around the points using a shrink factor of 0.1. The result is a less compact boundary enveloping the points.

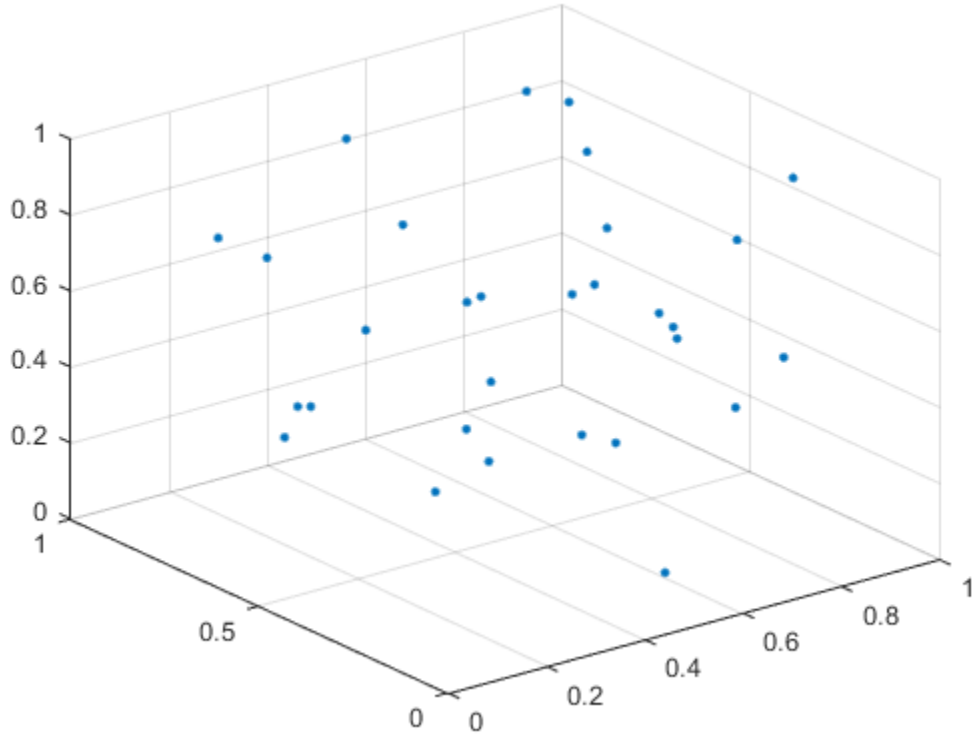
```
j = boundary(x,y,0.1);
hold on;
plot(x(j),y(j));
```



Boundary of 3-D Point Cloud

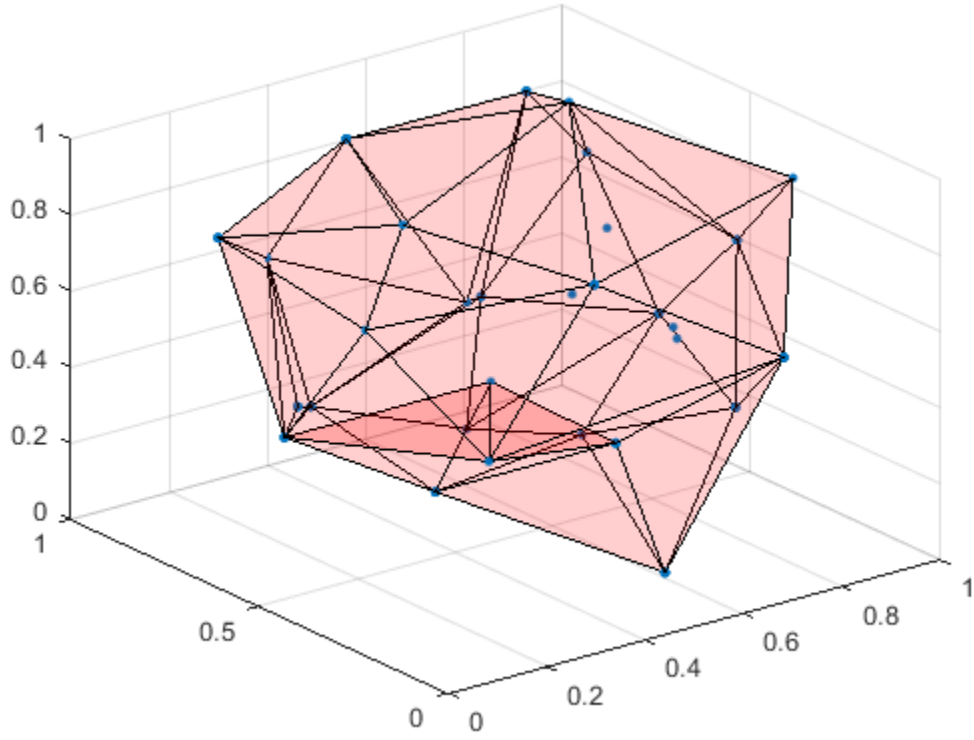
Create and plot a set of random 3-D points.

```
P = gallery('uniformdata',30,3,5);
plot3(P(:,1),P(:,2),P(:,3),'.','MarkerSize',10)
grid on
```



Plot the boundary using the default shrink factor.

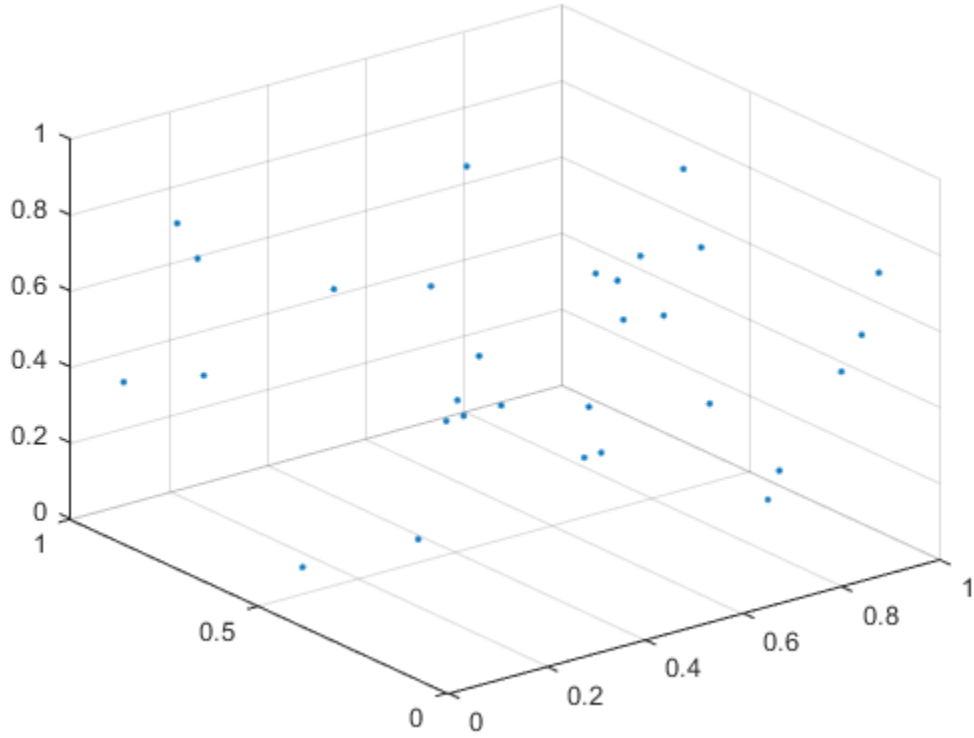
```
k = boundary(P);
hold on
trisurf(k,P(:,1),P(:,2),P(:,3),'Facecolor','red','FaceAlpha',0.1)
```



Shrink Factor Effect on 3-D Boundary

Create and plot a set of random 3-D points.

```
P = gallery('uniformdata',30,3,8);
plot3(P(:,1),P(:,2),P(:,3),'.')
grid on
```



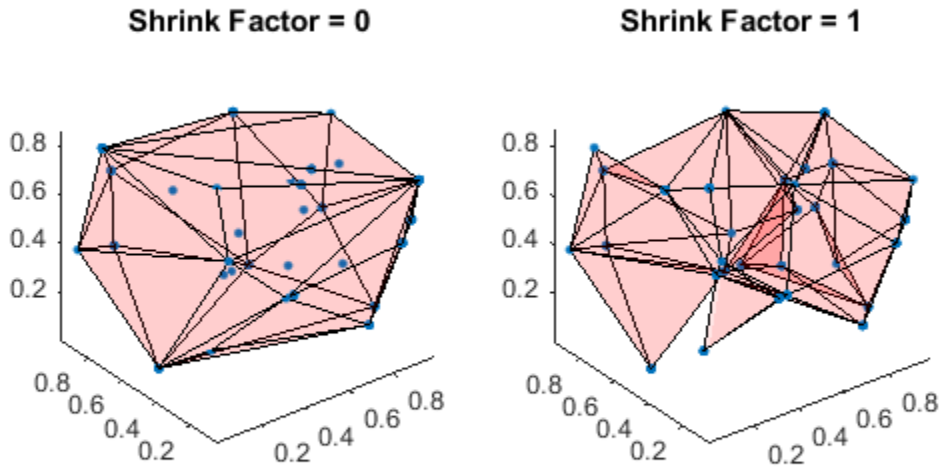
Compute two boundaries: one with a shrink factor of 0 and the other with a shrink factor of 1.

```
k = boundary(P,0);
j = boundary(P,1);
```

Compare the shrink factors by plotting the original points and the two boundaries side-by-side.

```
subplot(1,2,1);
plot3(P(:,1),P(:,2),P(:,3),'.','MarkerSize',10)
hold on
trisurf(k,P(:,1),P(:,2),P(:,3),'Facecolor','red','FaceAlpha',0.1)
axis equal
```

```
title('Shrink Factor = 0')

subplot(1,2,2);
plot3(P(:,1),P(:,2),P(:,3),'.','MarkerSize',10)
hold on
trisurf(j,P(:,1),P(:,2),P(:,3),'Facecolor','red','FaceAlpha',0.1)
axis equal
title('Shrink Factor = 1')
```

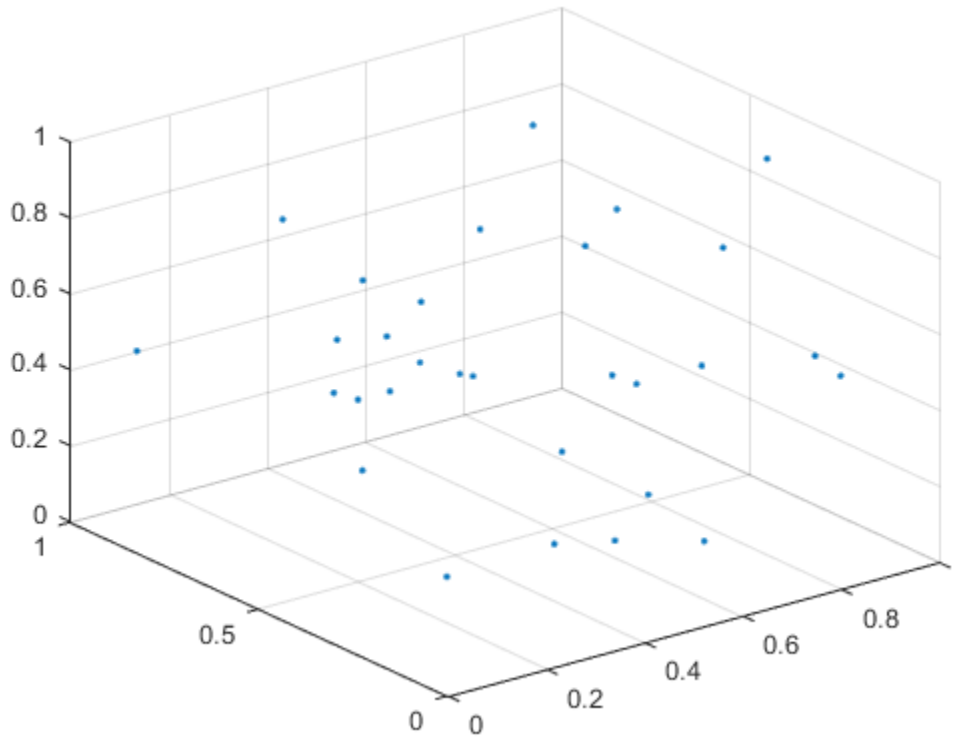


Volume of 3-D Boundary

Create and plot a set of random 3-D points.

```
P = gallery('uniformdata',30,3,1);
```

```
plot3(P(:,1),P(:,2),P(:,3),'.')
grid on
```



Use the `boundary` function to compute a boundary around the points, and find the volume of the resulting shape.

```
[~, vol] = boundary(P);
vol
```

```
vol =

    0.3012
```

Input Arguments

x — x-coordinates of points

column vector

x-coordinates of points, specified as a column vector.

Data Types: `double`

y — y-coordinates of points

column vector

y-coordinates of points, specified as a column vector.

Data Types: `double`

z — z-coordinates of points

column vector

z-coordinates of points, specified as a column vector.

Data Types: `double`

P — Point coordinates

matrix with two columns | matrix with three columns

Point coordinates, specified as a matrix with two columns (for a 2-D alpha shape) or a matrix with three columns (for a 3-D alpha shape).

- For 2-D, the columns of `P` represent `x` and `y` coordinates, respectively.
- For 3-D, the columns of `P` represent `x`, `y`, and `z` coordinates, respectively.

Data Types: `double`

s — Shrink factor

`0.5` (default) | scalar in range `[0,1]`

Shrink factor, specified as a scalar in the range of `[0,1]`.

- `s = 0` corresponds to the convex hull of the points.

- `s = 1` corresponds to the tightest single-region boundary around the points.

The default shrink factor is `0.5`. Specify a larger or smaller shrink factor to tighten or loosen the boundary around the points, respectively.

Example: `k = boundary(x,y,0.76)` specifies a shrink factor of `0.76`, producing a tighter boundary than the default.

Output Arguments

k — Boundary point indices

vector | matrix

Boundary point indices, returned as a vector or matrix. `k` contains the indices of the input points that lie on the boundary.

- For 2-D problems, `k` is a column vector of point indices representing the sequence of points around the boundary, which is a polygon.
- For 3-D problems, `k` is a triangulation matrix of size `mtri`-by-`3`, where `mtri` is the number of triangular facets on the boundary. Each row of `k` defines a triangle in terms of the point indices, and the triangles collectively form a bounding polyhedron.

v — Area or volume enclosed by boundary

scalar

Area or volume enclosed by boundary, returned as a scalar.

- For 2-D problems, `v` is the area enclosed by boundary `k`.
- For 3-D problems, `v` is the volume enclosed by boundary `k`.

See Also

`alphaShape` | `convhull` | `delaunayTriangulation` | `triangulation` | `trisurf`

Introduced in R2014b

box

Axes border

Syntax

```
box on
box off
box
box(axes_handle,...)
```

Description

`box on` displays the boundary of the current axes.

`box off` does not display the boundary of the current axes.

`box` toggles the visible state of the current axes boundary.

`box(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

More About

Algorithms

The `box` function sets the axes `Box` property to `on` or `off`.

See Also

`axes` | `grid`

Introduced before R2006a

break

Terminate execution of for or while loop

Syntax

```
break
```

Description

`break` terminates the execution of a `for` or `while` loop. Statements in the loop after the `break` statement do not execute.

In nested loops, `break` exits only from the loop in which it occurs. Control passes to the statement that follows the `end` of that loop.

Examples

Exit Loop Before Expression Is False

Sum a sequence of random numbers until the next random number is greater than an upper limit. Then, exit the loop using a `break` statement.

```
limit = 0.8;
s = 0;

while 1
    tmp = rand;
    if tmp > limit
        break
    end
    s = s + tmp;
```

```
end
```

More About

Tips

- The `break` statement exits a `for` or `while` loop completely. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.
- `break` is not defined outside a `for` or `while` loop. To exit a function, use `return`.

See Also

`continue` | `end` | `for` | `return` | `while`

Introduced before R2006a

brighten

Brighten or darken colormap

Syntax

```
brighten(beta)
brighten(h,beta)
newmap = brighten(beta)
newmap = brighten(cmap,beta)
```

Description

`brighten(beta)` increases or decreases the color intensities in a colormap by replacing the current colormap with a brighter or darker colormap of essentially the same colors. The modified colormap is brighter if `0 < beta < 1` and darker if `-1 < beta < 0`. `brighten(beta)`, followed by `brighten(-beta)`, where `beta < 1`, restores the original map.

`brighten(h,beta)` brightens all objects that are children of the figure having the handle `h`.

`newmap = brighten(beta)` returns a brighter or darker version of the current colormap without changing the display.

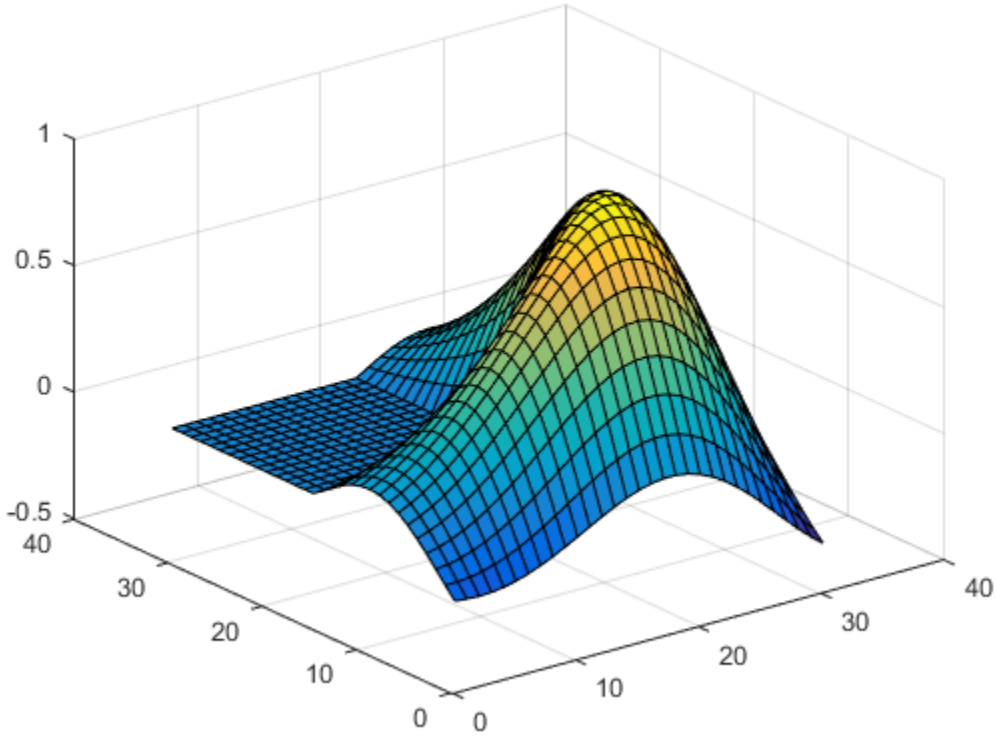
`newmap = brighten(cmap,beta)` returns a brighter or darker version of the colormap `cmap` without changing the display.

Examples

Brighten Colormap

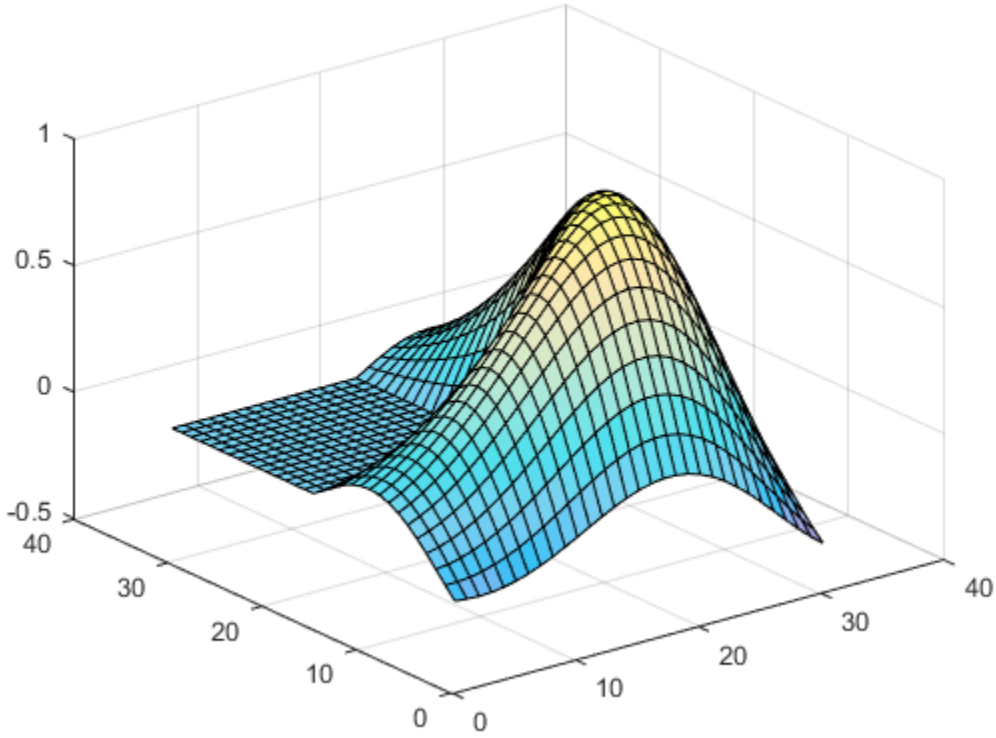
Display a surface plot of `membrane`.

```
surf(membrane)
```



Brighten the colormap.

```
beta = .7;
brighten(beta)
```



More About

Algorithms

`brighten` raises values in the colormap to the power of gamma, where gamma is

$$\gamma = \begin{cases} 1-\beta, & \beta > 0 \\ \dfrac{1}{1+\beta}, & \beta \le 0 \end{cases}$$

`brighten` has no effect on graphics objects defined with true color.

- “Altering Colormaps”

See Also

`colormap` | `rgbplot`

Introduced before R2006a

brush

Interactively mark, delete, modify, and save observations in graphs

Syntax

```
brush on
brush off
brush
brush color
brush(figure_handle,...)
brushobj = brush(figure_handle)
```

Description

Data brushing is a mode for interacting with graphs in figure windows in which you can click data points or drag a selection rectangle around data points to highlight observations in a color of your choice. Highlighting takes different forms for different types of graphs, and brushing marks persist—even in other interactive modes—until removed by deselecting them.

`brush on` turns on interactive data brushing mode.

`brush off` turns brushing mode off, leaving any brushed observations still highlighted.

`brush` by itself toggles the state of the data brushing tool.

`brush color` sets the current color used for brushing graphics to the specified `ColorSpec`. Changing brush color affects subsequent brushing, but does not change the color of observations already brushed or the brush tool's state.

`brush(figure_handle,...)` applies the function to the specified figure handle.

`brushobj = brush(figure_handle)` returns a *brush mode object* for that figure, useful for controlling and customizing the figure's brushing state. The following properties of such objects can be modified using `get` and `set`:

`Enable 'on' | {'off'}`	Specifies whether this figure mode is currently enabled on the figure.
`FigureHandle`	The associated figure handle. This property supports `get` only.
`Color`	Specifies the color to be used for brushing.

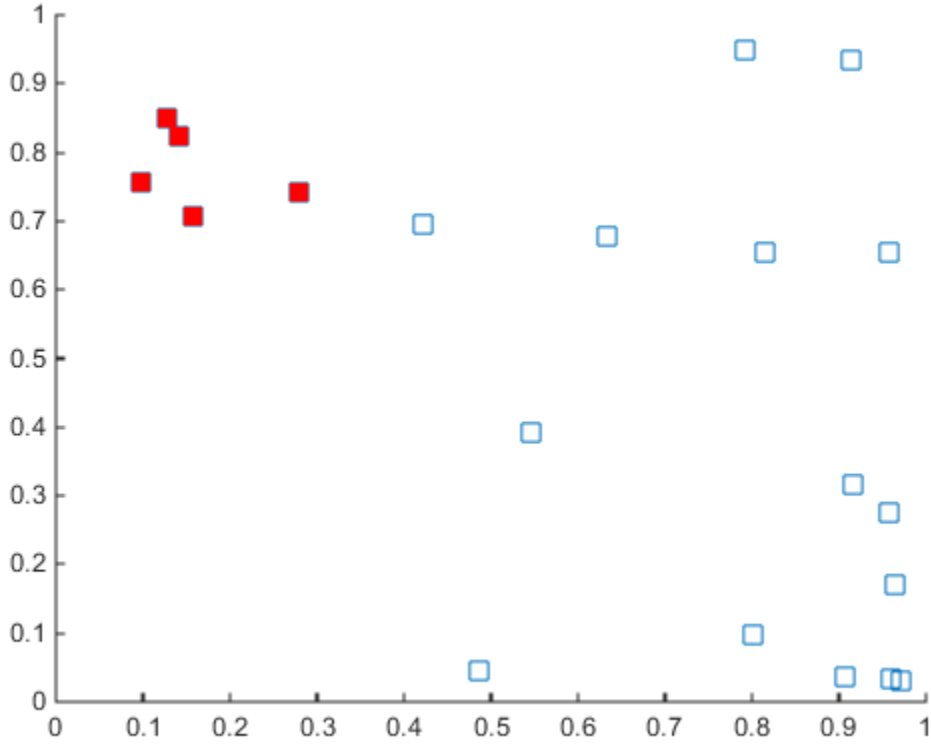
`brush` cannot return a brush mode object at the same time you are calling it to set a brushing option.

Examples

Example 1

On a scatter plot, drag out a rectangle to brush the graph:

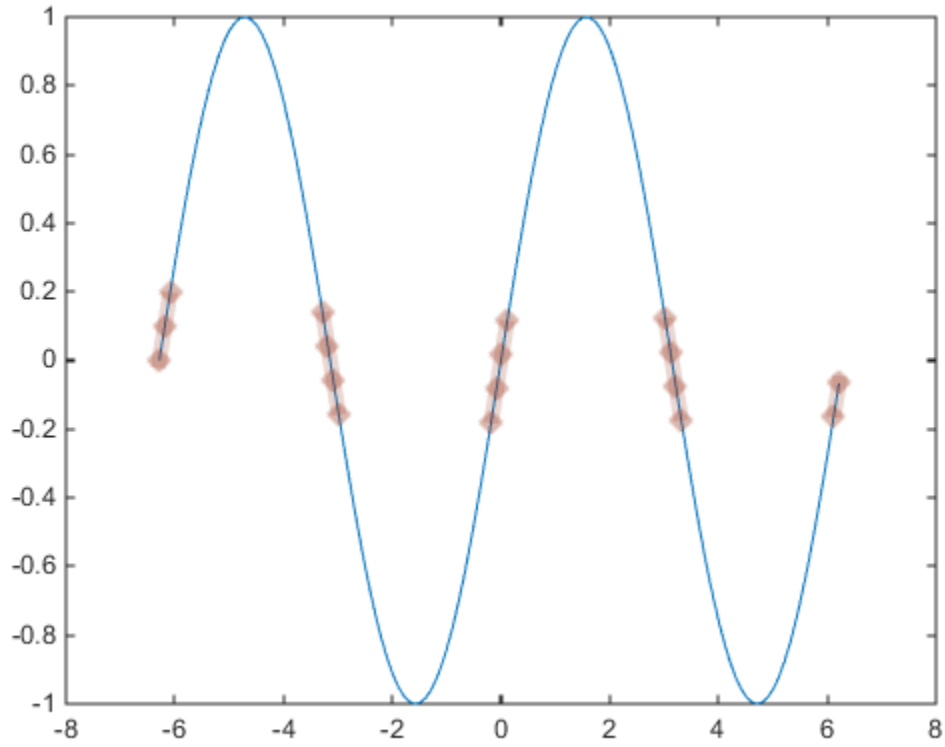
```
x = rand(20,1);
y = rand(20,1);
scatter(x,y,80,'s')
brush on
```



Example 2

Brush observations from -.2 to .2 on a line plot in dark red:

```
x = [-2*pi:.1:2*pi];
y = sin(x);
plot(x,y);
h = brush;
set(h,'Color',[.6 .2 .1],'Enable','on');
```

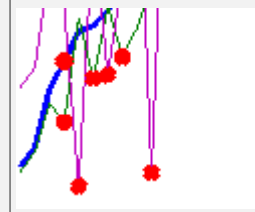
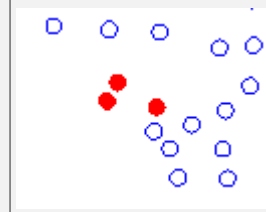
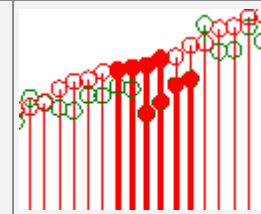
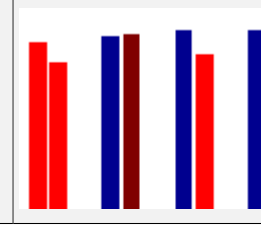


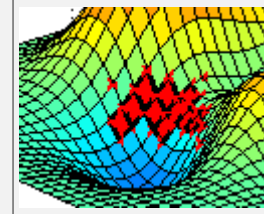
More About

Tips

Types of Plots You Can Brush

Data brushing places lines and patches on plots to create highlighting, marking different types of graphs as follows (brushing marks are shown in red):

Graph Type	Brushing Annotation	Overlays?	Example
line	Colored lines slightly wider than those in the line with a marker distinct from those on the line (filled circles if none) to identify brushed vertices. Only those line segments that connect brushed vertices are highlighted	Y	
scatter	Line with `LineStyle 'none'` and a marker with a color distinct from and slightly larger than the base scatter marker.	Y	
stem	The brushed stems and stem heads are shaded in the brushing color.	Y	
bar	The interior of selected bars is filled in the brushing color.	N	

Graph Type	Brushing Annotation	Overlays?	Example
area	Patches filling the region between selected points and the *x*-axis in the brushing color.	N	
surface	Faces with edges slightly wider than the surface line width and with a marker distinct from that of the surface (**X** if none) to identify brushed vertices. Faces are plotted only when all four vertices that define them are brushed. The brushed observations are the set of marked vertices, not the faces.	N	

When using the linked plots feature, a graph can become brushed when you brush another graph that displays some of the same data, potentially brushing the same observations more than once. The overlaid brushing marks (whether lines or markers) are slightly wider than the brushing marks that they overlay; this makes multiply brushed observations visually distinct. The wider brushing marks are placed under the narrower ones, so that if they happen to have different colors, you can see all the colors. See the subsection "How Data Linking Affects Data Brushing" on page 1-791 for more information about brushing linked figures.

As the above table indicates, only line, scatter, and stem brushing marks can be overlaid in this manner. Although you can brush them, you cannot overlay brushing marks on area, bar, or surface.

Plot Types You Cannot Brush

Currently, not all plot types enable data brushing. Graph functions that *do not* support brushing are:

- Line plots created with `line`
- Scatter plots created with `spy`
- Contour plots created with `contour`, `contourf`, or `contour3`

- Pie charts created with `pie` or `pie3`
- Radial graphs created with `polar`, `compass`, or `rose`
- Direction graphs created with `feather`, `quiver`, or `comet`
- Area and image plots created with `fill`, `image`, `imagesc`, or `pcolor`
- Bar graphs created with `pareto` or `errorbar`
- Functional plots created with `ezcontour` or `ezcontourf`
- 3-D plot types *other than* `plot3`, `stem3`, `scatter3`, `mesh`, `meshc`, `surf`, `surfl`, and `surfc`

You can use some of these functions to display base data that do not need to be brushable. For example, use `line` to plot mean *y*-values as horizontal lines that you do not need or want to brush.

Mode Exclusivity and Persistence

Data brushing mode is *exclusive*, like zoom, pan, data cursor, or plot edit mode. However, brush marks created in data brushing mode *persist* through all changes in mode. Brush marks that appear in other graphs while they are linked via `linkdata` also persist even when data linking is subsequently turned off. That is, severing connections to a graph's data sources does not remove brushing marks from it. The only ways to remove brushing marks are (in brushing mode):

- Brush an empty area in a brushed graph.
- Right-click and select **Clear all brushing** from the context menu.

Changing the brushing color for a figure does not recolor existing brush marks. If you change the brushing color and hold down the **Shift** key when brushing new data, all existing brush marks change to the new color. All brush marks that appear on linked plots in the same or different figure also change to the new color if the brushing action affects them. The behavior is the same whether you select a brushing color from the Brush Tool dropdown palette, set it by calling `brush(colorspec)`, or by setting the `Color` property of a brush mode object (e.g., `set(brushobj,'Color',colorspec)`).

How Data Linking Affects Data Brushing

When you use the Data Linking tool or call the `linkdata` function, brushing marks that you make on one plot appear on other plots that depict the same variable you are

brushing—if those plots are also linked. This happens even if the affected plot is not in Brushing mode. That is, brushing marks appear on a linked plot *in any mode* when you brush another plot linked to it via a common variable or brush that variable in the Variables editor. Be aware that the following conditions apply, however:

- The graph type must support data brushing (see "Types of Plots You Can Brush" on page 1-789 and "Plot Types You Cannot Brush" on page 1-790)
- The graphed variable must not be complex; if you can plot a complex variable you can brush it, but such graphs do not respond when you brush the complex variable in another linked plot. For more information about linking complex variables, see Example 3 in the `linkdata` reference page.
- Observations that you brush display in the same color in all linked graphs. The color is the brush color you have selected in the window you are interacting with, and can differ from the brushing colors selected in the other affected figures. When you brush linked plots, the brushing color is associated with the variable(s) you brush

The last bullet implies that brush marks on an unlinked graph can change color when data linking is turned on for that figure. Brushing marks can, in fact, vanish and be replaced by marks in the same or different color when the plot enters a linked state. In the linked state, brushing is tied to variables (data sources), not just the graphics. If different observations for the same variable on a linked figure are brushed, those variables override the brushed graphics on the newly linked plot. In other words, the newly linked graph loses all its previous brush marks when it "joins the club" of common data sources.

Mouse Gestures for Data Brushing

You can brush graphs in several ways. The basic operation is to drag the mouse to highlight all observations within the rectangle you define. The following table lists data brushing gestures and their effects.

Action	Gesture	Result
Select data using a Region Of Interest	ROI mouse drag	Region of interest (ROI) rectangle (or rectangular prism for 3-D axes) appears during the gesture and all brushable observations within the rectangle are highlighted. All other brushing marks in the axes are removed. The ROI rectangle disappears when the mouse button is released.

Action	Gesture	Result
Select a single point	Single left-click on a graphic object that supports data brushing	Produces an equivalent result to ROI rectangle, brushing where the rectangle encloses only the single vertex on the graphical object closest to the mouse. All other brushing annotations in the figure are removed.
Add a point to the selection or remove a highlighted one	Single left-click on a graphic object that supports data brushing, with the **Shift** key down	Equivalent brushing by dragging an ROI rectangle that encloses only the single vertex on the graphic object closest to the mouse. All other brushed regions in the figure remain brushed.
Select all data associated with a graphic object	Double left-click on a graphic object that supports data brushing	All vertices for the graphic object are brushed.
Add to or subtract from region of interest	Click or ROI drag with the **Shift** or **Ctrl** keys down	Region of interest grows; all unbrushed vertices within the rectangle become brushed and all brushed observations in it become unbrushed. All brushed vertices outside the ROI remain brushed.
Copy brushed data to Editor, Command Window, Variables editor, or Workspace Browser	Drag brushed data to another window or to a program/ icon on the system desktop	Equivalent to copying brushed data and pasting into other window or an existing/new variable.

Brush Mode Callbacks

You can program the following callbacks for brush mode operations.

- `ActionPreCallback <function_handle>` — Function to execute before brushing

  Use this callback to execute code when a brush operation begins. The function handle should reference a function with two implicit arguments:

  ```
  function myfunction(src,event_data)
  ```

```
% src         handle to the figure that has been clicked
% event_data   object containing event data
end
```

The event data has the following property:

`Axes`	The handle of the axes that is being brushed

- `ActionPostCallback <function_handle>` — Function to execute after brushing

  Use this callback to execute code when a brush operation ends. The function handle should reference a function with two implicit arguments:

```
function myfunction(src,event_data)
% src         handle to the figure that has been clicked
% event_data   object containing event data
%        (same as the event data of the
%             'ActionPreCallback' callback)
end
```

- "Marking Up Graphs with Data Brushing"

See Also

`linkaxes` | `pan` | `linkdata` | `rotate3d` | `zoom`

bsxfun

Apply element-by-element binary operation to two arrays with singleton expansion enabled

Syntax

```
C = bsxfun(fun,A,B)
```

Description

`C = bsxfun(fun,A,B)` applies the element-by-element binary operation specified by the function handle `fun` to arrays `A` and `B`, with singleton expansion enabled. `fun` can be one of the following built-in functions:

`@plus`	Plus
`@minus`	Minus
`@times`	Array multiply
`@rdivide`	Right array divide
`@ldivide`	Left array divide
`@power`	Array power
`@max`	Binary maximum
`@min`	Binary minimum
`@rem`	Remainder after division
`@mod`	Modulus after division
`@atan2`	Four-quadrant inverse tangent; result in radians
`@atan2d`	Four-quadrant inverse tangent; result in degrees
`@hypot`	Square root of sum of squares
`@eq`	Equal

`@ne`	Not equal
`@lt`	Less than
`@le`	Less than or equal to
`@gt`	Greater than
`@ge`	Greater than or equal to
`@and`	Element-wise logical AND
`@or`	Element-wise logical OR
`@xor`	Logical exclusive OR

`fun` can also be a handle to any binary element-wise function not listed above. A binary element-wise function of the form `C = fun(A,B)` accepts arrays `A` and `B` of arbitrary, but equal size and returns output of the same size. Each element in the output array `C` is the result of an operation on the corresponding elements of `A` and `B` only. `fun` must also support scalar expansion, such that if `A` or `B` is a scalar, `C` is the result of applying the scalar to every element in the other input array.

The corresponding dimensions of `A` and `B` must be equal to each other or equal to one. Whenever a dimension of `A` or `B` is singleton (equal to one), `bsxfun` virtually replicates the array along that dimension to match the other array. In the case where a dimension of `A` or `B` is singleton, and the corresponding dimension in the other array is zero, `bsxfun` virtually diminishes the singleton dimension to zero.

The size of the output array `C` is equal to:
`max(size(A),size(B)).*(size(A)>0 & size(B)>0)`.

Examples

Deviations from Mean

Use `bsxfun` to subtract the column mean from the corresponding column elements of a matrix, `A`.

```
A = [1 2 10; 1 4 20;1 6 15] ;
C = bsxfun(@minus, A, mean(A))
```

```
C =

     0    -2    -5
```

```
     0     0     5
     0     2     0
```

Singleton Expansion with Custom Function

Call a custom-defined binary function with `bsxfun` by specifying a handle to the function.

```
fun = @(A,B) A.*sin(B);
A = 1:7;
B = pi*[0 1/4 1/3 1/2 2/3 3/4 1].';
C = bsxfun(fun,A,B)
```

```
C =

         0         0         0         0         0         0         0
    0.7071    1.4142    2.1213    2.8284    3.5355    4.2426    4.9497
    0.8660    1.7321    2.5981    3.4641    4.3301    5.1962    6.0622
    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000
    0.8660    1.7321    2.5981    3.4641    4.3301    5.1962    6.0622
    0.7071    1.4142    2.1213    2.8284    3.5355    4.2426    4.9497
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
```

Singleton expansion allows `bsxfun` to expand the input vectors into a full matrix.

See Also

`repmat` | `arrayfun`

builddocsearchdb

Build searchable documentation database

Syntax

```
builddocsearchdb(folder)
```

Description

`builddocsearchdb(folder)` builds a searchable database, also referred to as a search index, from HTML files in the specified folder.

The `builddocsearchdb` function creates a subfolder, `helpsearch-v3`, to contain the database files. The database enables MATLAB to search for content within the HTML files assuming the MATLAB version is the same version used to create the database.

Beginning with MATLAB R2014b, you can maintain search indexes side by side. For instance, if you already have a search index for MATLAB R2014a or earlier, run `builddocsearchdb` against your help files using MATLAB R2014b or later. Then, when you run any MATLAB release, the help browser automatically uses the appropriate index for searching your documentation database.

Examples

Search Custom Help Files

Build a search database for custom help files.

MATLAB includes a set of sample files to demonstrate how to create a custom toolbox and supporting documentation. This sample toolbox is called the *Upslope Area Toolbox*. The `upslope` folder includes a file named `info.xml`, which is required to display custom documentation, and a subfolder named `html`, which contains HTML documentation and supporting files.

Copy the sample files to a temporary folder, and add the copied files to the path.

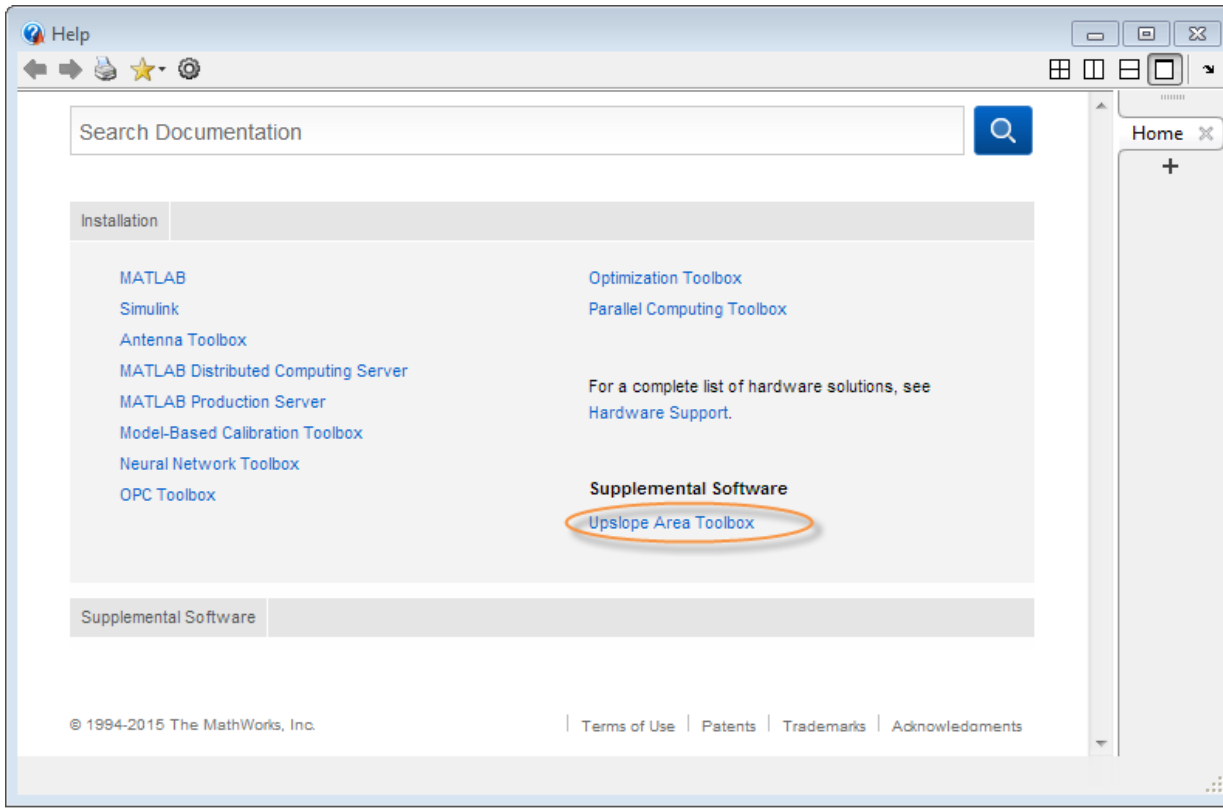
```
sample = fullfile(...
         matlabroot,'help','techdoc','matlab_env',...
```

```
        'examples','upslope');
tmp = tempname;
mkdir(tmp);
copyfile(sample,tmp);
addpath(tmp);
```

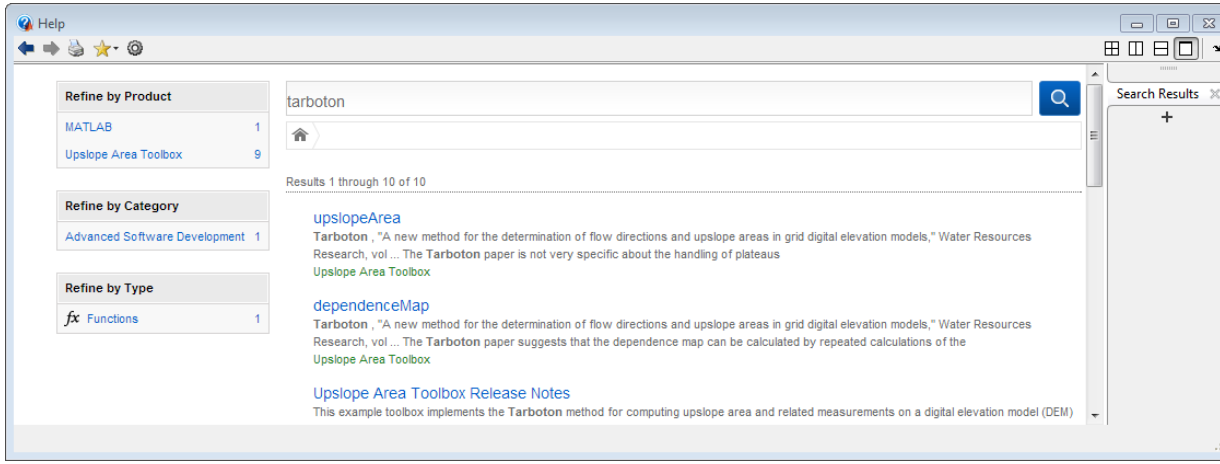
Create a search database.

```
folder = fullfile(tmp,'html');
builddocsearchdb(folder)
```

Under the Supplemental Software heading, click **Upslope Area Toolbox**.



Search the supplemental documentation for the term *tarboton*, which appears in several of the example help files. The search returns several results from the Upslope Area Toolbox.



Remove the temporary example files.

```
rmpath(tmp)
rmdir(tmp,'s')
```

Note: Your help appears alongside the MathWorks documentation only when you are viewing installed documentation. If you are viewing documentation on the web, your documentation displays in a help browser separate from the MathWorks documentation.

Input Arguments

`folder` — Full path to folder with HTML files

string

Full path to a folder with HTML files, specified as a string. The folder must be:

- On the MATLAB search path
- Outside the *`matlabroot`* folder
- Outside any installed Hardware Support Package help folder

To include a particular HTML document in the search database, the `builddocsearchdb` function requires that:

- The document has a title.

- The content is different from the title.

Example: `builddocsearchdb('c:\myfiles\html')`

More About

- " Display Custom Documentation"

See Also

`doc` | `help`

builtin

Execute built-in function from overloaded method

Syntax

```
builtin(function,x1,...,xn)
[y1,...,yn] = builtin(function,x1,...,xn)
```

Description

`builtin(function,x1,...,xn)` executes the built-in `function` with the input arguments `x1` through `xn`. Use `builtin` to execute the original built-in from within a method that overloads the function. To work properly, you must never overload `builtin`.

`[y1,...,yn] = builtin(function,x1,...,xn)` stores any output from `function` in `y1` through `yn`.

Examples

Run an Overloaded Function within a Class Definition

Execute the built-in functionality from within an overloaded method.

Create a simple class describing the speed of a particle and providing a `disp` method by pasting the following code into a file called `MyParticle.m`.

```
classdef MyParticle
    properties
        velocity;
    end
    methods
        function p = MyParticle(x,y,z)
            p.velocity.x = x;
            p.velocity.y = y;
            p.velocity.z = z;
        end
```

```
        function disp(p)
            builtin('disp',p) %call builtin
            if isscalar(p)
                disp('  Velocity')
                disp(['  x: ',num2str(p.velocity.x)])
                disp(['  y: ',num2str(p.velocity.y)])
                disp(['  z: ',num2str(p.velocity.z)])
            end
        end
    end
end
```

Create an instance `MyParticle`.

```
p = MyParticle(1,2,4)
```

```
p =

  MyParticle

  Properties:
    velocity: [1x1 struct]

  Methods

  Velocity
  x: 1
  y: 2
  z: 4
```

Input Arguments

`function` — Built-in function name

string

Valid built-in function name in the MATLAB path, specified as a string. `function` cannot be a function handle.

`x1,...,xn` — Valid input arguments for `function`

supported data types

Valid input arguments for `function`, specified by supported data types.

More About

built-in function

A built-in function is part of the MATLAB executable. MATLAB does not implement these functions in the MATLAB language. Although most built-in functions have a `.m` file associated with them, this file only supplies documentation for the function.

You can use the syntax `which` *`function`* to check whether a function is built-in.

See Also

`feval` | `which`

Introduced before R2006a

bvp4c

Solve boundary value problems for ordinary differential equations

Syntax

```
sol = bvp4c(odefun,bcfun,solinit)
sol = bvp4c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

Arguments

`odefun`	A function handle that evaluates the differential equations $f(x,y)$. It can have the form
	`dydx = odefun(x,y)`
	`dydx = odefun(x,y,parameters)`
	For a scalar `x` and a column vector `y`, `odefun(x,y)` must return a column vector, `dydx`, representing $f(x,y)$. `parameters` is a vector of unknown parameters.

`bcfun`	A function handle that computes the residual in the boundary conditions. For two-point boundary value conditions of the form $bc(y(a),y(b))$, `bcfun` can have the form
	`res = bcfun(ya,yb)`
	`res = bcfun(ya,yb,parameters)`
	where `ya` and `yb` are column vectors corresponding to $y(a)$ and $y(b)$. `parameters` is a vector of unknown parameters. The output `res` is a column vector.
	See "Multipoint Boundary Value Problems" on page 1-808 for a description of `bcfun` for multipoint boundary value problems.

`solinit`	A structure containing the initial guess for a solution. You create `solinit` using the function `bvpinit`. `solinit` has the following fields.

	`x`	Ordered nodes of the initial mesh. Boundary conditions are imposed at a = `solinit.x(1)` and b = `solinit.x(end)`.
	`y`	Initial guess for the solution such that `solinit.y(:,i)` is a guess for the solution at the node `solinit.x(i)`.
	`parameters`	Optional. A vector that provides an initial guess for unknown parameters.
	The structure can have any name, but the fields must be named `x`, `y`, and `parameters`. You can form `solinit` with the helper function `bvpinit`. See `bvpinit` for details.	
`options`	Optional integration argument. A structure you create using the `bvpset` function. See `bvpset` for details.	

Description

`sol = bvp4c(odefun,bcfun,solinit)` integrates a system of ordinary differential equations of the form

$y' = f(x,y)$

on the interval [a,b] subject to two-point boundary value conditions $bc(y(a),y(b)) = 0$.

`odefun` and `bcfun` are function handles. See the `function_handle` reference page for more information.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, as well as the boundary condition function `bcfun`, if necessary.

`bvp4c` can also solve multipoint boundary value problems. See "Multipoint Boundary Value Problems" on page 1-808. You can use the function `bvpinit` to specify the boundary points, which are stored in the input argument `solinit`. See the reference page for `bvpinit` for more information.

The `bvp4c` solver can also find unknown parameters p for problems of the form

$y' = f(x,y,p)$

$0 = bc(y(a),y(b),p)$

where p corresponds to `parameters`. You provide `bvp4c` an initial guess for any unknown parameters in `solinit.parameters`. The `bvp4c` solver returns the final values of these unknown parameters in `sol.parameters`.

`bvp4c` produces a solution that is continuous on `[a,b]` and has a continuous first derivative there. Use the function `deval` and the output `sol` of `bvp4c` to evaluate the solution at specific points `xint` in the interval `[a,b]`.

```
sxint = deval(sol,xint)
```

The structure `sol` returned by `bvp4c` has the following fields:

`sol.x`	Mesh selected by `bvp4c`
`sol.y`	Approximation to $y(x)$ at the mesh points of `sol.x`
`sol.yp`	Approximation to $y'(x)$ at the mesh points of `sol.x`
`sol.parameters`	Values returned by `bvp4c` for the unknown parameters, if any
`sol.solver`	`'bvp4c'`
`sol.stats`	Computational cost statistics (also displayed when the `stats` option is set with `bvpset`).

The structure `sol` can have any name, and `bvp4c` creates the fields `x`, `y`, `yp`, `parameters`, and `solver`.

`sol = bvp4c(odefun,bcfun,solinit,options)` solves as above with default integration properties replaced by the values in `options`, a structure created with the `bvpset` function. See `bvpset` for details.

`solinit = bvpinit(x, yinit, params)` forms the initial guess `solinit` with the vector `params` of guesses for the unknown parameters.

Singular Boundary Value Problems

`bvp4c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form

$y' = S \cdot y/x + F(x,y,p)$
$0 = bc(y(0),y(b),p)$

The interval is required to be [0, b] with $b > 0$. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only $f(x,y,p)$. The boundary conditions must be consistent with the necessary condition $S \cdot y(0) = 0$ and the initial guess should satisfy this condition.

Multipoint Boundary Value Problems

`bvp4c` can solve multipoint boundary value problems where $a = a_0 < a_1 < a_2 < ... < a_n = b$ are boundary points in the interval $[a,b]$. The points $a_1,a_2,...,a_{n-1}$ represent interfaces that divide $[a,b]$ into regions. `bvp4c` enumerates the regions from left to right (from a to b), with indices starting from 1. In region k, $[a_{k-1},a_k]$, `bvp4c` evaluates the derivative as

```
yp = odefun(x,y,k)
```

In the boundary conditions function

```
bcfun(yleft,yright)
```

`yleft(:,k)` is the solution at the left boundary of $[a_{k-1},a_k]$. Similarly, `yright(:,k)` is the solution at the right boundary of region k. In particular,

```
yleft(:,1) = y(a)
```

and

```
yright(:,end) = y(b)
```

When you create an initial guess with

```
solinit = bvpinit(xinit,yinit),
```

use double entries in `xinit` for each interface point. See the reference page for `bvpinit` for more information.

If `yinit` is a function, `bvpinit` calls `y = yinit(x,k)` to get an initial guess for the solution at `x` in region `k`. In the solution structure `sol` returned by `bpv4c`, `sol.x` has

double entries for each interface point. The corresponding columns of `sol.y` contain the left and right solution at the interface, respectively.

To see an example that solves a three-point boundary value problem, type `threebvp` at the MATLAB command prompt.

Note: The `bvp5c` function is used exactly like `bvp4c`, with the exception of the meaning of error tolerances between the two solvers. If $S(x)$ approximates the solution $y(x)$, `bvp4c` controls the residual $|S'(x) - f(x,S(x))|$. This controls indirectly the true error $|y(x) - S(x)|$. `bvp5c` controls the true error directly. `bvp5c` is more efficient than `bvp4c` for small error tolerances.

Examples

Using Initial Guess to Indicate Desired Solution

Boundary value problems can have multiple solutions. One purpose of the initial guess is to indicate which solution, among several, that you want.

The second-order differential equation

$$y'' + |y| = 0$$

has exactly two solutions that satisfy the boundary conditions

$$y(0) = 0,$$

$$y(4) = -2.$$

Before using `bvp4c` to solve the problem, you need to rewrite the differential equation as a system of two first-order ODEs,

$$y_1' = y_2,$$

$$y_2' = -|y_1|,$$

where $y_1 = y$ and $y_2 = y'$. This system has the required form

$$y' = f(x, y),$$

$$bc(y(a), y(b)) = 0.$$

The function, *f*, and the boundary conditions, *bc*, are coded in MATLAB as the functions `twoode` and `twobc`.

```
function dydx = twoode(x,y)
dydx = [ y(2); -abs(y(1)) ];

function res = twobc(ya,yb)
res = [ ya(1); yb(1) + 2 ];
```

Form a guess structure consisting of an initial mesh of five equally spaced points in [0,4] and a guess of the constant values

$$y_1(x) = 1,$$

$$y_2(x) = 0.$$

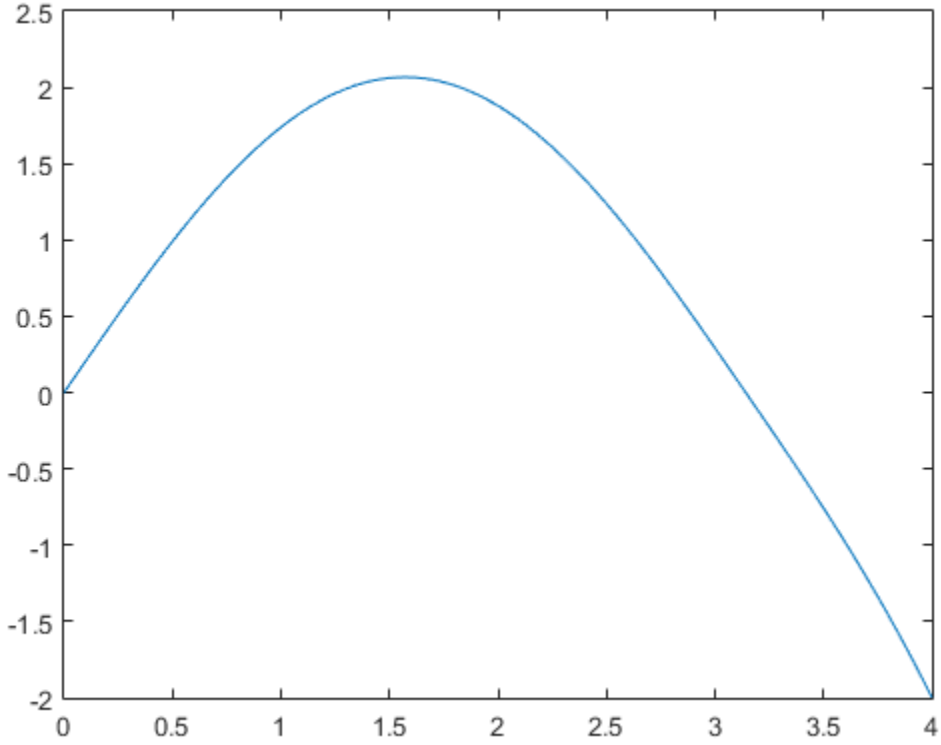
```
solinit = bvpinit(linspace(0,4,5),[1 0]);
```

Solve the problem using `bvp4c`.

```
sol = bvp4c(@twoode,@twobc,solinit);
```

Evaluate the numerical solution at 100 equally spaced points and plot *y(x)*.

```
x = linspace(0,4);
y = deval(sol,x);
plot(x,y(1,:))
```

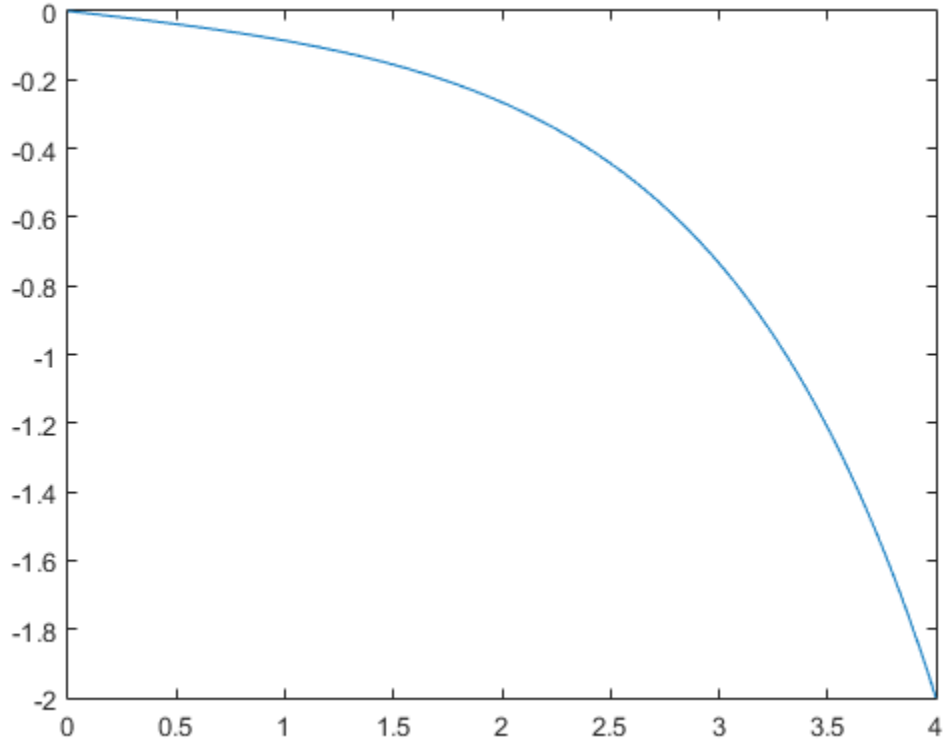


To obtain the other solution of this problem, use the initial guess

$$y_1(x) = -1,$$

$$y_2(x) = 0.$$

```
solinit = bvpinit(linspace(0,4,5),[-1 0]);
sol = bvp4c(@twoode,@twobc,solinit);
x = linspace(0,4);
y = deval(sol,x);
plot(x,y(1,:))
```



Compute Fourth Eigenvalue of Mathieu's Equation

This boundary value problem involves an unknown parameter. The task is to compute the fourth ($q = 5$) eigenvalue λ of Mathieu's equation

$$y'' + (\lambda - 2q\cos 2x)y = 0.$$

Because the unknown parameter λ is present, this second-order differential equation is subject to *three* boundary conditions:

$y'(0) = 0$
$y'(\pi) = 0$
$y(0) = 1$

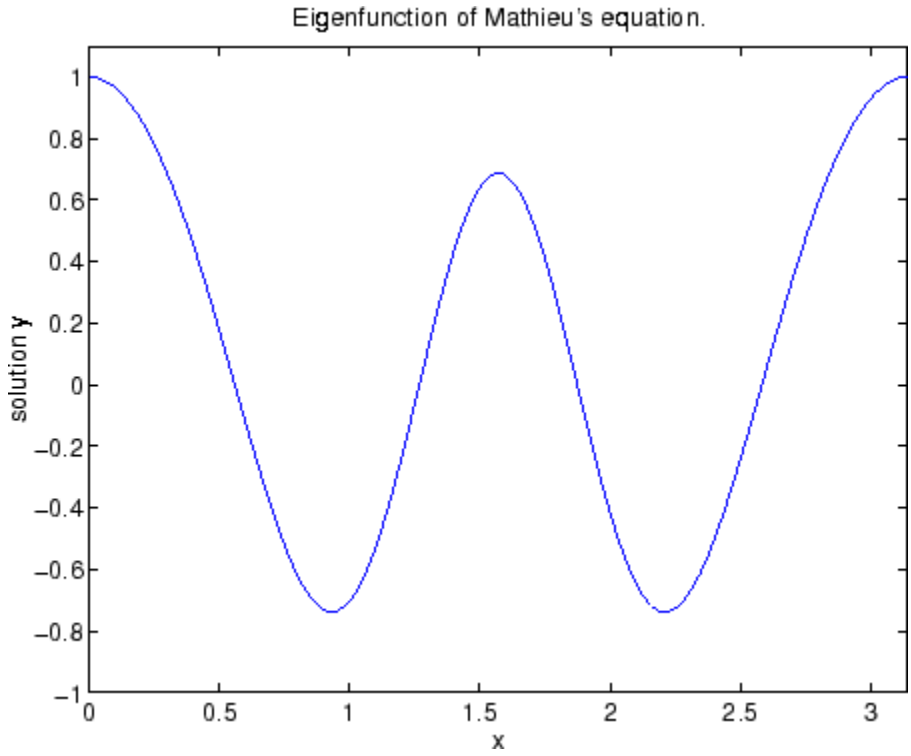
It is convenient to use local functions to place all the functions required by `bvp4c` in a single file.

```
function mat4bvp

lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
sol = bvp4c(@mat4ode,@mat4bc,solinit);

fprintf('The fourth eigenvalue is approximately %7.3f.\n',...
        sol.parameters)

xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
axis([0 pi -1 1.1])
title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')
% --------------------------------------------------------------
function dydx = mat4ode(x,y,lambda)
q = 5;
dydx = [  y(2)
         -(lambda - 2*q*cos(2*x))*y(1) ];
% --------------------------------------------------------------
function res = mat4bc(ya,yb,lambda)
res = [  ya(2)
         yb(2)
        ya(1)-1 ];
% --------------------------------------------------------------
function yinit = mat4init(x)
yinit = [  cos(4*x)
          -4*sin(4*x) ];
```

The differential equation (converted to a first-order system) and the boundary conditions are coded as local functions `mat4ode` and `mat4bc`, respectively. Because unknown parameters are present, these functions must accept three input arguments, even though some of the arguments are not used.

The guess structure `solinit` is formed with `bvpinit`. An initial guess for the solution is supplied in the form of a function `mat4init`. We chose $y = \cos 4x$ because it satisfies the boundary conditions and has the correct qualitative behavior (the correct number of sign changes). In the call to `bvpinit`, the third argument (`lambda = 15`) provides an initial guess for the unknown parameter λ.

After the problem is solved with `bvp4c`, the field `sol.parameters` returns the value $\lambda = 17.097$, and the plot shows the eigenfunction associated with this eigenvalue.



More About

Algorithms

`bvp4c` is a finite difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fourth-order accurate uniformly in `[a,b]`. Mesh selection and error control are based on the residual of the continuous solution.

References

[1] Shampine, L.F., M.W. Reichelt, and J. Kierzenka, “Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c,” available at http://www.mathworks.com/bvp_tutorial

See Also

`@` | `bvp5c` | `bvpget` | `bvpinit` | `bvpset` | `bvpxtend` | `deval`

Introduced before R2006a

bvp5c

Solve boundary value problems for ordinary differential equations

Syntax

```
sol = bvp5c(odefun,bcfun,solinit)
sol = bvp5c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

Arguments

`odefun`	A function handle that evaluates the differential equations $f(x,y)$. It can have the form
	`dydx = odefun(x,y)`
	`dydx = odefun(x,y,parameters)`
	For a scalar `x` and a column vector `y`, `odefun(x,y)` must return a column vector, `dydx`, representing $f(x,y)$. `parameters` is a vector of unknown parameters.	

`bcfun`	A function handle that computes the residual in the boundary conditions. For two-point boundary value conditions of the form $bc(y(a),y(b))$, `bcfun` can have the form
	`res = bcfun(ya,yb)`
	`res = bcfun(ya,yb,parameters)`
	where `ya` and `yb` are column vectors corresponding to $y(a)$ and $y(b)$. `parameters` is a vector of unknown parameters. The output `res` is a column vector.	

`solinit`	A structure containing the initial guess for a solution. You create `solinit` using the function `bvpinit`. `solinit` has the following fields.	
	`x`	Ordered nodes of the initial mesh. Boundary conditions are imposed at a = `solinit.x(1)` and b = `solinit.x(end)`.

	`y`	Initial guess for the solution such that `solinit.y(:,i)` is a guess for the solution at the node `solinit.x(i)`.
	`parameters`	Optional. A vector that provides an initial guess for unknown parameters.
	The structure can have any name, but the fields must be named `x`, `y`, and `parameters`. You can form `solinit` with the helper function `bvpinit`. See `bvpinit` for details.	
`options`	Optional integration argument. A structure you create using the `bvpset` function. See `bvpset` for details.	

Description

`sol = bvp5c(odefun,bcfun,solinit)` integrates a system of ordinary differential equations of the form
$y' = f(x,y)$

on the interval [a,b] subject to two-point boundary value conditions
$bc(y(a),y(b)) = 0$

`odefun` and `bcfun` are function handles. See the `function_handle` reference page for more information.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, as well as the boundary condition function `bcfun`, if necessary. You can use the function `bvpinit` to specify the boundary points, which are stored in the input argument `solinit`.

The `bvp5c` solver can also find unknown parameters p for problems of the form
$y' = f(x,y,p)$
$0 = bc(y(a),y(b),p)$

where p corresponds to `parameters`. You provide `bvp5c` an initial guess for any unknown parameters in `solinit.parameters`. The `bvp5c` solver returns the final values of these unknown parameters in `sol.parameters`.

`bvp5c` produces a solution that is continuous on `[a,b]` and has a continuous first derivative there. Use the function `deval` and the output `sol` of `bvp5c` to evaluate the solution at specific points `xint` in the interval `[a,b]`.

```
sxint = deval(sol,xint)
```

The structure `sol` returned by `bvp5c` has the following fields:

`sol.x`	Mesh selected by `bvp5c`
`sol.y`	Approximation to $y(x)$ at the mesh points of `sol.x`
`sol.parameters`	Values returned by `bvp5c` for the unknown parameters, if any
`sol.solver`	`'bvp5c'`
`sol.stats`	Computational cost statistics (also displayed when the `stats` option is set with `bvpset`).

The structure `sol` can have any name, and `bvp5c` creates the fields `x`, `y`, `parameters`, and `solver`.

`sol = bvp5c(odefun,bcfun,solinit,options)` solves as above with default integration properties replaced by the values in `options`, a structure created with the `bvpset` function. See `bvpset` for details.

`solinit = bvpinit(x, yinit, params)` forms the initial guess `solinit` with the vector `params` of guesses for the unknown parameters.

Singular Boundary Value Problems

`bvp5c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form
$y' = S \cdot y/x + f(x,y,p)$
$0 = bc(y(0),y(b),p)$

The interval is required to be $[0, b]$ with b > 0. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only $f(x,y,p)$. The boundary conditions must be consistent with the necessary condition $S \cdot y(0) = 0$ and the initial guess should satisfy this condition.

Multipoint Boundary Value Problems

`bvp5c` can solve multipoint boundary value problems where $a = a_0 < a_1 < a_2 < ... < a_n = b$ are boundary points in the interval $[a,b]$. The points $a_1,a_2, ... ,a_{n-1}$ represent interfaces

that divide [a,b] into regions. `bvp5c` enumerates the regions from left to right (from a to b), with indices starting from 1. In region k, [a_{k-1},a_k], `bvp5c` evaluates the derivative as

```
yp = odefun(x,y,k)
```

In the boundary conditions function

```
bcfun(yleft,yright)
```

yleft(:, k) is the solution at the left boundary of [a_{k-1},a_k]. Similarly, `yright(:,k)` is the solution at the right boundary of region k. In particular,

```
yleft(:,1) = y(a)
```

and

```
yright(:,end) = y(b)
```

When you create an initial guess with

```
solinit = bvpinit(xinit,yinit),
```

use double entries in `xinit` for each interface point. See the reference page for `bvpinit` for more information.

If `yinit` is a function, `bvpinit` calls `y = yinit(x, k)` to get an initial guess for the solution at `x` in region `k`. In the solution structure `sol` returned by `bvp5c`, `sol.x` has double entries for each interface point. The corresponding columns of `sol.y` contain the left and right solution at the interface, respectively.

To see an example of that solves a three-point boundary value problem, type `threebvp` at the MATLAB command prompt.

More About

Algorithms

`bvp5c` is a finite difference code that implements the four-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fifth-order accurate uniformly in `[a,b]`. The formula is implemented as an implicit Runge-Kutta formula. `bvp5c` solves the algebraic equations directly; `bvp4c`

uses analytical condensation. `bvp4c` handles unknown parameters directly; while `bvp5c` augments the system with trivial differential equations for unknown parameters.

References

[1] Shampine, L.F., M.W. Reichelt, and J. Kierzenka "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c" http://www.mathworks.com/bvp_tutorial.

Note: This tutorial uses the `bvp4c` function, however in most cases the solvers can be used interchangeably.

See Also

`@` | `bvp4c` | `bvpget` | `bvpinit` | `bvpset` | `bvpxtend` | `deval`

bvpget

Extract properties from options structure created with bvpset

Syntax

```
val = bvpget(options,'name')
val = bvpget(options,'name',default)
```

Description

`val = bvpget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = bvpget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = bvpget(options,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `options`.

See Also

`bvp4c` | `bvp5c` | `bvpinit` | `bvpset` | `deval`

Introduced before R2006a

bvpinit

Form initial guess for BVP solvers

Syntax

```
solinit = bvpinit(x,yinit)
solinit = bvpinit(x,yinit,parameters)
solinit = bvpinit(sol,[anew bnew])
solinit = bvpinit(sol,[anew bnew],parameters)
```

Description

`solinit = bvpinit(x,yinit)` forms the initial guess for a boundary value problem solver.

`x` is a vector that specifies an initial mesh. If you want to solve the BVP on $[a,b]$, then specify `x(1)` as a and `x(end)` as b. The solver adapts this mesh to the solution, so a guess like `xb=nlinspace(a,b,10)` often suffices. However, in difficult cases, you should place mesh points where the solution changes rapidly. The entries of `x` must be in

- Increasing order if $a<b$
- Decreasing order if $a>b$

For two-point boundary value problems, the entries of `x` must be distinct. That is, if $a<b$, the entries must satisfy `x(1)` < `x(2)` < ... < `x(end)`. If $a>b$, the entries must satisfy `x(1)` > `x(2)` > ... > `x(end)`

For multipoint boundary value problem, you can specify the points in $[a,b]$ at which the boundary conditions apply, other than the endpoints a and b, by repeating their entries in `x`. For example, if you set

```
x = [0, 0.5, 1, 1,  1.5, 2];
```

the boundary conditions apply at three points: the endpoints `0` and `2`, and the repeated entry 1. In general, repeated entries represent boundary points between regions in $[a,b]$. In the preceding example, the repeated entry 1 divides the interval `[0,2]` into two regions: `[0,1]` and `[1,2]`.

`yinit` is a guess for the solution. It can be either a vector, or a function:

- Vector – For each component of the solution, `bvpinit` replicates the corresponding element of the vector as a constant guess across all mesh points. That is, `yinit(i)` is a constant guess for the `i`th component `yinit(i,:)` of the solution at all the mesh points in `x`.
- Function – For a given mesh point, the guess function must return a vector whose elements are guesses for the corresponding components of the solution. The function must be of the form

  ```
  y = guess(x)
  ```

  where `x` is a mesh point and `y` is a vector whose length is the same as the number of components in the solution. For example, if the guess function is a function, `bvpinit` calls

  ```
  y(:,j) = guess(x(j))
  ```

  at each mesh point.

  For multipoint boundary value problems, the guess function must be of the form

  ```
  y = guess(x, k)
  ```

  where `y` an initial guess for the solution at `x` in region `k`. The function must accept the input argument `k`, which is provided for flexibility in writing the guess function. However, the function is not required to use `k`.

`solinit = bvpinit(x,yinit,parameters)` indicates that the boundary value problem involves unknown parameters. Use the vector `parameters` to provide a guess for all unknown parameters.

`solinit` is a structure with the following fields. The structure can have any name, but the fields must be named `x`, `y`, and `parameters`.

`x`	Ordered nodes of the initial mesh.
`y`	Initial guess for the solution with `solinit.y(:,i)` a guess for the solution at the node `solinit.x(i)`.
`parameters`	Optional. A vector that provides an initial guess for unknown parameters.

`solinit = bvpinit(sol,[anew bnew])` forms an initial guess on the interval `[anew bnew]` from a solution `sol` on an interval $[a,b]$. The new interval must be larger than the previous one, so either `anew <=` a < b `<= bnew` or `anew >=` a > b `>= bnew`. The solution `sol` is extrapolated to the new interval. If `sol` contains `parameters`, they are copied to `solinit`.

`solinit = bvpinit(sol,[anew bnew],parameters)` forms `solinit` as described above, but uses `parameters` as a guess for unknown parameters in `solinit`.

See Also

`@` | `bvp4c` | `bvp5c` | `bvpget` | `bvpset` | `bvpxtend` | `deval`

Introduced before R2006a

bvpset

Create or alter options structure of boundary value problem

Syntax

```
options = bvpset('name1',value1,'name2',value2,...)
options = bvpset(oldopts,'name1',value1,...)
options = bvpset(oldopts,newopts)
bvpset
```

Description

`options = bvpset('name1',value1,'name2',value2,...)` creates a structure `options` that you can supply to the boundary value problem solver `bvp4c`, in which the named properties have the specified values. Any unspecified properties retain their default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property. `bvpset` ignores case for property names.

`options = bvpset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = bvpset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`bvpset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `bvpget` to query the `options` structure for the value of a specific property.

BVP Properties

`bvpset` enables you to specify properties for the boundary value problem solver `bvp4c`. There are several categories of properties that you can set:

Error Tolerance Properties

Because `bvp4c` uses a collocation formula, the numerical solution is based on a mesh of points at which the collocation equations are satisfied. Mesh selection and error control are based on the residual of this solution, such that the computed solution $S(x)$ is the exact solution of a perturbed problem $S'(x) = f(x,S(x)) + res(x)$. On each subinterval of the mesh, a norm of the residual in the `i`th component of the solution, `res(i)`, is estimated and is required to be less than or equal to a tolerance. This tolerance is a function of the relative and absolute tolerances, `RelTol` and `AbsTol`, defined by the user.

$$|| \left(\text{res}(i) / \max\left(\text{abs}\left(f(i) \right), \text{AbsTol}(i) / \text{RelTol} \right) \right) || \le \text{RelTol}$$

The following table describes the error tolerance properties.

BVP Error Tolerance Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the residual vector. It is a measure of the residual relative to the size of $f(x,y)$. The default, `1e-3`, corresponds to 0.1% accuracy. The computed solution $S(x)$ is the exact solution of $S'(x) = F(x,S(x)) + \text{res}(x)$. On each subinterval of the mesh, the residual $\text{res}(x)$ satisfies $$|| \left(\text{res}(i) / \max\left(\text{abs}\left(F(i) \right), \text{AbsTol}(i) / \text{RelTol} \right) \right) || \le \text{RelTol}$$
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the corresponding components of the residual vector. `AbsTol(i)` is a threshold below which the values of the corresponding components are

Property	Value	Description
		unimportant. If a scalar value is specified, it applies to all components.

Vectorization

The following table describes the BVP vectorization property. Vectorization of the ODE function used by `bvp4c` differs from the vectorization used by the ODE solvers:

- For `bvp4c`, the ODE function must be vectorized with respect to the first argument as well as the second one, so that `F([x1 x2 ...],[y1 y2 ...])` returns `[F(x1,y1) F(x2,y2)...]`.
- `bvp4c` benefits from vectorization even when analytical Jacobians are provided. For stiff ODE solvers, vectorization is ignored when analytical Jacobians are used.

Vectorization Properties

Property	Value	Description

`Vectorized`	`on | {off}`	Set on to inform `bvp4c` that you have coded the ODE function F so that `F([x1 x2 ...],[y1 y2 ...])` returns `[F(x1,y1) F(x2,y2) ...]`. That is, your ODE function can pass to the solver a whole array of column vectors at once. This enables the solver to reduce the number of function evaluations and may significantly reduce solution time.
		
		With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. In the `shockbvp` example shown previously, the shockODE function has been vectorized using colon notation into the subscripts and by using the array multiplication (`.*`) operator.
		
		`function dydx = shockODE(x,y,e)`
		`pix = pi*x;`
		`dydx = [ y(2,:)...`
		`-x/e.*y(2,:)-pi^2*cos(pix)-`
		`pix/e.*sin(pix)];`

Analytical Partial Derivatives

By default, the `bvp4c` solver approximates all partial derivatives with finite differences. `bvp4c` can be more efficient if you provide analytical partial derivatives $\partial f/\partial y$ of the differential equations, and analytical partial derivatives, $\partial bc/\partial ya$ and $\partial bc/\partial yb$, of the boundary conditions. If the problem involves unknown parameters, you must also provide partial derivatives, $\partial f/\partial p$ and $\partial bc/\partial p$, with respect to the parameters.

The following table describes the analytical partial derivatives properties.

BVP Analytical Partial Derivative Properties

Property	Value	Description
`FJacobian`	Function handle	A function handle that computes the analytical partial derivatives of $f(x,y)$. When solving $y' = f(x,y)$, set this property to `@fjac` if `dfdy = fjac(x,y)` evaluates the Jacobian $\partial f/\partial y$. If the problem involves unknown parameters p, `[dfdy,dfdp] = fjac(x,y,p)` must also return the partial derivative $\partial f/\partial p$. For problems with constant partial derivatives, set this property to the value of `dfdy` or to a cell array `{dfdy,dfdp}`.
`BCJacobian`	Function handle	A function handle that computes the analytical partial derivatives of $bc(ya,yb)$. For boundary conditions $bc(ya,yb)$, set this property to `@bcjac` if `[dbcdya,dbcdyb] = bcjac(ya,yb)` evaluates the partial derivatives $\partial bc/\partial ya$, and $\partial bc/\partial yb$. If the problem involves unknown parameters p, `[dbcdya,dbcdyb,dbcdp] = bcjac(ya,yb,p)` must also return the partial derivative $\partial bc/\partial p$. For problems with constant partial derivatives, set this property to a cell array `{dbcdya,dbcdyb}` or `{dbcdya,dbcdyb,dbcdp}`.

Singular BVPs

`bvp4c` can solve singular problems of the form

$$y' = S\frac{y}{x} + f(x, y, p)$$

posed on the interval [0,*b*] where $b > 0$. For such problems, specify the constant matrix S as the value of `SingularTerm`. For equations of this form, `odefun` evaluates only the $f(x,y,p)$ term, where p represents unknown parameters, if any.

Singular BVP Property

Property	Value	Description
`SingularTerm`	Constant matrix	Singular term of singular BVPs. Set to the constant matrix S for equations of the form $$y' = S\frac{y}{x} + f(x, y, p)$$ posed on the interval [0,*b*] where $b > 0$.

Mesh Size Property

`bvp4c` solves a system of algebraic equations to determine the numerical solution to a BVP at each of the mesh points. The size of the algebraic system depends on the number of differential equations (`n`) and the number of mesh points in the current mesh (`N`). When the allowed number of mesh points is exhausted, the computation stops, `bvp4c` displays a warning message and returns the solution it found so far. This solution does not satisfy the error tolerance, but it may provide an excellent initial guess for computations restarted with relaxed error tolerances or an increased value of `NMax`.

The following table describes the mesh size property.

BVP Mesh Size Property

Property	Value	Description
`NMax`	positive integer {`floor(1000/n)`}	Maximum number of mesh points allowed when solving the BVP, where `n` is the

Property	Value	Description
		number of differential equations in the problem. The default value of `NMax` limits the size of the algebraic system to about 1000 equations. For systems of a few differential equations, the default value of `NMax` should be sufficient to obtain an accurate solution.

Solution Statistic Property

The `Stats` property lets you view solution statistics.

The following table describes the solution statistics property.

BVP Solution Statistic Property

Property	Value	Description
`Stats`	`on` | {`off`}	Specifies whether statistics about the computations are displayed. If the `stats` property is `on`, after solving the problem, `bvp4c` displays: • The number of points in the mesh • The maximum residual of the solution • The number of times it called the differential equation function `odefun` to evaluate $f(x,y)$ • The number of times it called the boundary condition function `bcfun` to evaluate $bc(y(a),y(b))$

Examples

To create an options structure that changes the relative error tolerance of `bvp4c` from the default value of `1e-3` to `1e-4`, enter

```
options = bvpset('RelTol',1e-4);
```

To recover the value of `'RelTol'` from `options`, enter

```
bvpget(options,'RelTol')

ans =

  1.0000e-004
```

See Also

`function_handle` | `deval` | `bvp4c` | `bvp5c` | `bvpget` | `bvpinit`

Introduced before R2006a

bvpxtend

Form guess structure for extending boundary value solutions

Syntax

```
solinit = bvpxtend(sol,xnew,ynew)
solinit = bvpxtend(sol,xnew,extrap)
solinit = bvpxtend(sol,xnew)
solinit = bvpxtend(sol,xnew,ynew,pnew)
solinit = bvpxtend(sol,xnew,extrap,pnew)
```

Description

`solinit = bvpxtend(sol,xnew,ynew)` uses solution `sol` computed on `[a,b]` to form a solution guess for the interval extended to `xnew`. The extension point `xnew` must be outside the interval `[a,b]`, but on either side. The vector `ynew` provides an initial guess for the solution at `xnew`.

`solinit = bvpxtend(sol,xnew,extrap)` forms the guess at `xnew` by extrapolating the solution `sol`. `extrap` is a string that determines the extrapolation method. `extrap` has three possible values:

- `'constant'` — `ynew` is a value nearer to end point of solution in `sol`.
- `'linear'` — `ynew` is a value at `xnew` of linear interpolant to the value and slope at the nearer end point of solution in `sol`.
- `'solution'` — `ynew` is the value of (cubic) solution in `sol` at `xnew`.

The value of `extrap` is case-insensitive and only the leading, unique portion needs to be specified.

`solinit = bvpxtend(sol,xnew)` uses the extrapolating solution where `extrap` is `'constant'`. If there are unknown parameters, values present in `sol` are used as the initial guess for parameters in `solinit`.

`solinit = bvpxtend(sol,xnew,ynew,pnew)` specifies a different guess `pnew`. `pnew` can be used with extrapolation, using the syntax `solinit =`

`bvpxtend(sol,xnew,extrap,pnew)`. To modify parameters without changing the interval, use `[]` as place holder for `xnew` and `ynew`.

See Also

`bvp4c` | `bvp5c` | `bvpinit`

caldays

Calendar duration in days

Syntax

```
D = caldays(X)
```

Description

`D = caldays(X)` returns an array representing calendar days equivalent to the values in array `X`.

- If `X` is a numeric array, then `D` is a `calendarDuration` array with each element equal to the number of calendar days in the corresponding element of `X`. Calendar days account for Daylight Saving Time shifts when used in calendar calculations.
- If `X` is a `calendarDuration` array, then `D` is a `double` array with each element equal to the number of whole calendar days in the corresponding element of `X`.

Examples

Create Array of Calendar Days

```
X = magic(5);
D = caldays(X)
```

```
D =

   17d   24d    1d    8d   15d
   23d    5d    7d   14d   16d
    4d    6d   13d   20d   22d
   10d   12d   19d   21d    3d
   11d   18d   25d    2d    9d
```

Convert Calendar Durations to Calendar Days

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar days.

```
X = caldays(8:10) + hours(1.2345)
```

```
X =

    8d 1h 14m 4.2s    9d 1h 14m 4.2s   10d 1h 14m 4.2s
```

```
D = caldays(X)
```

```
D =

     8     9    10
```

`caldays` returns a numeric array.

Current Time at Future Date

Add two calendar days to the current date and time.

```
t = datetime('now') + caldays(2)
```

```
t =

   25-Feb-2015 10:05:24
```

Create Sequence of Dates

Create a sequence of consecutive dates beginning on March 18, 2014.

```
T = datetime([2014,03,18]) + caldays(0:4)
```

```
T =
```

```
   18-Mar-2014   19-Mar-2014   20-Mar-2014   21-Mar-2014   22-Mar-2014
```

Create a sequence of dates beginning on March 18, 2014, spaced 2 days apart.

```
T = datetime([2014,03,18]) + caldays(0:2:8)
```

```
T =

   18-Mar-2014   20-Mar-2014   22-Mar-2014   24-Mar-2014   26-Mar-2014
```

Input Arguments

X — Input array

numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If `X` is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

More About

Tips

- `caldays` creates days that account for Daylight Saving Time shifts when used in calendar calculations. To create exact fixed-length (24 hour) days, use the `days` function.

See Also

`calendarDuration` | `days` | `hours`

Introduced in R2014b

caldiff

Calendar math successive differences

Syntax

```
dt = caldiff(t)
dt = caldiff(t,components)
dt = caldiff(t,components,dim)
```

Description

`dt = caldiff(t)` calculates time differences between adjacent datetime values in `t` in terms of the calendar components years, months, days, and time. `caldiff` calculates differences along the first array dimension whose size does not equal 1.

- If `t` is a vector of length `m`, then `dt = caldiff(t)` returns a vector of length `m-1`. The elements of `dt` are the differences between adjacent elements of `t`.

  ```
  dt = [between(t(1),t(2)), between(t(2),t(3)),..., between(t(m-1),t(m))]
  ```

- If `t` is a nonvector `p`-by-`m` matrix, then `dt = caldiff(t)` returns a matrix of size `(p-1)`-by-`m`, whose elements are the differences between the rows of `t`.

  ```
  dt(:,I) = [between(t(1,I),t(2,I), between(t(2,I),t(3,I)), ...,
       between(t(p-1,I),t(p,I))]
  ```

`dt = caldiff(t,components)` finds the differences between successive datetimes in `t` in terms of the specified calendar or time components.

`dt = caldiff(t,components,dim)` finds the differences between successive datetimes along the dimension specified by `dim`.

Examples

Calendar Differences Between Datetime Values

Create a `datetime` array and then compute the differences between the values in terms of calendar components.

```
t = [datetime('yesterday');datetime('today');datetime('tomorrow')]

t =

   22-Feb-2015
   23-Feb-2015
   24-Feb-2015

D = caldiff(t)

D =

   1d
   1d
```

Differences Using Specific Calendar Components

Create a `datetime` array and then compute the differences between the values in terms of days.

```
t = datetime('now') + calmonths(0:3)

t =

Columns 1 through 3

   23-Feb-2015 09:54:57   23-Mar-2015 09:54:57   23-Apr-2015 09:54:57

Columns 4 through 4

   23-May-2015 09:54:57

D = caldiff(t,'days')

D =

   28d   31d   30d
```

Computer the differences between the datetime values in terms of weeks and days.

```
D = caldiff(t,{'weeks','days'})
```

```
D =

       4w    4w 3d    4w 2d
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`components` — Calendar or time components

`'years'` | `'quarters'` | `'months'` | `'weeks'` | `'days'` | `'time'` | cell array of strings

Calendar or time components, specified as one of the following strings, or a cell array containing one ore more of these strings.

- `'years'`
- `'quarters'`
- `'months'`
- `'weeks'`
- `'days'`
- `'time'`

Except for `'time'`, the above components are flexible lengths of time. For example, one month represents a different length of time when added to a datetime in January than when added to a datetime in February.

`caldiff` operates on the calendar or time components in decreasing order, starting with the largest component.

In general, `t(2:m)` is not equal to `t(1:m-1) + dt`, unless you include `'time'` in `components`.

Example: `{'years','quarters'}`

Data Types: `char` | `cell`

`dim` — Dimension to operate along

positive integer

Dimension to operate along, specified as a positive integer. If no value is specified, the default is the first array dimension whose size does not equal 1.

Output Arguments

`dt` — Difference array

scalar | vector | matrix | multidimensional array

Difference array, returned as a scalar, vector, matrix, or multidimensional `calendarDuration` array.

More About

Tips

- To compute successive differences between datetimes in `t1` and `t2` as exact, fixed-length units of hours, minutes, and seconds, use `diff(t)`.

See Also

`between` | `calendarDuration` | `diff` | `minus`

Introduced in R2014b

calendar

Calendar for specified month

Syntax

```
c = calendar
c = calendar(d)
c = calendar(y, m)
```

Description

`c = calendar` returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.

`c = calendar(d)`, where `d` is a serial date number or a date string, returns a calendar for the specified month.

`c = calendar(y, m)`, where `y` and `m` are integers, returns a calendar for the specified month of the specified year.

If you do not specify an output argument, then `calendar` displays a calendar in the Command Window but does not return a value.

Examples

The command

```
calendar(1957,10)
```

reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

```
                Oct 1957
    S     M     Tu    W     Th    F     S
    0     0     1     2     3     4     5
    6     7     8     9     10    11    12
```

```
    13    14    15    16    17    18    19
    20    21    22    23    24    25    26
    27    28    29    30    31     0     0
     0     0     0     0     0     0     0
```

See Also

`datenum`

Introduced before R2006a

calendarDuration

Create calendar duration array from numeric values

The `calendarDuration` function creates an array that represents elapsed time in calendar units of variable length. For example, the number of days in 1 month depend on the particular month of the year. Calculations with calendar durations account for daylight saving time changes and leap years.

Syntax

```
L = calendarDuration(Y,M,D)
L = calendarDuration(Y,M,D,H,MI,S)
L = calendarDuration(Y,M,D,T)

L = calendarDuration( ___ ,'Format',displayFormat)
```

Description

`L = calendarDuration(Y,M,D)` creates an array of calendar durations from numeric arrays containing the number of years, months, and days specified by `Y`, `M`, and `D`, respectively. Alternatively, you can specify the input arguments as a matrix with three columns, `[Y,M,D]`.

`L = calendarDuration(Y,M,D,H,MI,S)` also includes the hours, minutes, and seconds specified by `H`, `MI`, and `S`, respectively. Alternatively, you can specify the input arguments as a matrix with six columns, `[Y,M,D,H,MI,S]`.

`L = calendarDuration(Y,M,D,T)` creates an array of calendar durations from numeric arrays containing the number of years, months, and days, and a duration array `T` containing elapsed times.

`L = calendarDuration( ___ ,'Format',displayFormat)` additionally specifies the format in which `L` displays, using any of the input arguments from the previous syntaxes.

Examples

Create Calendar Duration from Numeric Scalars

```
L = calendarDuration(1,3,15)
```

```
L =

   1y 3mo 15d
```

Create Calendar Duration from Year, Month, Day, and Duration Arrays

Create a 2-by-2 array representing numbers of days.

```
D = magic(2)
```

```
D =

     1     3
     4     2
```

Create a 2-by-2 `duration` array of hours.

```
T = hours([1 2; 25 12])
```

```
T =

     1 hr    2 hrs
   25 hrs   12 hrs
```

Create a `calendarDuration` array using scalar values for the year and month inputs.

```
L = calendarDuration(1,13,D,T)
```

```
L =

    2y 1mo 1d 1h 0m 0s    2y 1mo 3d 2h 0m 0s
   2y 1mo 4d 25h 0m 0s   2y 1mo 2d 12h 0m 0s
```

Month values greater than 12 carry over to years in the display. Hour values greater than 24 do not carry over to days in the display, because the number of hours in a calendar day is not necessarily 24 hours.

Specify Display Format of Calendar Duration

Create a `calendarDuration` array and specify a format that displays the values in terms of months, weeks, days, and time.

```
L = calendarDuration(1,1,5:9,12,0,0,'Format','mwdt')
```

```
L =

Columns 1 through 3

      13mo 5d 12h 0m 0s      13mo 6d 12h 0m 0s      13mo 1w 12h 0m 0s

Columns 4 through 5

   13mo 1w 1d 12h 0m 0s   13mo 1w 2d 12h 0m 0s
```

Input Arguments

`Y,M,D` — Year, month, and day values

scalar | vector | matrix | multidimensional array

Year, month, and day values, specified as scalars, vectors, matrices, or multidimensional arrays. These values must be the same size, or any can be a scalar. `Y,M,D` should contain only integer values.

If `Y,M,D` are all scalars or all column vectors, you can specify the input arguments as a matrix with three columns, `[Y,M,D]`.

Specifying month values greater than 12 is equivalent to a number of years plus a number of months. For example, 25 months are equal to 2 years and 1 month. However, day values are not equivalent to a number of months, because the number of days in a month is not fixed, and cannot be determined until you add the calendar duration to a specific datetime.

Example: `2,10,24`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y,M,D,H,MI,S` — Year, month, day, hour, minute, and second values

scalar | vector | matrix | multidimensional array

Year, month, day, hour, minute and second values, specified as scalars, vectors, matrices, or multidimensional arrays. These values must be the same size, or any can be a scalar. Specify fractional seconds as part of the seconds input, `S`. The `Y,M,D,H,MI` inputs should contain only integer values.

If `Y,M,D,H,MI,S` are all scalars or all column vectors, you can specify the input arguments as a matrix with six columns, `[Y,M,D,H,MI,S]`.

Specifying month values greater than 12 is equivalent to a number of years plus a number of months. For example, 25 months are equal to 2 years and 1 month. Minute values greater than 60 carry over to a number of hours, and second values greater than 60 carry over to a number of minutes. However, day values are not equivalent to a number of months because the number of days in a month is not fixed and cannot be determined until you add the calendar duration to a specific datetime. Similarly, hour values are not equivalent to a number of calendar days.

Example: `2,10,24,12,45,07.451`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y,M,D,T` — Year, month, day, and duration values

scalar | vector | matrix | multidimensional array

Year, month, day and duration values, specified as scalars, vectors, matrices, or multidimensional arrays. These values must be the same size, or any can be a scalar. `Y`, `M`, and `D` are numeric arrays, and `T` is a `duration` array containing elapsed times.

Specifying month values greater than 12 is equivalent to a number of years plus a number of months. For example, 25 months are equal to 2 years and 1 month. However, day values are not equivalent to a number of months and duration values are not equivalent to a number of days. The number of days in a month and the number of hours in a day are not fixed and cannot be determined until you add the calendar duration to a specific datetime.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`displayFormat` — Display format

`'ymdt'` (default) | string

Display format of the values in `L`, specified as a string that can include the characters `y`, `q`, `m`, `w`, `d`, and `t`, in this order. The string must include `m` to display the number of months, `d` to display the number of days, and `t` to display the number of hours, minutes, and seconds. The `y` (year), `q` (quarter), and `w` (week) characters are optional.

This table describes the date and time components that the characters represent.

Character	Date or Time Unit	Details
`y`	Years	Multiples of 12 months display as a number of years
`q`	Quarters	Multiples of 3 months display as a number of quarters
`m`	Months	Must be included in `displayFormat`
`w`	Weeks	Multiples of 7 days display as a number of weeks
`d`	Days	Must be included in `displayFormat`
`t`	Time (hours, minutes, and seconds)	Must be included in `displayFormat`

If the value of a date or time component is zero, it is not displayed.

Changing the display format of a `calendarDuration` array does not change the values in the array.

Example: `'yqmdt'`

Output Arguments

`L` — Calendar duration

`calendarDuration` array

Calendar duration, returned as a `calendarDuration` array.

You can change the display format of a calendar duration by modifying the `Format` property. For example, to display the calendar duration, `L`, as a number of months, type:

```
L.Format = 'm'
```

More About

Tips

- When you add a `calendarDuration` array that contains more than one unit to a datetime, MATLAB always adds the larger units first. If `t` is a datetime, then this command:

  ```
  t + calendarDuration(1,2,3)
  ```
  is the same as:

  ```
  t + calyears(1) + calmonths(2) + caldays(3)
  ```

- "Represent Dates and Times in MATLAB"

See Also

`duration`

Introduced in R2014b

calllib

Call function in shared library

Syntax

```
[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)
```

Description

`[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)` calls function, funcname, in library, libname, passing input arguments, arg1,...,argN, and returns output values obtained from funcname in x1,...,xN.

Examples

Call `addStructByRef` Function

Load the library.

```
if ~libisloaded('shrlibsample')
   addpath(fullfile(matlabroot,'extern','examples','shrlib'))
   loadlibrary('shrlibsample')
end
```

Display function signature.

```
libfunctionsview shrlibsample
```

```
[double, c_structPtr] addStructByRef(c_structPtr)
```

The input argument is a pointer to a `c_struct` data type.

Create a MATLAB structure, `struct`:

```
struct.p1 = 4; struct.p2 = 7.3; struct.p3 = -290;
```

Call the function.

```
[res,st] = calllib('shrlibsample','addStructByRef',struct);
```

Display the results.

```
res
```

```
res =
          -279
```

Cleanup.

```
unloadlibrary shrlibsample
```

Input Arguments

`libname` — Name of shared library

string

Name of shared library, specified as a string. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

`funcname` — Name of function in library

string

Name of function in library, specified as a string.

Data Types: `char`

`arg1,...,argN` — Input arguments

any type

Input arguments, 1 through N, required by `funcname` (if any), specified by any type. The `funcname` argument list specifies the argument type.

Output Arguments

`x1,...,xN` — Output arguments

any type

Output arguments, 1 through N, from `funcname` (if any), returned as any type. The `funcname` argument list specifies the argument type.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

More About

- Passing Arguments
- "MATLAB Crashes Calling Function in Shared Library"

See Also

`libfunctionsview` | `loadlibrary`

Introduced before R2006a

callSoapService

Send SOAP (Simple Object Access Protocol) message to endpoint

Compatibility

`callSoapService` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
response = callSoapService(endpoint,soapAction,message)
```

Description

`response = callSoapService(endpoint,soapAction,message)` sends message to the soapAction service at endpoint.

Examples

Retrieve Book Information from Library Database

This example assumes the library is on a local intranet and does not use an actual endpoint; therefore, you cannot run it.

Retrieve the name of the author of a book titled “In the Fall.” The relative path of the library service is `urn:LibraryCatalog`. To get the author's name, use the `getAuthor` function, which takes the book name as the input value. The `getAuthor` parameter is `nameToLookUp`. The XML data type for title is `{http://www.w3.org/2001/XMLSchema}string`. The SOAP message style is `rpc` by default.

Create the SOAP message.

```
message = createSoapMessage(...
```

```
  'urn:LibraryCatalog',...
  'getAuthor',...
  {'In the Fall'},...
  {'nameToLookUp'},...
  {'{http://www.w3.org/2001/XMLSchema}string'})

message =

[#document: null]
```

This response does not necessarily indicate that the message is valid, although certain input problems produce error messages.

Send the message to the server for processing, and get the author's name back. The server endpoint is `http://test/soap/services/LibraryCatalog`. The server method is `urn:LibraryCatalog#getAuthor`.

```
response = callSoapService(...
  'http://test/soap/services/LibraryCatalog',...
  'urn:LibraryCatalog#getAuthor',...
  message)

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<getAuthorResponse xmlns="urn:LibraryCatalog">
<ns1:getAuthorReturn xmlns:ns1="http://latestversion.soap.test">
Kate Alvin
</ns1:getAuthorReturn>
</getAuthorResponse>
</soapenv:Body>
</soapenv:Envelope>
```

MATLAB returns the message in a single line, displayed here on separate lines for legibility.

Extract the author's name.

```
author = parseSoapResponse(response)

author = Kate Alvin
```

MATLAB automatically converted the XML string data type to `char`.

Input Arguments

`endpoint` — URL identifying a built-in HTTP service

string

URL identifying a built-in HTTP service, specified as a string.

Example: `'http://test/soap/services/LibraryCatalog'`

`soapAction` — Name of service

string

Name of service, specified as a string.

Example: `'urn:LibraryCatalog#getAuthor'`

`message` — Java® document object model (DOM)

string

Java document object model (DOM), specified as a string. Use `createSoapMessage` to create message.

Output Arguments

`response` — Result of `soapAction`

string

Result of `soapAction`, returned as a string. To read information in response, use the `parseSoapResponse` function.

See Also

`createSoapMessage` | `matlab.wsdl.createWSDLClient` | `parseSoapResponse` | `urlread` | `xmlread`

Introduced before R2006a

calmonths

Calendar duration in months

Syntax

```
M = calmonths(X)
```

Description

`M = calmonths(X)` returns an array representing calendar months equivalent to the values in `X`.

- If `X` is a numeric array, then `M` is a `calendarDuration` array with each element equal to the number of calendar months in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `M` is a `double` array with each element equal to the number of whole calendar months in the corresponding element of `X`.

Examples

Create Array of Calendar Months

```
X = magic(4);
M = calmonths(X)
```

```
M =

   1y 4mo      2mo      3mo   1y 1mo
      5mo     11mo     10mo      8mo
      9mo      7mo      6mo       1y
      4mo   1y 2mo   1y 3mo      1mo
```

Convert Calendar Durations to Calendar Months

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar months.

```
X = calmonths(15:17) + caldays(8) + hours(1.2345)

X =

   1y 3mo 8d 1h 14m 4.2s   1y 4mo 8d 1h 14m 4.2s   1y 5mo 8d 1h 14m 4.2s

M = calmonths(X)

M =

    15    16    17
```

Input Arguments

X — Input array

numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If `X` is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

See Also

`calendarDuration` | `calweeks`

Introduced in R2014b

calquarters

Calendar duration in quarters

Syntax

```
Q = calquarters(X)
```

Description

`Q = calquarters(X)` returns an array representing calendar quarters equivalent to the values in array `X`.

- If `X` is a numeric array, then `Q` is a `calendarDuration` array with each element equal to the number of calendar quarters in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `Q` is a `double` array with each element equal to the number of whole calendar quarters in the corresponding element of `X`.

  ```
  Q = fix(calmonths(t)/3)
  ```

Examples

Create Array of Calendar Quarters

```
X = magic(4);
Q = calquarters(X)
```

```
Q =

   16q    2q    3q   13q
    5q   11q   10q    8q
    9q    7q    6q   12q
    4q   14q   15q    1q
```

Convert Calendar Durations to Quarters

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar quarters.

```
X = calmonths(2:2:6) + caldays(8)
```

```
X =

   2mo 8d   4mo 8d   6mo 8d
```

```
Q = calquarters(X)
```

```
Q =

     0     1     2
```

Input Arguments

X — Input array

numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If `X` is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

See Also

`calendarDuration` | `calmonths`

Introduced in R2014b

calweeks

Calendar duration in weeks

Syntax

```
W = calweeks(X)
```

Description

`W = calweeks(X)` returns an array representing calendar weeks equivalent to the values in `X`.

- If `X` is a numeric array, then `W` is a `calendarDuration` array with each element equal to the number of calendar weeks in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `calweeks` returns the number of whole weeks equivalent to each calendar duration in `X`.

Examples

Create Array of Calendar Weeks

```
X = magic(4);
W = calweeks(X)
```

```
W =

   16w    2w    3w   13w
    5w   11w   10w    8w
    9w    7w    6w   12w
    4w   14w   15w    1w
```

Convert Calendar Durations to Calendar Weeks

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar weeks.

```
X = caldays(15:17) + hours(1.2345)

X =

   15d 1h 14m 4.2s   16d 1h 14m 4.2s   17d 1h 14m 4.2s

W = calweeks(X)

W =

     2     2     2
```

Input Arguments

X — Input array

numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If `X` is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

Output Arguments

W — Calendar weeks

scalar | vector | matrix | multidimensional array

Calendar weeks, returned as a scalar, vector, matrix, or multidimensional array. `W` is the same size as `X`. The data type of `W` depends on `X`.

- If `X` is a numeric array, then `W` is an array of calendar durations in units of equivalent flexible-length calendar weeks.
- If `X` is a `calendarDuration` array, then `W` is a `double` array of integer values representing whole calendar weeks.

See Also

`caldays` | `calendarDuration`

Introduced in R2014b

calyears

Calendar duration in years

Syntax

```
Y = calyears(X)
```

Description

`Y = calyears(X)` returns an array representing calendar years equivalent to the values in `X`. Calendar years account for leap days when used in calendar calculations.

- If `X` is a numeric array, then `Y` is a `calendarDuration` array with each element equal to the number of calendar years in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `calyears` returns the number of whole years equivalent to each calendar duration in `X`.

Examples

Create Array of Calendar Years

```
X = magic(4);
Y = calyears(X)
```

```
Y =

   16y    2y    3y   13y
    5y   11y   10y    8y
    9y    7y    6y   12y
    4y   14y   15y    1y
```

Convert Calendar Durations to Calendar Years

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar years.

```
X = calmonths(21:25) + caldays(8)

X =

    1y 9mo 8d   1y 10mo 8d   1y 11mo 8d        2y 8d    2y 1mo 8d

Y = calyears(X)

Y =

     1     1     1     2     2
```

Input Arguments

X — Input array

numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If `X` is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

Output Arguments

Y — Calendar years

scalar | vector | matrix | multidimensional array

Calendar years, returned as a scalar, vector, matrix, or multidimensional array. `Y` is the same size as `X`. The data type of `Y` depends on `X`.

- If `X` is a numeric array, then `Y` is an array of calendar durations in units of equivalent flexible-length calendar years.
- If `X` is a `calendarDuration` array, then `Y` is a `double` array of integer values representing whole calendar years.

More About

Tips

- `calyears` creates years that account for leap days when used in calendar calculations. To create exact fixed-length (365.2425 day) years, use the `years` function.

See Also

`calendarDuration` | `calmonths` | `years`

Introduced in R2014b

camdolly

Move camera position and target

Syntax

```
camdolly(dx,dy,dz)
camdolly(dx,dy,dz,'targetmode')
camdolly(dx,dy,dz,'targetmode','coordsys')
camdolly(axes_handle,...)
```

Description

`camdolly(dx,dy,dz)` moves the camera position and the camera target by the specified amounts `dx`, `dy`, and `dz`.

`camdolly(dx,dy,dz,'targetmode')` uses the `targetmode` argument to determine how the camera moves:

- `movetarget` (default) — Move both the camera and the target.
- `fixtarget` — Move only the camera.

`camdolly(dx,dy,dz,'targetmode','coordsys')` uses the `coordsys` argument to determine how MATLAB interprets `dx`, `dy`, and `dz`:

- `camera` (default) — Move in the coordinate system of the camera. `dx` moves left/right, `dy` moves down/up, and `dz` moves along the viewing axis. MATLAB normalizes the units to the scene.

  For example, setting `dx` to 1 moves the camera to the right, which pushes the scene to the left edge of the box formed by the axes position rectangle. A negative value moves the scene in the other direction. Setting `dz` to 0.5 moves the camera to a position halfway between the camera position and the camera target.
- `pixels` — Interpret `dx` and `dy` as pixel offsets and ignore `dz`.
- `data` — Interpret `dx`, `dy`, and `dz` as offsets in axes data coordinates.

`camdolly(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camdolly` operates on the current axes.

`camdolly` sets the axes `CameraPosition` and `CameraTarget` properties, which in turn sets the `CameraPositionMode` and `CameraTargetMode` properties to `manual`.

Examples

Move the camera along the *x*- and *y*-axes in a series of steps:

```
surf(peaks)
axis vis3d
t = 0:pi/20:4*pi;
dx = sin(t)./40;
dy = cos(t)./40;
for i = 1:length(t);
    camdolly(dx(i),dy(i),0)
    drawnow
end
```

More About

- "Camera Graphics Terminology"

See Also

`axes` | `campos` | `camproj` | `camup` | `camva` | `camtarget`

Introduced before R2006a

cameratoolbar

Control camera toolbar programmatically

Syntax

```
cameratoolbar
cameratoolbar('NoReset')
cameratoolbar('SetMode',mode)
cameratoolbar('SetCoordSys',coordsys)
cameratoolbar('Show')
cameratoolbar('Hide')
cameratoolbar('Toggle')
cameratoolbar('ResetCameraAndSceneLight')
cameratoolbar('ResetCamera')
cameratoolbar('ResetSceneLight')
cameratoolbar('ResetTarget')
h = cameratoolbar(...)
mode = cameratoolbar('GetMode')
paxis = cameratoolbar('GetCoordsys')
tf = cameratoolbar('GetVisible')
cameratoolbar('Close')
cameratoolbar(fig,...)
```

Description

`cameratoolbar` creates a toolbar that enables interactive manipulation of the axes camera and light when you drag the mouse on the figure window. Several axes camera properties are set when the toolbar is initialized.

`cameratoolbar('NoReset')` creates the toolbar without setting any camera properties.

`cameratoolbar('SetMode',mode)` sets the toolbar mode (depressed button). `mode` can be `'orbit'`, `'orbitscenelight'`, `'pan'`, `'dollyhv'`, `'dollyfb'`, `'zoom'`, `'roll'`, `'nomode'`. For descriptions of the various modes, see “Camera Toolbar”. You can also set these modes using the toolbar, by clicking the respective buttons.

`cameratoolbar('SetCoordSys',coordsys)` sets the principal axis of the camera motion. `coordsys` can be: `'x'`, `'y'`, `'z'`, `'none'`.

`cameratoolbar('Show')` shows the toolbar on the current figure.

`cameratoolbar('Hide')` hides the toolbar on the current figure.

`cameratoolbar('Toggle')` toggles the visibility of the toolbar.

`cameratoolbar('ResetCameraAndSceneLight')` resets the current camera and scenelight.

`cameratoolbar('ResetCamera')` resets the current camera.

`cameratoolbar('ResetSceneLight')` resets the current scenelight.

`cameratoolbar('ResetTarget')` resets the current camera target.

`h = cameratoolbar(...)` returns the handle to the toolbar.

`mode = cameratoolbar('GetMode')` returns the current mode.

`paxis = cameratoolbar('GetCoordsys')` returns the current principal axis.

`tf = cameratoolbar('GetVisible')` returns the visibility of the toolbar (`1` if visible, `0` if not visible).

`cameratoolbar('Close')` removes the toolbar from the current figure.

`cameratoolbar(fig,...)` specifies the figure to operate on by passing the figure handle as the first argument.

In general, the use of OpenGL hardware improves rendering performance.

Alternatives

Display the toolbar by selecting **Camera Toolbar** from the figure window's **View** menu.

More About

- "Camera Toolbar"

See Also

`rotate3d` | `zoom`

Introduced before R2006a

camlight

Create or move light object in camera coordinates

Syntax

```
camlight('headlight')
camlight('right')
camlight('left')
camlight
camlight(az,el)
camlight(...,'style')
camlight(light_handle,...)
light_handle = camlight(...)
```

Description

`camlight('headlight')` creates a light at the camera position.

`camlight('right')` creates a light right and up from camera.

`camlight('left')` creates a light left and up from camera.

`camlight` with no arguments is the same as `camlight('right')`.

`camlight(az,el)` creates a light at the specified azimuth (`az`) and elevation (`el`) with respect to the camera position. The camera target is the center of rotation and `az` and `el` are in degrees.

`camlight(...,'style')` defines the style argument using one of two values:

- `local` (default) — The light is a point source that radiates from the location in all directions.
- `infinite` — The light shines in parallel rays.

`camlight(light_handle,...)` uses the light specified in `light_handle`.

`light_handle = camlight(...)` returns the light object handle.

`camlight` sets the light object `Position` and `Style` properties. A light created with `camlight` does not track the camera. In order for the light to stay in a constant position relative to the camera, call `camlight` whenever you move the camera.

Examples

Create a light positioned to the left of the camera and then reposition the light each time the camera moves:

```
surf(peaks)
axis vis3d
h = camlight('left');
for i = 1:20;
   camorbit(10,0)
   camlight(h,'left')
   pause(.1)
end
```

More About

- "Lighting Overview"

See Also

`lightangle` | `light`

Introduced before R2006a

camlookat

Position camera to view object or group of objects

Syntax

```
camlookat(object_handles)
camlookat(axes_handle)
camlookat
```

Description

`camlookat(object_handles)` views the objects identified in the vector `object_handles`. The vector can contain the handles of axes `Children`.

`camlookat(axes_handle)` views the objects that are children of the axes identified by `axes_handle`.

`camlookat` views the objects that are in the current axes by moving the camera position and camera target while preserving the relative view direction and camera view angle. The viewed object (or objects) roughly fill the axes position rectangle. To change the view, `camlookat` sets the axes `CameraPosition` and `CameraTarget` properties.

Examples

Create three spheres at different locations and then progressively position the camera so that the scene composes around each sphere individually:

```
% Create three spheres using the sphere function:
[x y z] = sphere;
s1 = surf(x,y,z);
hold on
s2 = surf(x+3,y,z+3);
s3 = surf(x,y,z+6);
% Set the data aspect ratio using daspect:
daspect([1 1 1])
% Set the view:
```

```
view(30,10)
% Set the projection type using camproj:
camproj perspective
% Compose the scene around the current axes
camlookat(gca)
pause(2)
% Compose the scene around sphere s1
camlookat(s1)
pause(2)
% Compose the scene around sphere s2
camlookat(s2)
pause(2)
% Compose the scene around sphere s3
camlookat(s3)
pause(2)
camlookat(gca)
```

More About

- “Camera Graphics Terminology”

See Also

`campos` | `camtarget`

Introduced before R2006a

camorbit

Rotate camera position around camera target

Syntax

```
camorbit(dtheta,dphi)
camorbit(dtheta,dphi,'coordsys')
camorbit(dtheta,dphi,'coordsys','direction')
camorbit(axes_handle,...)
```

Description

`camorbit(dtheta,dphi)` rotates the camera position around the camera target by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`camorbit(dtheta,dphi,'coordsys')` rotates the camera position around the camera target, using the *`coordsys`* argument to determine the center of rotation. *`coordsys`* can take on two values:

- `data` (default) — Rotate the camera around an axis defined by the camera target and the `direction` (default is the positive *z* direction).
- `camera` — Rotate the camera about the point defined by the camera target.

`camorbit(dtheta,dphi,'coordsys','direction')` defines the axis of rotation for the data coordinate system using the `direction` argument in conjunction with the camera target. Specify `direction` as a three-element vector containing the *x*-, *y*-, and *z*-components of the direction or one of the characters, `x`, `y`, or `z`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`camorbit(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camorbit` operates on the current axes.

The behavior of `camorbit` differs from the `rotate3d` function in that while the `rotate3d` tool modifies the `View` property of the axes, the `camorbit` function fixes the

aspect ratio and modifies the `CameraTarget`, `CameraPosition`, and `CameraUpVector` properties of the axes. See Axes Properties for more information on all axes properties.

Examples

Rotate the camera horizontally about a line defined by the camera target point and a direction that is parallel to the *y*-axis. Visualize this rotation as a cone formed with the camera target at the apex and the camera position forming the base:

```
surf(peaks)
axis vis3d
for i=1:36
   camorbit(10,0,'data',[0 1 0])
   drawnow
end
```

Rotate in the `camera` coordinate system to orbit the camera around the axes along a circle while keeping the center of a circle at the camera target:

```
surf(peaks)
axis vis3d
for i=1:36
   camorbit(10,0,'camera')
   drawnow
end
```

Alternatives

Enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

More About

- "Camera Graphics Terminology"

See Also

`axes` | `axis` | `camdolly` | `campan` | `camzoom` | `camroll`

Introduced before R2006a

campan

Rotate camera target around camera position

Syntax

```
campan(dtheta,dphi)
campan(dtheta,dphi,coordsys)
campan(dtheta,dphi,coordsys,direction)
campan(axes_handle,...)
```

Description

`campan(dtheta,dphi)` rotates the camera target of the current axes around the camera position by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`campan(dtheta,dphi,coordsys)` determine the center of rotation using the `coordsys` argument. It can take on two values:

- `'data'` (default) — Rotate the camera target around an axis defined by the camera position and the `direction` (default is the positive *z* direction)
- `'camera'` — Rotate the camera about the point defined by the camera target.

`campan(dtheta,dphi,coordsys,direction)` defines the axis of rotation for the data coordinate system using the `direction` argument with the camera position. Specify `direction` as a three-element vector containing the *x*-, *y*-, and *z*-components of the direction or one of the characters, `'x'`, `'y'`, or `'z'`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`campan(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `campan` operates on the current axes.

Examples

Move the camera target to pan the object in a circular motion.

```
sphere;
axis vis3d
hPan = sin(-pi:1:pi);
vPan = cos(-pi:1:pi);
for k=1:length(hPan)
   campan(hPan(k),vPan(k))
   pause(.1)
end
```

More About

- "Camera Graphics Terminology"

See Also

axes | camdolly | camorbit | camtarget | camzoom | camroll

Introduced before R2006a

campos

Set or query camera position

Syntax

```
campos
campos([camera_position])
campos('mode')
campos('auto')
campos('manual')
campos(axes_handle,...)
```

Description

`campos` returns the camera position in the current axes.

`campos([camera_position])` sets the position of the camera in the current axes to the specified value. Specify the position as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`campos('mode')` returns the value of the camera position mode, which can be either `auto` (the default) or `manual`.

`campos('auto')` sets the camera position mode to `auto`.

`campos('manual')` sets the camera position mode to `manual`.

`campos(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `campos` operates on the current axes.

`campos` sets or queries values of the axes `CameraPosition` and `CameraPositionMode` properties. The camera position is the point in the Cartesian coordinate system of the axes from which you view the scene.

Examples

Move the camera along the *x*-axis in a series of steps:

```
surf(peaks)
axis vis3d off
for x = -200:5:200
    campos([x,5,10])
    drawnow
end
```

More About

- "Camera Graphics Terminology"

See Also

`axis` | `camproj` | `camup` | `camva` | `camtarget`

Introduced before R2006a

camproj

Set or query projection type

Syntax

```
camproj
camproj('projection_type')
camproj(axes_handle,...)
```

Description

`camproj` returns the projection type setting in the current axes. The projection type determines whether MATLAB 3-D views use a perspective or orthographic projection.

`camproj('projection_type')` sets the projection type in the current axes to the specified value. Possible values for *`projection_type`* are `orthographic` and `perspective`.

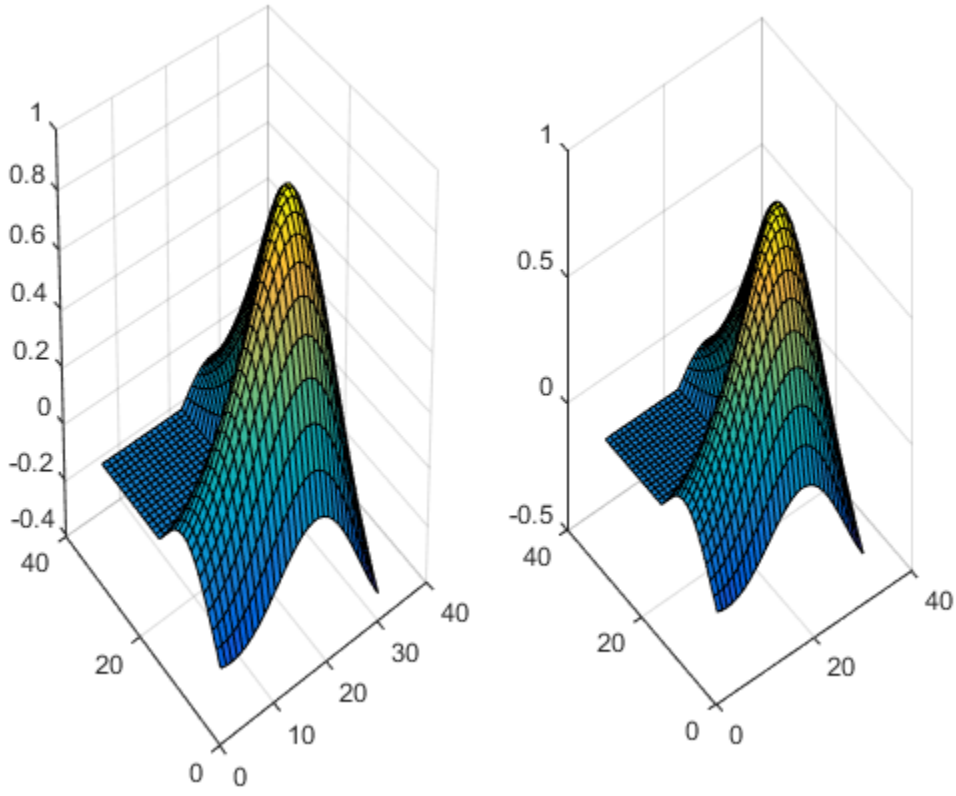
`camproj(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camproj` operates on the current axes.

`camproj` sets or queries values of the axes object `Projection` property.

Examples

Compare the different `camproj` settings using `subplot`:

```
subplot(1,2,1); surf(membrane); camproj('perspective')
subplot(1,2,2); surf(membrane); camproj('orthographic')
```



More About

- "Camera Graphics Terminology"

See Also

`axis` | `campos` | `camtarget` | `camup` | `camva`

Introduced before R2006a

camroll

Rotate camera about view axis

Syntax

```
camroll(dtheta)
camroll(axes_handle,dtheta)
```

Description

`camroll(dtheta)` rotates the camera around the camera viewing axis by the amounts specified in `dtheta` (in degrees). The viewing axis is the line passing through the camera position and the camera target.

`camroll(axes_handle,dtheta)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camroll` operates on the current axes.

`camroll` sets the axes `CameraUpVector` property and also sets the `CameraUpVectorMode` property to `manual`.

Examples

Rotate the camera around the viewing axis:

```
surf(peaks)
axis vis3d
for i=1:36
 camroll(10)
 drawnow
end
```

More About

- "Camera Graphics Terminology"

See Also

`axes` | `axis` | `camdolly` | `camorbit` | `camzoom` | `campan`

Introduced before R2006a

camtarget

Set or query location of camera target

Syntax

```
camtarget
camtarget([camera_target])
camtarget('mode')
camtarget('auto')
camtarget('manual')
camtarget(axes_handle,...)
```

Description

`camtarget` returns the location of the camera target in the current axes. The camera target is the location in the axes that the camera points to. The camera remains oriented toward this point regardless of its position.

`camtarget([camera_target])` sets the camera target in the current axes to the specified value. Specify the target as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`camtarget('mode')` returns the value of the camera target mode, which can be either `auto` (default) or `manual`.

`camtarget('auto')` sets the camera target mode to `auto`. When the camera target mode is `auto`, the camera target is the center of the axes plot box.

`camtarget('manual')` sets the camera target mode to `manual`.

`camtarget(axes_handle,...)` performs the set or query on the axes identified by `axes_handle`. When you do not specify an axes handle, `camtarget` operates on the current axes.

`camtarget` sets or queries values of the axes object `CameraTarget` and `CameraTargetMode` properties.

Examples

Move the camera position and the camera target along the *x*-axis in a series of steps:

```
surf(peaks);
axis vis3d
xp = linspace(-150,40,50);
xt = linspace(25,50,50);
for i=1:50
     campos([xp(i),25,5]);
     camtarget([xt(i),30,0])
     drawnow
end
```

More About

- "Camera Graphics Terminology"

See Also

`axis` | `campos` | `camup` | `camva`

Introduced before R2006a

camup

Set or query camera up vector

Syntax

```
camup
camup([up_vector])
camup('mode')
camup('auto')
camup('manual')
camup(axes_handle,...)
```

Description

`camup` returns the camera up vector setting in the current axes. The camera up vector specifies the direction that is oriented up in the scene.

`camup([up_vector])` sets the up vector in the current axes to the specified value. Specify the up vector as *x*, *y*, and *z* components.

`camup('mode')` returns the current value of the camera up vector mode, which can be either `auto` (default) or `manual`.

`camup('auto')` sets the camera up vector mode to `auto`. In `auto` mode, `[0 1 0]` is the up vector of for 2-D views. This means the *y*-axis points up. For 3-D views, the up vector is `[0 0 1]`, meaning the *z*-axis points up.

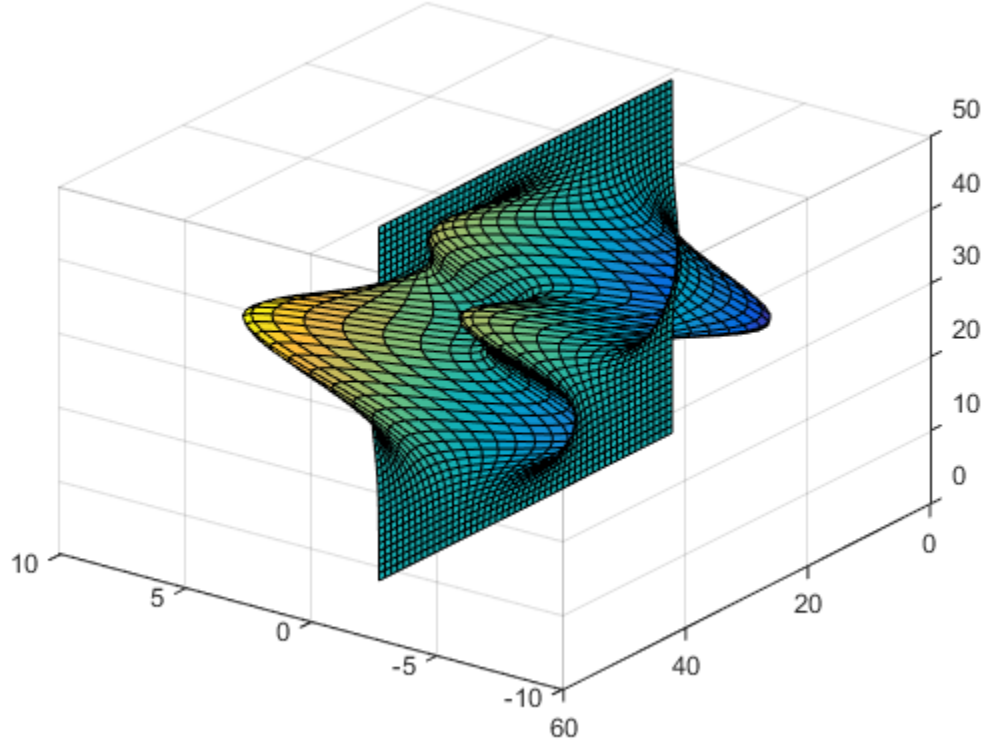
`camup('manual')` sets the camera up vector mode to `manual`. In `manual` mode, the value of the camera up vector does not change unless you set it.

`camup(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camup` operates on the current axes.

Examples

Set the *x*-axis to be the up axis:

```
surf(peaks)
camup([1 0 0]);
```



More About

- "Camera Graphics Terminology"

See Also

`axis` | `campos` | `camup` | `camtarget`

Introduced before R2006a

camva

Set or query camera view angle

Syntax

```
camva
camva(view_angle)
camva('mode')
camva('auto')
camva('manual')
camva(axes_handle,...)
```

Description

`camva` returns the camera view angle setting in the current axes. The camera view angle determines the field of view of the camera. Larger angles produce a smaller view of the scene. Implement zooming by changing the camera view angle.

`camva(view_angle)` sets the view angle in the current axes to the specified value. Specify the view angle in degrees.

`camva('mode')` returns the current value of the camera view angle mode, which can be either `auto` (the default) or `manual`.

`camva('auto')` sets the camera view angle mode to `auto`.

`camva('manual')` sets the camera view angle mode to `manual`.

`camva(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camva` operates on the current axes.

Tips

The `camva` function sets or queries values of the axes object `CameraViewAngle` and `CameraViewAngleMode` properties.

When the camera view angle mode is `auto`, the camera view angle adjusts so that the scene fills the available space in the window. If you move the camera to a different position, the camera view angle changes to maintain a view of the scene that fills the available area in the window.

Setting a camera view angle or setting the camera view angle to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the camera view angle to its current value,

```
camva(camva)
```

can cause a change in the way the graph looks. See `axes` for more information.

Examples

Create two pushbuttons, one that zooms in and another that zooms out:

```
% Set the range checking in the callback statements to keep
% the values for the camera view angle in the range greater
% than zero and less than 180.
uicontrol('Style','pushbutton',...
  'String','Zoom In',...
  'Position',[20 20 60 20],...
  'Callback','if camva <= 1;return;else;camva(camva-1);end');
uicontrol('Style','pushbutton',...
  'String','Zoom Out',...
  'Position',[100 20 60 20],...
  'Callback',...
  'if camva >= 179;return;else;camva(camva+1);end');
% Now create a graph to zoom in and out on:
surf(peaks);
```

More About

- “Camera Graphics Terminology”

See Also

`axis` | `campos` | `camup` | `camtarget`

Introduced before R2006a

camzoom

Zoom in and out on scene

Syntax

```
camzoom(zoom_factor)
camzoom(axes_handle,...)
```

Description

`camzoom(zoom_factor)` zooms in or out on the scene depending on the value specified by `zoom_factor`. If `zoom_factor` is greater than 1, the scene appears larger; if `zoom_factor` is greater than zero and less than 1, the scene appears smaller.

`camzoom(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camzoom` operates on the current axes.

More About

Tips

`camzoom` sets the axes `CameraViewAngle` property, which in turn causes the `CameraViewAngleMode` property to be set to `manual`. Note that setting the `CameraViewAngle` property disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This may result in a change to the aspect ratio of your graph. See the `axes` function for more information on this behavior.

- "Camera Graphics Terminology"

See Also

`axes` | `camdolly` | `campan` | `camorbit` | `camroll` | `camva`

Introduced before R2006a

cartToBary

Class: TriRep

(Will be removed) Convert point coordinates from cartesian to barycentric

---

Note: `cartToBary(TriRep)` will be removed in a future release. Use `cartesianToBarycentric(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
B = cartToBary(TR, SI, XC)
```

Description

`B = cartToBary(TR, SI, XC)` returns the barycentric coordinates of each point in `XC` with respect to its associated simplex `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`.
`XC`	Matrix that represents the Cartesian coordinates of the points to be converted. `XC` is of size `m`-by-`n`, where `m` is of `length(SI)`, the number of points to convert, and `n` is the dimension of the space where the triangulation resides.

Output Arguments

`B`	Matrix of dimension `m`-by-`k` where `k` is the number of vertices per simplex.

Definitions

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Compute the Delaunay triangulation of a set of points.

```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
dt = DelaunayTri(x,y)
```

Compute the barycentric coordinates of the incenters.

```
cc = incenters(dt);
tri = dt(:,:);
```

Plot the original triangulation and reference points.

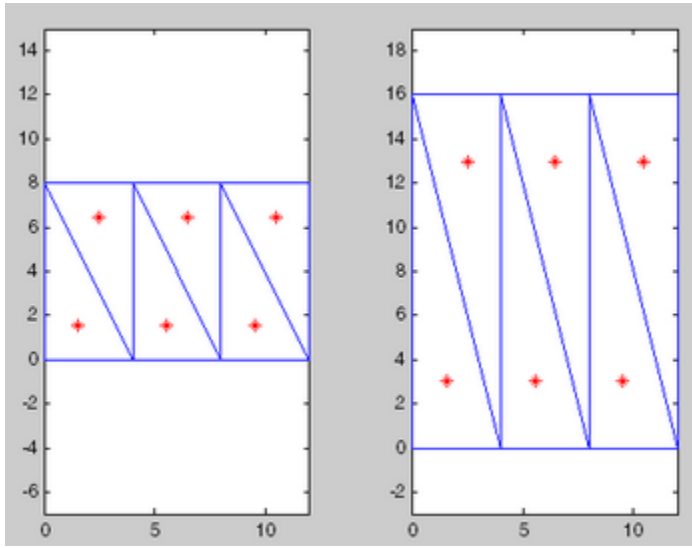
```
figure
subplot(1,2,1);
triplot(dt); hold on;
plot(cc(:,1), cc(:,2), '*r');
hold off;
axis equal;
```

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

```
b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(tr);
hold on;
plot(xc(:,1), xc(:,2), '*r');
hold off;
axis equal;
```



See Also

`delaunayTriangulation` | `barycentricToCartesian` | `pointLocation` | `triangulation`

cart2pol

Transform Cartesian coordinates to polar or cylindrical

Syntax

```
[THETA,RHO,Z] = cart2pol(X,Y,Z)
[THETA,RHO] = cart2pol(X,Y)
```

Description

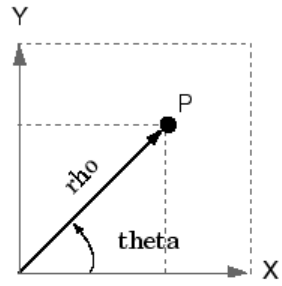
`[THETA,RHO,Z] = cart2pol(X,Y,Z)` transforms three-dimensional Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z`, into cylindrical coordinates. `THETA` is a counterclockwise angular displacement in radians from the positive *x*-axis, `RHO` is the distance from the origin to a point in the *x*-*y* plane, and `Z` is the height above the *x*-*y* plane. Arrays `X`, `Y`, and `Z` must be the same size (or any can be scalar).

`[THETA,RHO] = cart2pol(X,Y)` transforms two-dimensional Cartesian coordinates stored in corresponding elements of arrays `X` and `Y` into polar coordinates.

More About

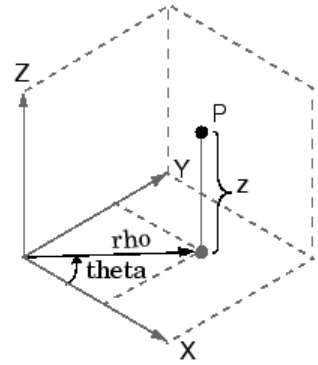
Algorithms

The mapping from two-dimensional Cartesian coordinates to polar coordinates, and from three-dimensional Cartesian coordinates to cylindrical coordinates is



Two-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
```



Three-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
```

See Also

`cart2sph` | `pol2cart` | `sph2cart`

Introduced before R2006a

cart2sph

Transform Cartesian coordinates to spherical

Syntax

```
[azimuth,elevation,r] = cart2sph(X,Y,Z)
```

Description

`[azimuth,elevation,r] = cart2sph(X,Y,Z)` transforms Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z` into spherical coordinates. `azimuth` and `elevation` are angular displacements in radians. `azimuth` is the counterclockwise angle in the *x*-*y* plane measured from the positive *x*-axis. `elevation` is the elevation angle from the *x*-*y* plane. `r` is the distance from the origin to a point.

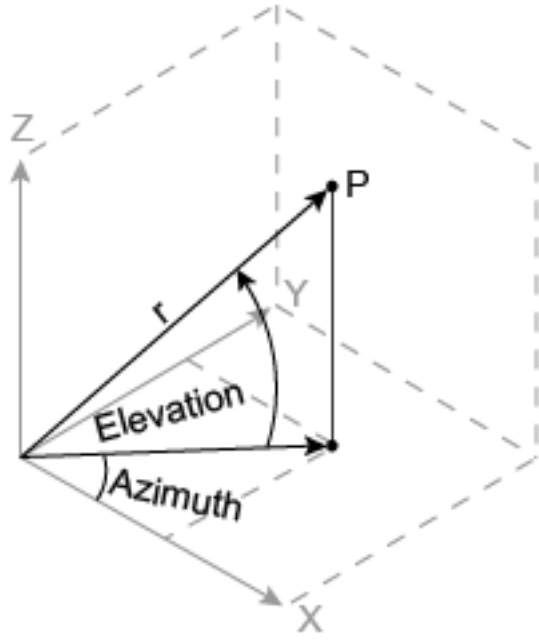
Arrays `X`, `Y`, and `Z` must be the same size (or any of them can be scalar).

More About

Algorithms

The mapping from three-dimensional Cartesian coordinates to spherical coordinates is

```
azimuth = atan2(y,x)
elevation = atan2(z,sqrt(x.^2 + y.^2))
r = sqrt(x.^2 + y.^2 + z.^2)
```



The notation for spherical coordinates is not standard. For the `cart2sph` function, `elevation` is measured from the *x*-*y* plane. Notice that if `elevation = 0`, the point is in the *x*-*y* plane. If `elevation = pi/2`, then the point is on the positive *z*-axis.

See Also

`cart2pol` | `pol2cart` | `sph2cart`

Introduced before R2006a

cast

Cast variable to different data type

Syntax

```
B = cast(A,newclass)
B = cast(A,'like',p)
```

Description

`B = cast(A,newclass)` converts `A` to class `newclass`, where `newclass` is the name of a built-in data type compatible with `A`. The `cast` function truncates any values in `A` that are too large to map into `newclass`.

`B = cast(A,'like',p)` converts `A` to the same data type and sparsity as the variable `p`. If `A` and `p` are both real, then `B` is also real. Otherwise, `B` is complex.

Examples

Convert Numeric Data Type

Convert an `int8` value to `uint8`.

Define a scalar 8-bit integer.

```
a = int8(5);
```

Convert `a` to an unsigned 8-bit integer.

```
b = cast(a,'uint8');
class(b)
```

```
ans =
uint8
```

Match Data Type and Complex Nature of p

Define a single precision vector `p` that is complex valued.

```
p = single([1+i 2]);
```

Define a 2-by-3 matrix of ones.

```
A = ones(2,3);
```

Convert A to the same data type and complexity (real or complex) as p.

```
B = cast(A,'like',p)
```

```
B =

   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
```

Check the class of B.

```
class(B)
```

```
ans =

single
```

More About

- "Class Support for Array-Creation Functions"

See Also

`class` | `typecast`

Introduced before R2006a

cat

Concatenate arrays along specified dimension

Syntax

```
C = cat(dim, A, B)
C = cat(dim, A1, A2, A3, A4, ...)
```

Description

`C = cat(dim, A, B)` concatenates the arrays `A` and `B` along array the dimension specified by `dim`. The `dim` argument must be a real, positive, integer value.

`C = cat(dim, A1, A2, A3, A4, ...)` concatenates all the input arrays (`A1`, `A2`, `A3`, `A4`, and so on) along array dimension `dim`.

For nonempty arrays, `cat(2, A, B)` is the same as `[A, B]`, and `cat(1, A, B)` is the same as `[A; B]`.

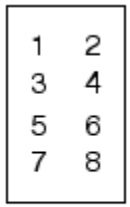
If your input arrays are tables, `dim` must be either 1 or 2. `cat` then concatenates by calling `horzcat` or `vertcat` respectively.

Examples

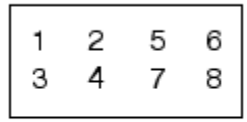
Given

```
A =                 B =
     1     2                 5     6
     3     4                 7     8
```

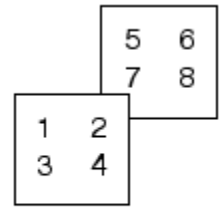
concatenating along different dimensions produces



C = cat(1,A,B)



C = cat(2,A,B)



C = cat(3,A,B)

The commands

```
A = magic(3); B = pascal(3);
C = cat(4, A, B);
```

produce a 3-by-3-by-1-by-2 array.

More About

Tips

When used with comma-separated list syntax, `cat(dim, C{:})` or `cat(dim, C.field)` is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.

You can concatenate categorical arrays with cell arrays of strings. For more information, see “Combine Categorical Arrays”.

If all the input arrays are ordinal categorical arrays, they must have the same sets of categories including their order. For more information, see “Ordinal Categorical Arrays”.

You can concatenate datetime arrays with cell arrays of strings.

You can concatenate duration arrays and calendar duration arrays. The result is a calendar duration array.

You can concatenate duration or calendar duration arrays with numeric arrays. Prior to concatenation, MATLAB converts the numeric array to an array of equivalent days using the `days` function.

For information on combining unlike integer types, integers with nonintegers, cell arrays with non-cell arrays, or empty matrices with other elements, see “Valid Combinations of Unlike Classes”

See Also

`vertcat` | `strcat` | `char` | `special character` | `horzcat` | `num2cell` | `reshape` | `squeeze` | `strjoin` | `shiftdim`

Introduced before R2006a

categorical

Create categorical array

Use `categorical` to create a categorical array from data with values from a finite set of discrete categories. To group numeric data into categories, use `discretize`.

Syntax

```
B = categorical(A)
B = categorical(A,valueset)
B = categorical(A,valueset,catnames)
B = categorical(A, ___ ,Name,Value)
```

Description

`B = categorical(A)` creates a categorical array from the array, `A`. The categories of `B` are the sorted unique values from `A`.

For more information on creating and using categorical arrays, see “Categorical Arrays”.

`B = categorical(A,valueset)` creates one category for each value in valueset. The categories of `B` are in the same order as the values of valueset.

You can use valueset to include categories for values not present in `A`. Conversely, if `A` contains any values not present in `valueset`, the corresponding elements of `B` are undefined.

`B = categorical(A,valueset,catnames)` names the categories in `B` by matching the category values in valueset with the names in catnames.

`B = categorical(A, ___ ,Name,Value)` creates a categorical array with additional options specified by one or more Name,Value pair arguments. You can include any of the input arguments in previous syntaxes.

For example, you can specify that the categories have a mathematical ordering.

Examples

Create Categorical Array from Strings

Convert a cell array of strings to a categorical array.

Create a cell array of strings.

```
A = {'r' 'b' 'g'; 'g' 'r' 'b'; 'b' 'r' 'g'}
```

```
A =

    'r'    'b'    'g'
    'g'    'r'    'b'
    'b'    'r'    'g'
```

A is a 3-by-3 cell array containing three unique values.

Convert the cell array of strings, A, to a categorical array, B.

```
B = categorical(A)
```

```
B =

     r      b      g
     g      r      b
     b      r      g
```

The contents of B match the contents of A.

Display the categories of B.

```
categories(B)
```

```
ans =

    'b'
    'g'
    'r'
```

The categories of B are the unique values from A in alphabetical order.

Create Categorical Array and Specify Possible Unique Values

Convert a cell array of strings, A, to a categorical array. Specify a list of categories that includes values that are not present in A.

Create a cell array of strings.

```
A = {'republican' 'democrat'; 'democrat' 'democrat'; 'democrat' 'republican'}
```

```
A =

    'republican'    'democrat'
    'democrat'      'democrat'
    'democrat'      'republican'
```

A is a 3-by-2 cell array containing two unique values.

Convert the cell array of strings, A, to a categorical array, B and include a category for `independent`.

```
valueset = {'democrat' 'republican' 'independent'}
B = categorical(A,valueset)
```

```
B =

     republican      democrat
     democrat        democrat
     democrat        republican
```

The contents of B match the contents of A.

Display the categories of B.

```
categories(B)
```

```
ans =

    'democrat'
    'republican'
    'independent'
```

The categories of B are in the same order as the values specified in `valueset`.

Create Categorical Array and Specify Category Names

Create a cell array of strings.

```
A = {'r' 'b' 'g'; 'g' 'r' 'b'; 'b' 'r' 'g'}
```

```
A =
```

```
    'r'    'b'    'g'
    'g'    'r'    'b'
    'b'    'r'    'g'
```

A is a 3-by-3 cell array containing three unique values.

Convert the cell array of strings, A, to a categorical array, B, and specify category names.

```
B = categorical(A,{'r' 'g' 'b'},{'red' 'green' 'blue'})
```

```
B =

     red        blue       green
     green      red        blue
     blue       red        green
```

B uses the specified category names for the contents from A.

Display the categories of B.

```
categories(B)
```

```
ans =

    'red'
    'green'
    'blue'
```

The categories of B are in the order they were specified.

Create Categorical Array from Integers

Create a 2-by-3 numeric array.

```
A = gallery('integerdata',3,[2,3],3)
```

```
A =

     2     1     2
     1     1     3
```

A contains the values 1, 2, and 3.

Convert the numeric array, A, to a categorical array. Use the values 1, 2, and 3 to define the categories `car`, `bus`, and `bike`, respectively.

```
valueset = 1:3;
catnames = {'car' 'bus' 'bike'};

B = categorical(A,valueset,catnames)

B =

     bus      car      bus
     car      car      bike
```

`categorical` maps the numeric values in `valueset` to the category names in `catnames`.

The 2-by-3 categorical array, `B`, is not ordinal. Therefore, you can only compare the values in `B` for equality. To compare the values in `B` using relational operators, such as less than and greater than, you must include the `'Ordinal',true` name-value pair argument.

Create Ordinal Categorical Array from Integers

Create a 5-by-2 numeric array.

```
A = gallery('integerdata',3,[5,2],1)

A =

     3     2
     3     3
     3     2
     2     1
     3     2
```

`A` contains the values `1`, `2`, and `3`.

Convert the numeric array, `A`, to an ordinal categorical array where 1, 2, and 3 represent child, adult, and senior respectively.

```
valueset = [1:3];
catnames = {'child' 'adult' 'senior'};

B = categorical(A,valueset,catnames,'Ordinal',true)

B =
```

```
     senior      adult
     senior      senior
     senior      adult
     adult       child
     senior      adult
```

Since B is ordinal, the categories of B have a mathematical ordering, `child < adult < senior`.

Create Categorical Array by Binning Numeric Data

Use the `discretize` function (instead of `categorical`) to bin 100 random numbers into three categories.

```
x = rand(100,1);
y = discretize(x,[0 .25 .75 1],'categorical',{'small','medium','large'});
summary(y)
```

```
     small       22
     medium      46
     large       32
```

Input Arguments

A — Input array

numeric array | logical array | categorical array | cell array of strings | ...

Input array, specified as a numeric array, logical array, categorical array, or cell array of strings.

If A contains missing values, the corresponding element of B is `<undefined>`. Missing values are `NaN` for numeric arrays, the empty string (`''`) for cell arrays of strings, and `<undefined>` for categorical arrays. B does not have a category for undefined values. To create an explicit category for missing or undefined values, you must include the desired category name in `catnames`, and `NaN`, the empty string, or `<undefined>` in `valueset`.

In addition to an array, A can be an object with the following class methods:

- `unique`
- `eq`

`valueset` — Values to define categories

`unique(A)` (default) | vector of unique values

Values to define categories, specified as a vector of unique values. The data type of `valueset` and the data type of `A` must be the same.

`catnames` — Category names

cell array of strings

Category names, specified as a cell array of strings. If you do not specify the `catnames` input argument, `categorical` uses the values in `valueset` as category names.

To merge multiple distinct values in `A` into a single category in `B`, include duplicate names corresponding to those values.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Ordinal',true` specifies that the categories have a mathematical ordering

`'Ordinal'` — Mathematical ordering indicator

`false` (default) | `true` | `0` | `1`

Mathematical ordering indicator, specified as the comma-separated pair consisting of `'Ordinal'` and either `false`, `true`, `0`, or `1`.

`false`	`categorical` creates a categorical array that is not ordinal. This is the default behavior. The categories of `B` have no mathematical ordering. Therefore, you can only compare the values in `B` for equality.
`true`	`categorical` creates an ordinal categorical array. The categories of `B` have a mathematical ordering, such that the first category specified is the smallest and the last category is the largest. You can compare the values in `B` using relational operators, such as less than and greater than, in addition to comparing the values for equality.

'Protected' — Category protection indicator

false | true | 0 | 1

Category protection indicator specified as the comma-separated pair consisting of 'Protected' and either false, true, 0, or 1. The categories of ordinal categorical arrays are always protected. The default value is true when you specify 'Ordinal',true and false otherwise.

false	When you assign new values to B, the categories update automatically. Therefore, you can combine (nonordinal) categorical arrays that have different categories. The categories can update accordingly to include the categories from both arrays.
true	When you assign new values to B, the values must belong to one of the existing categories. Therefore, you can only combine arrays that have the same categories. To add new categories to B, you must use the function addcats.

More About

- "Advantages of Using Categorical Arrays"

See Also

categories | discretize

categories

Categories of categorical array

Syntax

```
C = categories(A)
```

Description

`C = categories(A)` returns a cell array of strings containing the categories of the categorical array, `A`.

Examples

List Categories in Categorical Array

Create a categorical array, `A`.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'})
```

```
A =

     plane      car      train      car      plane
```

`A` is a 1-by-5 categorical array.

Display the categories of `A`.

```
C = categories(A)
```

```
C =

    'car'
    'plane'
    'train'
```

Since you created A by specifying only an input array, the categories appear in alphabetical order.

List Categories in Ordinal Categorical Array

Create an ordinal categorical array.

```
A = categorical({'medium' 'large'; 'small' 'xlarge'; 'large' 'medium'},...
    {'small' 'medium' 'large' 'xlarge'},'Ordinal',true)
```

```
A =

     medium      large
     small       xlarge
     large       medium
```

A is a 3-by-2 ordinal categorical array.

Display the categories of A.

```
C = categories(A)
```

```
C =

    'small'
    'medium'
    'large'
    'xlarge'
```

The categories appear in the order in which you specified them. Since A is ordinal, the categories have the mathematical ordering `small < medium < large < xlarge`.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

More About

Tips

- `C` includes all categories in `A`, even if `A` does not contain any data from a category. To see the unique values in `A`, use `unique(A)`.
- The order of the categories listed in `C` is the same order used by functions, such as `summary` and `hist`. To change the order of the categories, use `reordercats`.

See Also

`addcats` | `categorical` | `hist` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `reordercats` | `setcats` | `unique`

caxis

Color axis scaling

Syntax

```
caxis([cmin cmax])
caxis auto
caxis manual
caxis(caxis)
v = caxis
caxis(axes_handle,...)
```

Description

`caxis` controls the mapping of data values to the colormap. It affects any `surface`, `patch`, or `image` with indexed `CData` and `CDataMapping` set to scaled. It does not affect surfaces, patches, or images with true color `CData` or with `CDataMapping` set to `direct`.

`caxis([cmin cmax])` sets the color limits to specified minimum and maximum values. Data values less than `cmin` or greater than `cmax` map to `cmin` and `cmax`, respectively. Values between `cmin` and `cmax` linearly map to the current colormap.

`caxis auto` computes the color limits automatically using the minimum and maximum data values. This is the default behavior. Color values set to `Inf` map to the maximum color, and values set to `-Inf` map to the minimum color. Faces or edges with color values set to `NaN` are not drawn.

`caxis manual` and `caxis(caxis)`freeze the color axis scaling at the current limits. This enables subsequent plots to use the same limits when `hold` is `on`.

`v = caxis` returns a two-element row vector containing the `[cmin cmax]` currently in use.

`caxis(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

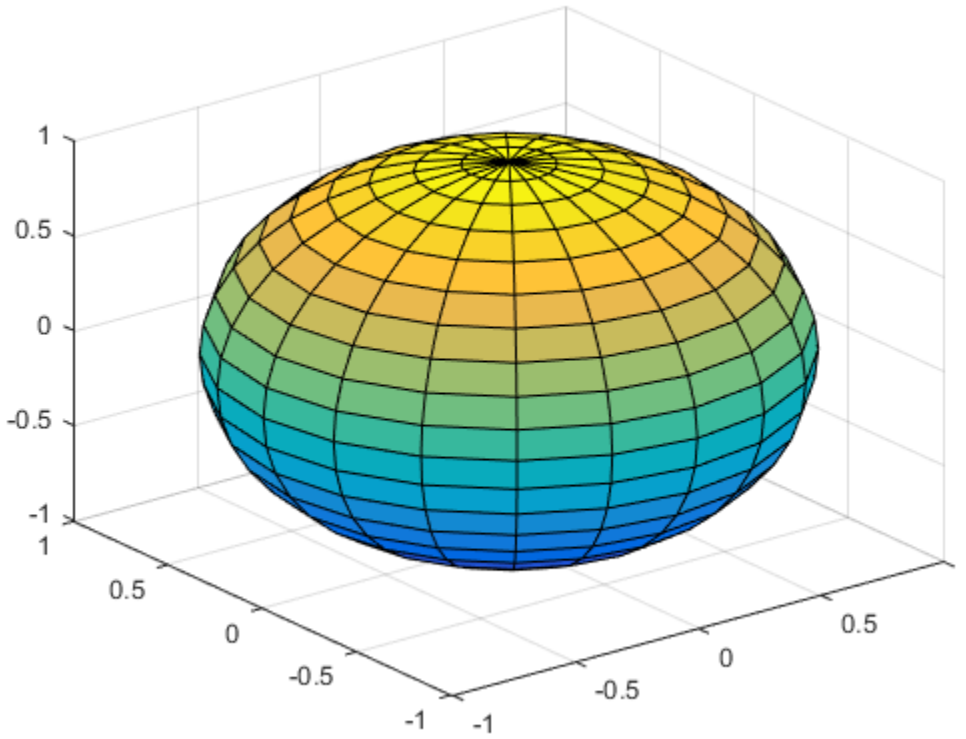
`caxis` changes the `CLim` and `CLimMode` properties of axes graphics objects.

Examples

Change Color Axis Scaling

Define `X`, `Y`, and `Z` as data for a sphere and view the data as a surface. Define the colors for the surface, `C`.

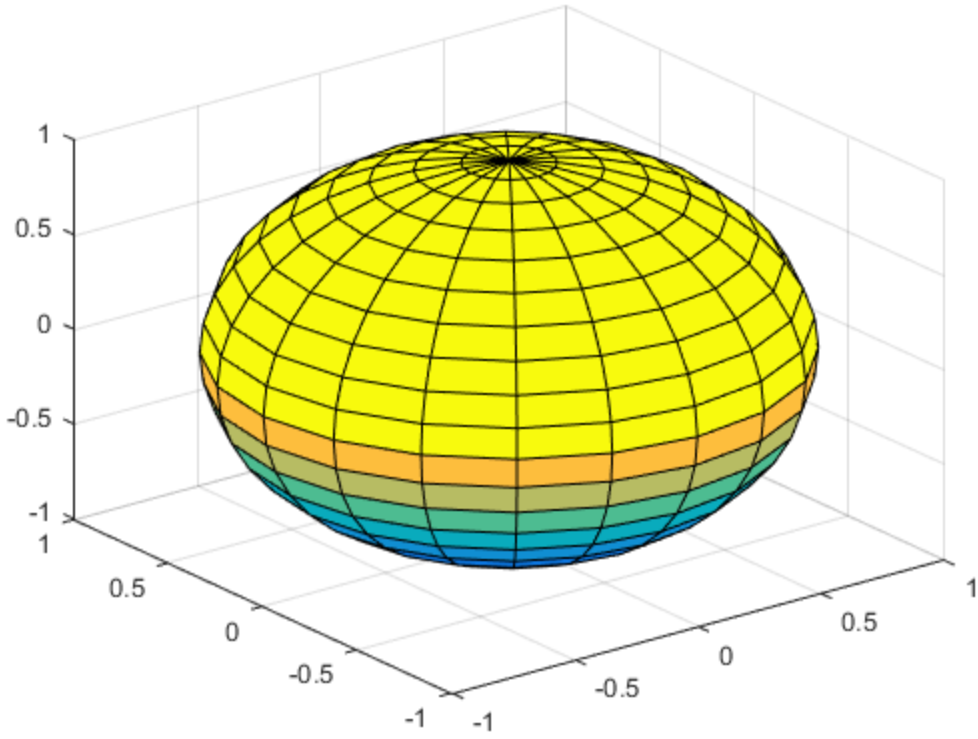
```
[X,Y,Z] = sphere;
C = Z;
surf(X,Y,Z,C)
```



Values of `C` are in the range `[-1,1]`. Values of `C` near -1 are assigned the lowest values in the colormap. Values of `C` near 1 are assigned the highest values in the colormap.

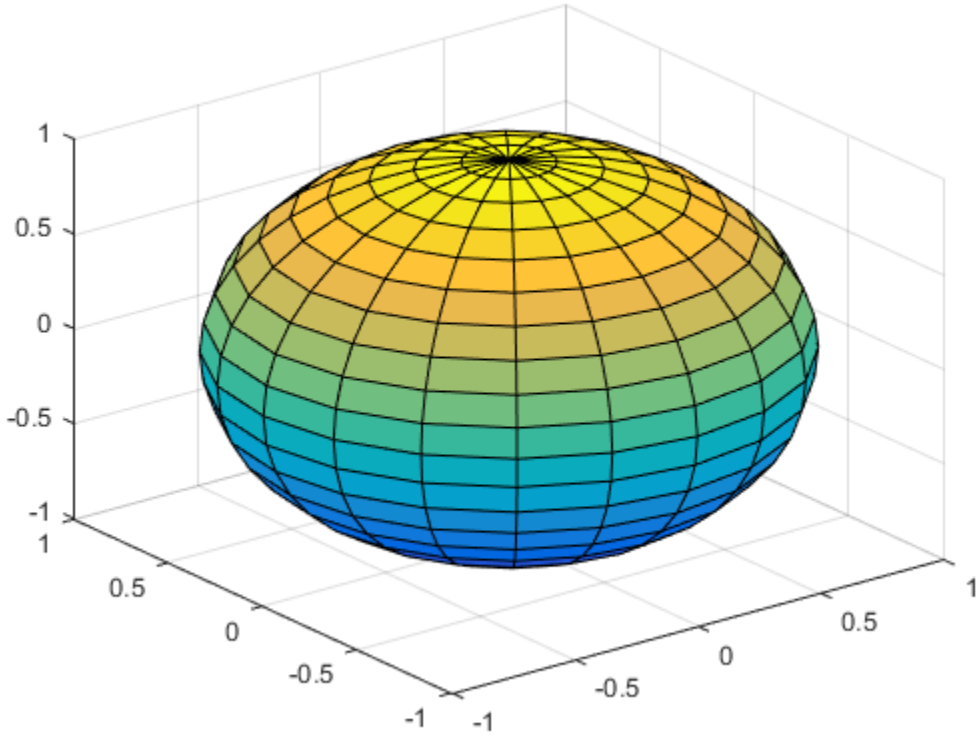
Map the top half of the surface to the highest value in the color table by setting the maximum color limit to 0.

```
caxis([-1,0])
```



Reset the axis scaling back to its default range.

```
caxis auto
```



Return the values of the current color limits.

```
v = caxis
```

```
v =

    -1     1
```

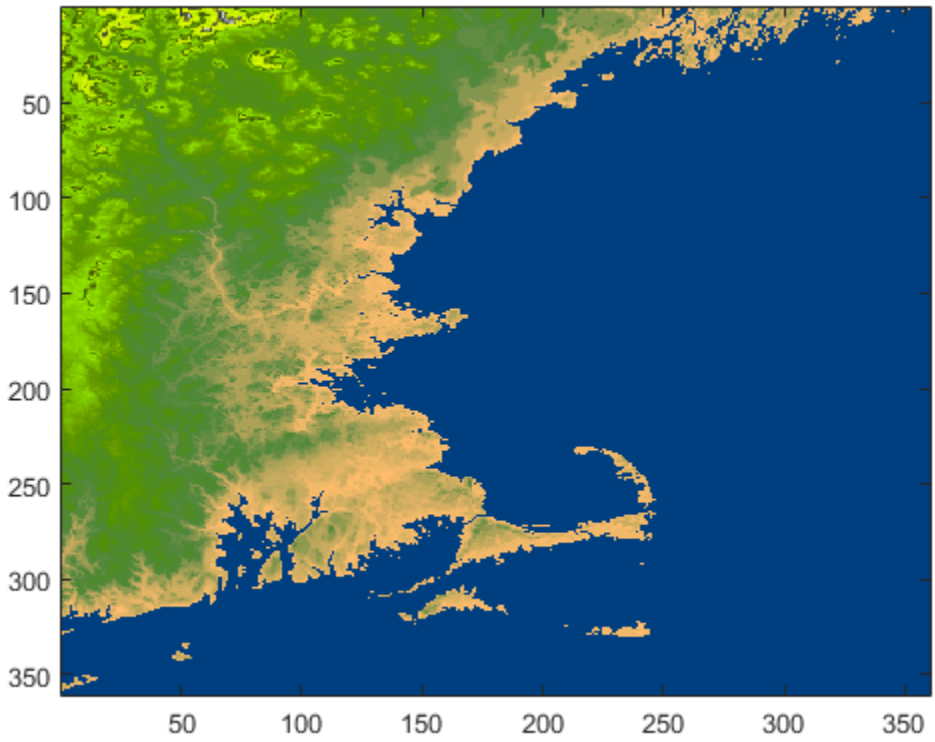
Set Color Axis Scaling for Image Data

Load the `cape` file which contains the image data, `X`, and the colormap, `map`, for Cape Cod, Massachusetts.

```
load cape
```

Display the image with `CDataMapping` set to `scaled`, and use the `map` colormap.

```
figure
image(X,'CDataMapping','scaled')
colormap(map)
```

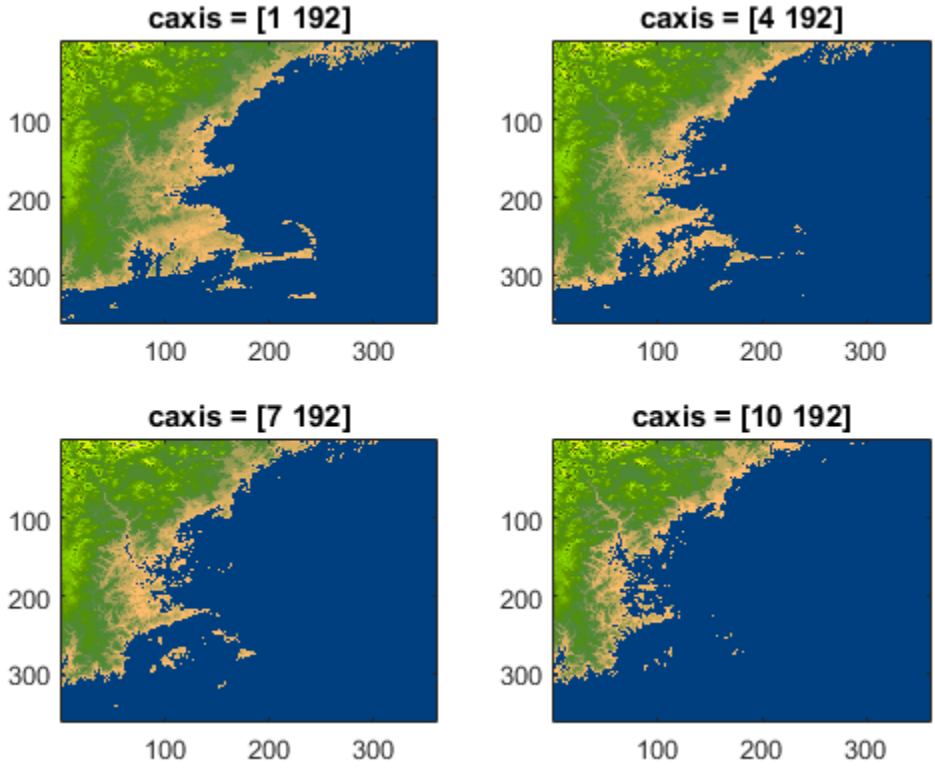


The color limits span the range of the image data, which is 1 to 192. The blue color of the ocean is the first color in the colormap and is mapped to the lowest data value, 1. You can effectively move sea level by changing the lower color limit value using `caxis`.

Display the image data using four different color limit values.

```
load cape
```

```
figure
colormap(map)

subplot(2,2,1)
image(X,'CDataMapping','scaled')
title('caxis = [1 192]')

subplot(2,2,2)
image(X,'CDataMapping','scaled')
caxis([4,192]) % change caxis
title('caxis = [4 192]')

subplot(2,2,3)
image(X,'CDataMapping','scaled')
caxis([7,192]) % change caxis
title('caxis = [7 192]')

subplot(2,2,4)
image(X,'CDataMapping','scaled')
caxis([10,192]) % change caxis
title('caxis = [10 192]')
```



More About

Tips

How Color Axis Scaling Works

Surface, patch, and image graphics objects having indexed `CData` and `CDataMapping` set to `scaled` map `CData` values to colors in the figure colormap each time they render. `CData` values equal to or less than `cmin` map to the first color value in the colormap, and `CData` values equal to or greater than `cmax` map to the last color value in the colormap. The following linear transformation is performed on the intermediate values (referred to

as `C` below) to map them to an entry in the colormap (whose length is `m`, and whose row index is referred to as `index` below).

```
index = fix((C-cmin)/(cmax-cmin)*m)+1;
%Clamp values outside the range [1 m]
index(index<1) = 1;
index(index>m) = m;
```

See Also

`axes` | `axis` | `colormap` | `get` | `mesh` | `pcolor` | `set` | `surf`

Introduced before R2006a

cd

Change current folder

Syntax

```
cd(newFolder)
oldFolder = cd(newFolder)
cd
```

Description

`cd(newFolder)` changes the current folder to newFolder.

`oldFolder = cd(newFolder)` returns the existing current folder as a string to oldFolder, and then changes the current folder to newFolder.

`cd` displays the current folder.

Input Arguments

`newFolder`

A string specifying the folder to which you want to change the current folder. Valid values can be any one of the following:

- A full or relative path.
- `../`, which indicates one level up from the current folder.
- Multiple strings of `../`, which indicates multiple levels up from the current folder.
- `./`, which indicates a path relative to the current folder, although without the `./`, `cd` assumes that the path is relative to the current folder.

Output Arguments

`oldFolder`

A string specifying the current folder that was in place when you issued the `cd` command.

Definitions

The current folder is a reference location that MATLAB uses to find files. This folder is sometimes referred to as the *current directory*, *current working folder*, or *present working directory*.

Examples

Use `cd` with the `matlabroot` function to change the current folder to the examples directory for the currently running version of MATLAB:

```
cd(fullfile(matlabroot, '/help/techdoc/matlab_env/examples'))
```

On a Microsoft Windows platform, specify the full path to change the current folder from any location to the examples directory for MATLAB Version 7.11 (R2010b), assuming that version is installed on your `C:` drive:

```
cd('C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/examples')

% Change the current folder from
% C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/examples to
% C:/Program Files/MATLAB/R2010b/help/techdoc:

cd ../..

% Use a relative path to change the current folder from
% C:/Program Files/MATLAB/R2010b/help/techdoc back to
% C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/examples:

cd matlab_env/examples

% Change the current folder from its current location to a new
% location,but save its previous location. Later, change the
```

```
% current folder to the previous location.

% This returns
% C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/examples
% to oldFolder, and then changes the current folder to
% C:/Program Files:

oldFolder = cd('C:/Program Files')

% Display current folder:

pwd

% Change the current folder to the previous location:
cd(oldFolder)

pwd
```

On a UNIX platform, change the current folder to the examples directory for the currently running version of MATLAB, assuming it is installed in your home location:

```
cd('~/help/techdoc/matlab_env/examples')
```

More About

Tips

- When using the command syntax (`cd newfolder`), if the `newFolder` string contains spaces, enclose the string in single quotation marks.
- On UNIX platforms, use the ~ (tilde) character to represent the user home directory.
- If you use `cd` within a local function, the folder change persists after program control returns from the function. That is, the scope of the folder change is global.

- "Specify File Names"
- "Files and Folders that MATLAB Accesses"

See Also

`dir` | `fileparts` | `path` | `pwd` | `what`

Introduced before R2006a

convexHull

Class: DelaunayTri

(Will be removed) Convex hull

Note: `convexHull(DelaunayTri)` will be removed in a future release. Use `convexHull(delaunayTriangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
K = convexHull(DT)
[K AV] = convexHull(DT)
```

Description

`K = convexHull(DT)` returns the indices into the array of points `DT.X` that correspond to the vertices of the convex hull.

`[K AV] = convexHull(DT)` returns the convex hull and the area or volume bounded by the convex hull.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

`K`	If the points lie in 2-D space, `K` is a column vector of length `numf`. Otherwise `K` is a matrix of size `numf`-by-`ndim`, `numf` being the number of

	facets in the convex hull, and `ndim` the dimension of the space where the points reside.
`AV`	The area or volume of the convex hull.

Definitions

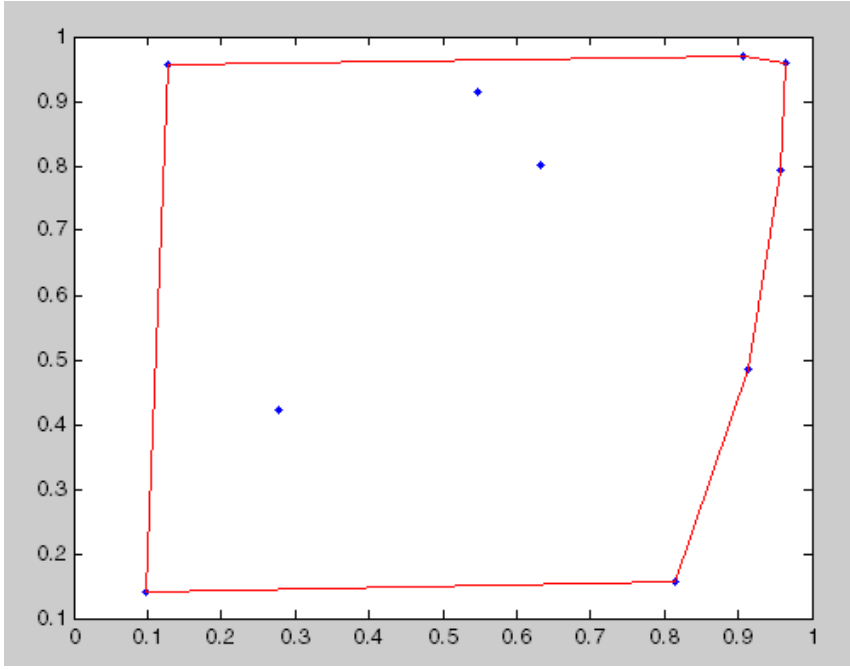
The convex hull of a set of points `X` is the smallest convex region containing all of the points of `X`.

Examples

Example 1

Compute the convex hull of a set of random points located within a unit square in 2-D space.

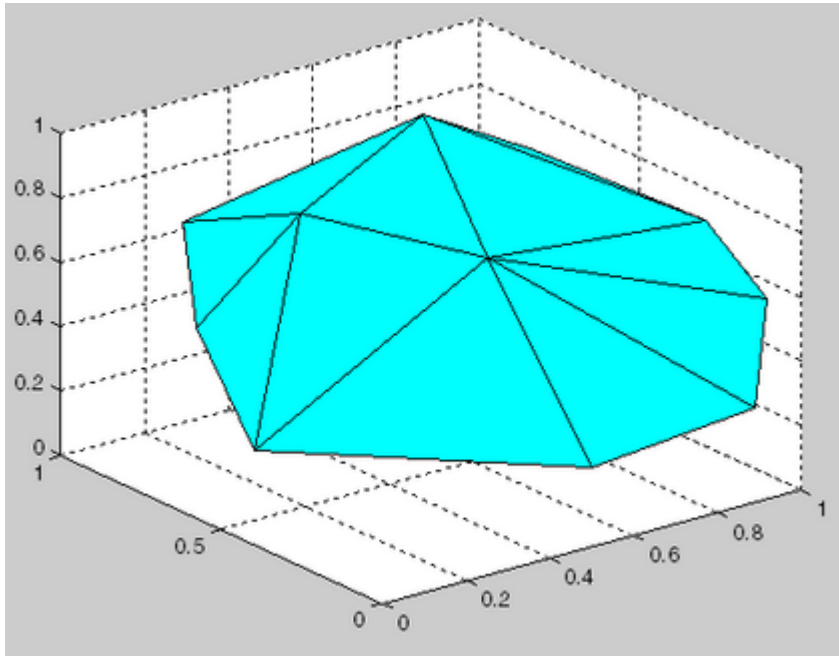
```
x = rand(10,1)
y = rand(10,1)
dt = DelaunayTri(x,y)
k = convexHull(dt)
plot(dt.X(:,1),dt.X(:,2), '.', 'markersize',10); hold on;
plot(dt.X(k,1),dt.X(k,2), 'r'); hold off;
```



Example 2

Compute the convex hull of a set of random points located within a unit cube in 3-D space, and the volume bounded by the convex hull.

```
X = rand(25,3)
dt = DelaunayTri(X)
[ch v] = convexHull(dt)
trisurf(ch, dt.X(:,1),dt.X(:,2),dt.X(:,3), 'FaceColor', 'cyan')
```



See Also

`delaunayTriangulation` | `voronoiDiagram` | `triangulation` | `convhull` | `convhulln`

cd

Class: FTP

Change or view current folder on FTP server

Syntax

```
cd(ftpobj,folder)
cd(ftpobj)
```

Description

`cd(ftpobj,folder)` changes the current folder on the FTP server.

`cd(ftpobj)` displays the current folder on the server.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`folder`

String enclosed in single quotation marks that specifies the target folder. To specify the folder above the current one, use `'..'`.

Examples

Connect to the MathWorks FTP server, change to the `pub` folder, and view its contents:

```
mw=ftp('ftp.mathworks.com');
cd(mw,'pub');
dir(mw)
```

See Also

`dir` | `ftp`

Introduced before R2006a

cdf2rdf

Convert complex diagonal form to real block diagonal form

Syntax

```
[V,D] = cdf2rdf(V,D)
```

Description

If the eigensystem `[V,D] = eig(X)` has complex eigenvalues appearing in complex-conjugate pairs, `cdf2rdf` transforms the system so `D` is in real diagonal form, with 2-by-2 real blocks along the diagonal replacing the complex pairs originally there. The eigenvectors are transformed so that

```
X = V*D/V
```

continues to hold. The individual columns of `V` are no longer eigenvectors, but each pair of vectors associated with a 2-by-2 block in `D` spans the corresponding invariant vectors.

Examples

The matrix

```
X =
    1     2     3
    0     4     5
    0    -5     4
```

has a pair of complex eigenvalues.

```
[V,D] = eig(X)

V =

    1.0000      -0.0191 - 0.4002i     -0.0191 + 0.4002i
         0            0 - 0.6479i           0 + 0.6479i
         0       0.6479                 0.6479
```

```
D =

    1.0000            0                       0
         0       4.0000 + 5.0000i             0
         0            0                 4.0000 - 5.0000i
```

Converting this to real block diagonal form produces

```
[V,D] = cdf2rdf(V,D)

V =

    1.0000    -0.0191     -0.4002
         0          0     -0.6479
         0     0.6479           0

D =

    1.0000          0          0
         0     4.0000     5.0000
         0    -5.0000     4.0000
```

More About

Algorithms

The real diagonal form for the eigenvalues is obtained from the complex form using a specially constructed similarity transformation.

See Also

`eig` | `rsf2csf`

Introduced before R2006a

cdfepoch

Convert date string or serial date number to CDF formatted dates

Syntax

```
E = cdfepoch(date)
```

Description

`E = cdfepoch(date)` converts the date, specified by `date`, into a `cdfepoch` object. `date` must be a valid date string, returned by `datestr`, or a serial date number, returned by `datenum`. `date` can also be a `cdfepoch` object.

When writing data to a CDF file using `cdfwrite`, use `cdfepoch` to convert MATLAB date strings or serial date numbers to CDF formatted dates. The MATLAB `cdfepoch` object simulates the `CDFEPOCH` data type in CDF files.

To convert a `cdfepoch` object into a MATLAB serial date number, use the `todatenum` function.

Definitions

The MATLAB serial date number calculates dates differently than CDF epochs.

A MATLAB serial date number represents the whole and fractional number of days from 0-Jan-0000 to a specific date. The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time.

A CDF epoch is the number of milliseconds since 1-Jan-0000.

Examples

Convert the current time in serial date number format into a CDF epoch object.

```
% NOW function returns current time as serial date number
```

```
dateobj = cdfepoch(now)

dateobj =

     cdfepoch object:
     11-Mar-2009 15:09:25
```

Convert the current time in date string format into a CDF epoch object.

```
% DATESTR function returns date as string
dateobj2 = cdfepoch(datestr(now))

dateobj2 =

     cdfepoch object:
     11-Mar-2009 15:09:25
```

Convert the CDF epoch object into a serial date number.

```
dateobj = cdfepoch(now);
mydatenum = todatenum(dateobj)

mydatenum =

     7.3384e+005
```

See Also

`datenum` | `datestr` | `todatenum` | `cdfinfo` | `cdfread` | `datetime`

Introduced before R2006a

cdfinfo

Information about Common Data Format (CDF) file

Syntax

```
info = cdfinfo(filename)
```

Description

`info = cdfinfo(filename)` returns information about the Common Data Format (CDF) file specified in the string `filename`.

Note Because `cdfinfo` creates temporary files, the current working directory must be writeable.

The following table lists the fields returned in the structure, `info`. The table lists the fields in the order that they appear in the structure.

Field	Description
`Filename`	Text string specifying the name of the file
`FileModDate`	Text string indicating the date the file was last modified
`FileSize`	Double scalar specifying the size of the file, in bytes
`Format`	Text string specifying the file format
`FormatVersion`	Text string specifying the version of the CDF library used to create the file
`FileSettings`	Structure array containing library settings used to create the file
`Subfiles`	Filenames containing the CDF file's data, if it is a multi-file format CDF
`Variables`	N-by-6 cell array, where N is the number of variables, containing information about the variables in the file. The columns present the following information:

Field	Description	
	Column 1	Text string specifying name of variable
	Column 2	Double array specifying the dimensions of the variable, as returned by the `size` function
	Column 3	Double scalar specifying the number of records assigned for the variable
	Column 4	Text string specifying the data type of the variable, as stored in the CDF file

	Column 5	Text string specifying the record and dimension variance settings for the variable. The single `T` or `F` to the left of the slash designates whether values vary by record. The zero or more `T` or `F` letters to the right of the slash designate whether values vary at each dimension. Here are some examples.
		`T/  (scalar variable`
		`F/T (one-dimensional variable)`
		T/TFF (three-dimensional variable)

`GlobalAttributes`	Structure array that contains one field for each global attribute. The name of each field corresponds to the name of an attribute. The data in each field, contained in a cell array, represents the entry values for that attribute.	
`VariableAttributes`	Structure array that contains one field for each variable attribute. The name of each field corresponds to the name of an attribute. The data in each field is contained in a *n*-by-2 cell array, where *n* is the number of variables. The first column of this cell array contains the variable names associated with the entries. The second column contains the entry values.	

Note Attribute names returned by `cdfinfo` might not match the names of the attributes in the CDF file exactly. Attribute names can contain characters that are illegal in MATLAB field names. `cdfinfo` removes illegal characters that appear at the beginning of attributes and replaces other illegal characters with underscores ('_'). When `cdfinfo` modifies an attribute name, it appends the attribute's internal number to the end

of the field name. For example, the attribute name `Variable%Attribute` becomes `Variable_Attribute_013`.

Note: To improve performance, turn off the file validation which the CDF library does by default when opening files. For more information, see `cdflib.setValidate`.

Examples

Get Information About CDF File

Get information about the sample file, `example.cdf`.

```
info = cdfinfo('example.cdf')
```

```
info =

             Filename: 'example.cdf'
          FileModDate: '10-May-2010 21:35:00'
             FileSize: 1310
               Format: 'CDF'
        FormatVersion: '2.7.0'
         FileSettings: [1x1 struct]
             Subfiles: {}
            Variables: {6x6 cell}
      GlobalAttributes: [1x1 struct]
    VariableAttributes: [1x1 struct]
```

View information about the variables in the file.

```
info.Variables
```

```
ans =

  Columns 1 through 5

    'Time'                [1x2 double]    [24]    'epoch'     'T/'
    'Longitude'           [1x2 double]    [ 1]    'int8'      'F/FT'
    'Latitude'            [1x2 double]    [ 1]    'int8'      'F/TF'
    'Data'                [1x3 double]    [ 1]    'double'    'T/TTT'
```

```
    'multidimensional'    [1x4 double]    [ 1]    'uint8'     'T/TTTT'
    'Temperature'         [1x2 double]    [10]    'int16'     'T/TT'

  Column 6

    'Full'
    'Full'
    'Full'
    'Full'
    'Full'
    'Full'
```

See Also

`cdflib.setValidate` | `cdfread`

Introduced before R2006a

cdflib

Interact directly with CDF library

Description

MATLAB provides direct access to dozens of functions in the CDF library. Using these functions, you can read and write data, create variables, attributes, and entries, and take advantage of other features of the CDF library. To use these functions, you must be familiar with the CDF C interface. Documentation about CDF is available at the CDF website.

The MATLAB functions correspond to functions in the CDF library new Standard Interface. In most cases, the syntax of a MATLAB function is similar to the syntax of the corresponding CDF library function. To use these functions, you must prefix the function name with the package name, `cdflib`. For example, to use the CDF library function to open an existing CDF file, use this syntax:

```
cdfid = cdflib.open('example.cdf');
```

MATLAB supports CDF version 3.3.0. For copyright information, see the `cdfcopyright.txt` file.

The following tables list all of the functions in the MATLAB CDF library package, grouped by category.

Note: For information about MATLAB support for the Network Common Data Form (netCDF), which is a completely separate, incompatible format, see `netcdf`.

Library Information

cdflib.getConstantNames	Names of Common Data Format (CDF) library constants
cdflib.getConstantValue	Numeric value corresponding to Common Data Format (CDF) library constant

cdflib.getFileBackward	Return current backward compatibility mode setting
cdflib.getLibraryCopyright	Copyright notice of Common Data Format (CDF) library
cdflib.getLibraryVersion	Library version and release information
cdflib.getValidate	Library validation mode
cdflib.setFileBackward	Set backward compatibility mode
cdflib.setValidate	Specify library validation mode

File Operations

cdflib.close	Close Common Data Format (CDF) file
cdflib.create	Create Common Data Format (CDF) file
cdflib.delete	Delete existing Common Data Format (CDF) file
cdflib.getCacheSize	Number of cache buffers used
cdflib.getChecksum	Checksum mode
cdflib.getCompression	Compression settings
cdflib.getCompressionCacheSize	Number of compression cache buffers
cdflib.getCopyright	Copyright notice in Common Data Format (CDF) file

cdflib.getFormat	Format of Common Data Format (CDF) file
cdflib.getMajority	Majority of variables
cdflib.getName	Name of Common Data Format (CDF) file
cdflib.getReadOnlyMode	Read-only mode
cdflib.getStageCacheSize	Number of cache buffers for staging
cdflib.getVersion	Common Data Format (CDF) library version and release information
cdflib.inquire	Basic characteristics of Common Data Format (CDF) file
cdflib.open	Open existing Common Data Format (CDF) file
cdflib.setCacheSize	Specify number of dotCDF cache buffers
cdflib.setChecksum	Specify checksum mode
cdflib.setCompression	Specify compression settings
cdflib.setCompressionCacheSize	Specify number of compression cache buffers
cdflib.setFormat	Specify format of Common Data Format (CDF) file
cdflib.setMajority	Specify majority of variables
cdflib.setReadOnlyMode	Specify read-only mode

cdflib.setStageCacheSize	Specify number of staging cache buffers for Common Data Format (CDF) file

Variables

cdflib.closeVar	Close specified variable from multifile format Common Data Format (CDF) file
cdflib.createVar	Create new variable
cdflib.deleteVar	Delete variable
cdflib.deleteVarRecords	Delete range of records from variable
cdflib.getVarAllocRecords	Number of records allocated for variable
cdflib.getVarBlockingFactor	Blocking factor for variable
cdflib.getVarCacheSize	Number of multifile cache buffers
cdflib.getVarCompression	Information about compression used by variable
cdflib.getVarData	Single value from record in variable
cdflib.getVarMaxAllocRecNum	Maximum allocated record number for variable
cdflib.getVarMaxWrittenRecNum	Maximum written record number for variable
cdflib.getVarsMaxWrittenRecNum	Maximum written record number for CDF file

cdflib.getVarName	Variable name, given variable number
cdflib.getVarNum	Variable number, given variable name
cdflib.getVarNumRecsWritten	Number of records written to variable
cdflib.getVarPadValue	Pad value for variable
cdflib.getVarRecordData	Entire record for variable
cdflib.getVarReservePercent	Compression reserve percentage for variable
cdflib.getVarSparseRecords	Information about how variable handles sparse records
cdflib.hyperGetVarData	Read hyperslab of data from variable
cdflib.hyperPutVarData	Write hyperslab of data to variable
cdflib.inquireVar	Information about variable
cdflib.putVarData	Write single value to variable
cdflib.putVarRecordData	Write entire record to variable
cdflib.renameVar	Rename existing variable
cdflib.setVarAllocBlockRecords	Specify range of records to be allocated for variable
cdflib.setVarBlockingFactor	Specify blocking factor for variable

cdflib.setVarCacheSize	Specify number of multi-file cache buffers for variable
cdflib.setVarCompression	Specify compression settings used with variable
cdflib.setVarInitialRecs	Specify initial number of records written to variable
cdflib.setVarPadValue	Specify pad value used with variable
cdflib.SetVarReservePercent	Specify reserve percentage for variable
cdflib.setVarsCacheSize	Specify number of cache buffers used for all variables
cdflib.setVarSparseRecords	Specify how variable handles sparse records

Attributes

cdflib.createAttr	Create attribute
cdflib.deleteAttr	Delete attribute
cdflib.deleteAttrEntry	Delete attribute entry
cdflib.deleteAttrgEntry	Delete entry in global attribute
cdflib.getAttrEntry	Value of entry in attribute with variable scope
cdflib.getAttrgEntry	Value of entry in global attribute

cdflib.getAttrMaxEntry	Number of last entry for variable attribute
cdflib.getAttrMaxgEntry	Number of last entry for global attribute
cdflib.getAttrName	Name of attribute, given attribute number
cdflib.getAttrNum	Attribute number, given attribute name
cdflib.getAttrScope	Scope of attribute
cdflib.getNumAttrEntries	Number of entries for attribute with variable scope
cdflib.getNumAttrgEntries	Number of entries for attribute with global scope
cdflib.getNumAttributes	Number of attributes with variable scope
cdflib.getNumgAttributes	Number of attributes with global scope
cdflib.inquireAttr	Information about attribute
cdflib.inquireAttrEntry	Information about entry in attribute with variable scope
cdflib.inquireAttrgEntry	Information about entry in attribute with global scope
cdflib.putAttrEntry	Write value to entry in attribute with variable scope
cdflib.putAttrgEntry	Write value to entry in attribute with global scope

cdflib.renameAttr	Rename existing attribute

Utility Functions

cdflib.computeEpoch	Convert time value to `CDF_EPOCH` value
cdflib.computeEpoch16	Convert time value to `CDF_EPOCH16` value
cdflib.epoch16Breakdown	Convert `CDF_EPOCH16` value to time value
cdflib.epochBreakdown	Convert `CDF_EPOCH` value into time value

See Also

`cdfread` | `cdfinfo`

Tutorials

- “Import CDF Files Using Low-Level Functions”
- “Export to CDF Files”

cdflib.close

Close Common Data Format (CDF) file

Syntax

```
cdflib.close(cdfId)
```

Description

`cdflib.close(cdfId)` closes the specified CDF file. `cdfId` identifies the CDF file.

You must close a CDF to guarantee that all modifications you made since opening the CDF are actually written to the file.

Examples

Open the example CDF file and then close it.

```
cdfid = cdflib.open('example.cdf');
cdflib.close(cdfid)
```

References

This function corresponds to the CDF library C API routine `CDFcloseCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.open` | `cdflib.create`

cdflib.closeVar

Close specified variable from multifile format Common Data Format (CDF) file

Syntax

```
cdflib.closeVar(cdfId,varNum)
```

Description

`cdflib.closeVar(cdfId,varNum)` closes a variable in a multifile format CDF.

`cdfId` identifies the CDF file and `varNum` is a numeric value that specifies the variable. Variable identifiers (variable numbers) are zero-based.

For multifile CDFs, you must close all open variable files to guarantee that all modifications you have made are actually written to the CDF file(s). You do not need to call this function for variables in a single-file format CDF.

Examples

Create a multifile CDF, create a variable, and then close the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_multifile.cdf');

% Make it a multifile format CDF
cdflib.setFormat(cdfid,'MULTI_FILE')

% Create a variable in the CDF.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

%  Close the variable.
cdflib.closeVar(cdfid, varnum)

% Clean up
cdflib.delete(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFclosezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarNum` | `cdflib.setFormat` | `cdflib.getFormat`

cdflib.computeEpoch

Convert time value to CDF_EPOCH value

Syntax

```
epoch = cdflib.computeEpoch(timeval)
```

Description

`epoch = cdflib.computeEpoch(timeval)` converts the time value specified by `timeval` into a CDF_EPOCH value.

Input Arguments

`timeval`

7-by-1 time vector. The following table describes the time components.

Component	Description
year	AD e.g. 1994
month	1–12
day	1–31
hour	0–23
minute	0–59
second	0–59
millisecond	0–999

Output Arguments

`epoch`

MATLAB `double` representing a CDF_EPOCH time value.

Examples

Convert a time value into a `CDF_EPOCH` value.

```
timeval = [1999 12 31 23 59 59 0];
epoch = cdflib.computeEpoch(timeval);
```

References

This function corresponds to the CDF library C API routine `computeEPOCH`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch16` | `cdflib.epochBreakdown` | `cdflib.epoch16Breakdown`

cdflib.computeEpoch16

Convert time value to CDF_EPOCH16 value

Syntax

```
epoch16 = cdflib.computeEpoch16(timeval)
```

Description

`epoch16 = cdflib.computeEpoch16(timeval)` converts the time value specified by `timeval` into a CDF_EPOCH16 value.

Input Arguments

`timeval`

10-by-1 time vector. The following table describes the time components. To specify multiple time values, use additional columns.

Component	Description
year	AD e.g. 1994
month	1–12
day	1–31
hour	0–23
minute	0–59
second	0–59
millisecond	0–999
microsecond	0–999
nanosecond	0–999
picosecond	0–999

Output Arguments

`epoch16`

CDF Epoch16 time value. If the input argument `timeval` has `m-by-10` elements, the return value `epoch16` will have size `2-by-m`

Examples

Convert the time value into an `CDF_EPOCH16` value:

```
timeval = [1999; 12; 31; 23; 59; 59; 50; 100; 500; 999];
epoch16 = cdflib.computeEpoch16(timeval);
```

References

This function corresponds to the CDF library C API routine `computeEPOCH16`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch` | `cdflib.epochBreakdown` | `cdflib.epoch16Breakdown`

cdflib.create

Create Common Data Format (CDF) file

Syntax

```
cdfId = cdflib.create(filename)
```

Description

`cdfId = cdflib.create(filename)` creates a new CDF file with the name specified by the text string *`filename`*. Returns the CDF file identifier `cdfId`.

Examples

Create a CDF file. To run this example, you must have write permission in your current directory.

```
cdfId = cdflib.create('myfile.cdf');

% Clean up
cdflib.delete(cdfId);

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFcreateCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.open` | `cdflib.close`

cdflib.createAttr

Create attribute

Syntax

```
attrnum = cdflib.createAttr(cdfId,attrName,scope)
```

Description

`attrnum = cdflib.createAttr(cdfId,attrName,scope)` creates an attribute in a CDF file with the specified scope.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrName`

Text string that specifies the name you want to assign to the attribute.

`scope`

One of the following text strings, or its numeric equivalent, that specifies the scope of the attribute.

Text String	Description
`'global_scope'`	Attribute applies to the CDF as a whole.
`'variable_scope'`	Attribute applies only to the variable

To get the numeric equivalent of these text string constants, use the `cdflib.getConstantValue` function.

Output Arguments

attrNum

Numeric value identifying the attribute. Attribute numbers are zero-based.

Examples

Create a CDF, and then create an attribute in the CDF. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create attribute
attrNum = cdflib.createAttr(cdfId,'Purpose','global_scope');

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFcreateAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrNum` | `cdflib.deleteAttr` | `cdflib.getConstantValue` | `cdflib.getConstantNames`

cdflib.createVar

Create new variable

Syntax

```
varnum = cdflib.createVar(cdfId, varname, datatype, numElements,
dims, recVariance, dimVariance)
```

Description

`varnum = cdflib.createVar(cdfId, varname, datatype, numElements, dims, recVariance, dimVariance)` creates a new variable in the Common Data Format (CDF) file with the specified characteristics.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varname`

Text string that specifies the name you want to assign to the variable.

`datatype`

Data type of the variable. One of the following text strings, or its numeric equivalent, that specifies a valid CDF data type.

CDF Data Type	Description
`CDF_BYTE`	1-byte, signed integer
`CDF_CHAR`	1 byte, signed character data type that maps to the MATLAB `char` class
`CDF_INT1`	1-byte, signed integer
`CDF_UCHAR`	1 byte, unsigned character data type that maps to the MATLAB `uint8` class

CDF Data Type	Description
CDF_UINT1	1-byte, unsigned integer
CDF_INT2	2-byte, signed integer
CDF_UINT2	2-byte, unsigned integer
CDF_INT4	4-byte, signed integer
CDF_UINT4	4-byte, unsigned integer
CDF_FLOAT	4-byte, floating point
CDF_REAL4	4-byte, floating point
CDF_REAL8	8-byte, floating point.
CDF_DOUBLE	8-byte, floating point
CDF_EPOCH	8-byte, floating point
CDF_EPOCH16	two 8-byte, floating point

numElements

Number of elements per datum. Value should be 1 for all data types, except CDF_CHAR and CDF_UCHAR.

dims

A vector of the dimensions extents; empty if there are no dimension extents.

recVariance

Specifies record variance: true or false.

dimVariance

A vector of logicals; empty if there are no dimensions.

Output Arguments

varNum

The numeric identifier for the variable. Variable numbers are zero-based.

Examples

Create a CDF file and then create a variable named `'Time'` in the CDF. The variable has no dimensions and varies across records. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no variables.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Retrieve info about the file again to verify variable was created.
% Note value of numVars field is now 1.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 1
    numvAttrs: 0
    numgAttrs: 0

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFcreatezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteVar` | `cdflib.closeVar`

cdflib.delete

Delete existing Common Data Format (CDF) file

Syntax

```
cdflib.delete(cdfId)
```

Description

`cdflib.delete(cdfId)` deletes the existing CDF file specified by the identifier `cdfId`. If the CDF file is a multi-file format CDF, the `cdflib.delete` function also deletes the variable files (having file extensions of `.z0`, `.z1`, etc.).

Examples

Create a CDF file, and then delete it. To run this example, you must be in a writable folder.

```
cdfId = cdflib.create('mytempfile.cdf');

% Verify that the file was created.
ls *.cdf

mytempfile.cdf

% Delete the file.
cdflib.delete(cdfId)

% Verify that the file no longer exists.
ls *.cdf
```

References

This function corresponds to the CDF library C API routine `CDFdeleteCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.create` | `cdflib.setFormat`

cdflib.deleteAttr

Delete attribute

Syntax

```
cdflib.deleteAttr(cdfId,attrNum)
```

Description

`cdflib.deleteAttr(cdfId,attrNum)` deletes the specified attribute from the CDF file.

`cdfId` identifies the Common Data Format (CDF) file.`attrNum` is a numeric identifier that specifies the attribute. Attribute numbers are zero-based.

Examples

Create a CDF file, and then create an attribute in the file. Then delete the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create attribute.
attrNum = cdflib.createAttr(cdfId,'Purpose','global_scope');

% Prove it exists.
anum = cdflib.getAttrNum(cdfid,'Purpose')

anum =

     0

% Delete the attribute.
cdflib.deleteAttr(cdfid,attrNum);


% Clean up
```

```
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrNum`

cdflib.deleteAttrEntry

Delete attribute entry

Syntax

```
cdflib.deleteAttrEntry(cdfId,attrNum,entryNum)
```

Description

`cdflib.deleteAttrEntry(cdfId,attrNum,entryNum)` deletes an entry from an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

`entryNum`

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Examples

Create a CDF, and then create an attribute in the file. Write a value to an entry for the attribute, and then delete the entry. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');
```

```
% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create an attribute with variable scope in the file.
attrNum = cdflib.createAttr(cdfid,'my_var_scope_attr','variable_scope');

% Write a value to an entry for the attribute
cdflib.putAttrEntry(cdfid,attrNum,0,'CDF_CHAR','My attr value');

% Get the value of the attribute entry
value = cdflib.getAttrEntry(cdfid,attrNum,0)

value =

My attr value

% Delete the entry
cdflib.deleteAttrEntry(cdfid,attrNum,0);

% Now try to view the value of the entry
% Should return NO_SUCH_ENTRY failure.
value = cdflib.getAttrEntry(cdfid,attrNum,0)  % Should fail

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteAttr`

cdflib.deleteAttrgEntry

Delete entry in global attribute

Syntax

```
cdflib.deleteAttrgEntry(cdfId,attrNum,entryNum)
```

Description

`cdflib.deleteAttrgEntry(cdfId,attrNum,entryNum)` deletes an entry from a global attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Examples

Create a CDF and create a global attribute in the file. Write a value to an entry for the attribute and then delete the entry. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');
```

```
% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create an attribute with global scope in the file.
attrNum = cdflib.createAttr(cdfid,'my_global_attr','global_scope');

% Write a value to an entry for the attribute
cdflib.putAttrgEntry(cdfid,attrNum,0,'CDF_CHAR','My global attr');

% Get the value of the global attribute entry
value = cdflib.getAttrgEntry(cdfid,attrNum,0)

value =

My global attr

% Delete the entry
cdflib.deleteAttrgEntry(cdfid,attrNum,0);

% Now try to view the value of the entry
% Should return NO_SUCH_ENTRY failure.
value = cdflib.getAttrgEntry(cdfid,attrNum,0)  % Should fail

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteAttr` | `cdflib.deleteAttrEntry`

cdflib.deleteVar

Delete variable

Syntax

```
cdflib.deleteVar(cdfId,varNum)
```

Description

`cdflib.deleteVar(cdfId,varNum)` deletes a variable from a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that specifies the variable. Variable numbers are zero-based.

Examples

Create a CDF, create a variable in the CDF, and then delete it.

```
cdfid = cdflib.create('mycdf.cdf');

% Initially the file contains no variables.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a variable in the CDF.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);
```

```
% Retrieve info about the variable in the CDF.

info = cdflib.inquireVar(cdfid, 0)

info =

           name: 'Time'
       datatype: 'cdf_int1'
    numElements: 1
           dims: []
    recVariance: 1
    dimVariance: []

% Delete the variable from the CDF

cdflib.deleteVar(cdfid,0);

% Check to see if the variable was deleted from the CDF.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeletezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

cdflib.deleteVarRecords

Delete range of records from variable

Syntax

```
cdflib.deleteVarRecords(cdfId,varNum,startRec,endRec)
```

Description

`cdflib.deleteVarRecords(cdfId,varNum,startRec,endRec)` deletes a range of records from a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable. Variable numbers are zero-based.

`startRec`

Numeric value that specifies the record at which to start deleting records. Record numbers are zero-based.

`endRec`

Numeric value that specifies the record at which to stop deleting records. Record numbers are zero-based.

Examples

Make a writable copy of the example CDF, get the number of a variable in the CDF, and delete specific records in the variable. To run this example, you must be in a writable folder.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.cdf');
copyfile(srcFile,'myfile.cdf');
fileattrib('myfile.cdf','+w');
cdfid = cdflib.open('myfile.cdf');
varnum = cdflib.getVarNum(cdfid,'Temperature');
cdflib.deleteVarRecords(cdfid,varnum,1,2);
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFdeletezVarRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarNumRecsWritten` | `cdflib.putVarRecordData`

cdflib.epoch16Breakdown

Convert CDF_EPOCH16 value to time value

Syntax

```
timeVec = cdflib.epoch16Breakdown(epoch16Time)
```

Description

`timeVec = cdflib.epoch16Breakdown(epoch16Time)` convert a CDF_EPOCH16 value into a time vector. `timeVec` will have 10-by-n elements, where n is the number of CDF_EPOCH16 values.

The following table describes the time value components.

timeVec Element	Description	Valid Values
`timeVec(1,:)`	Year AD	e.g. 1994
`timeVec(2,:)`	Month	1–12
`timeVec(3,:)`	Day	1–31
`timeVec(4,:)`	Hour	0–23
`timeVec(5,:)`	Minute	0–59
`timeVec(6,:)`	Second	0–59
`timeVec(7,:)`	Millisecond	0–999
`timeVec(8,:)`	Microsecond	0–999
`timeVec(9,:)`	Nanosecond	0–999
`timeVec(10,:)`	Picosecond	0–999

Examples

Convert CDF_EPOCH16 value into time value.

```
timeval = [1999; 12; 31; 23; 59; 59; 50; 100; 500; 999];
epoch16 = cdflib.computeEpoch16(timeval);

timevec = cdflib.epoch16Breakdown(epoch16)

timevec =

        1999
          12
          31
          23
          59
          59
          50
         100
         500
         999
```

References

This function corresponds to the CDF library C API routine `EPOCH16breakdown`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch16`

cdflib.epochBreakdown

Convert CDF_EPOCH value into time value

Syntax

```
timeVec = cdflib.epochBreakdown(epochTime)
```

Description

`timeVec = cdflib.epochBreakdown(epochTime)` decomposes the CDF_EPOCH value, `epochTime` value into individual time components. `timeVec` will have 7-by-*n* elements, where *n* is the number of CDF_EPOCH values in `epochTime`.

The return value `timeVec` has the following elements:

timeVec Element	Description	Valid Values
`timeVec(1,:)`	Year AD	e.g. 1994
`timeVec(2,:)`	Month	1–12
`timeVec(3,:)`	Day	1–31
`timeVec(4,:)`	Hour	0–23
`timeVec(5,:)`	Minute	0–59
`timeVec(6,:)`	Second	0–59
`timeVec(7,:)`	Millisecond	0–999

Examples

Convert a CDP_EPOCH value into a time vector.

```
% First convert a time vector into a CDF_EPOCH value
timeval = [1999 12 31 23 59 59 0];
epoch = cdflib.computeEpoch(timeval);
```

```
% Convert the CDF_EPOCH value into a time vector
timevec = cdflib.epochBreakdown(epoch)

timevec =

        1999
          12
          31
          23
          59
          59
           0
```

References

This function corresponds to the CDF library C API routine `EPOCHbreakdown`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch` | `cdflib.epochBreakdown` | `cdflib.epoch16Breakdown`

cdflib.getAttrEntry

Value of entry in attribute with variable scope

Syntax

```
value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)
```

Description

`value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)` returns the value of an attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

`entryNum`

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Output Arguments

`Value`

Value of the entry.

Examples

Open the example CDF and get the value of an entry associated with an attribute with variable scope in the file.

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrEntry(cdfid,3,0)

dtype =

cdf_char


numel =

    10

% Get the value of the entry for this attribute.
% Note that it's a character string, 10 characters in length
value = cdflib.getAttrEntry(cdfid,3,0)

value =

Time value

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putAttrEntry` | `cdflib.getAttrgEntry` | `cdflib.putAttrgEntry`

cdflib.getAttrgEntry

Value of entry in global attribute

Syntax

```
value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)
```

Description

`value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)` returns the value of a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Output Arguments

`Value`

Value of the entry.

Examples

Open the example CDF, and then get the value of an entry associated with a global attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attributes have global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE

% Get information about the first entry for global attribute
[dtype numel] = cdflib.inquireAttrgEntry(cdfid,0,0)

dtype =

cdf_char


numel =

    23

% Get the value of the first entry for this global attribute.
value = cdflib.getAttrgEntry(cdfid,0,0)

value =

This is a sample entry.

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putAttrgEntry` | `cdflib.getAttrEntry` | `cdflib.putAttrEntry`

cdflib.getAttrMaxEntry

Number of last entry for variable attribute

Syntax

```
maxEntry = cdflib.getAttrMaxEntry(cdfId,attrNum)
```

Description

`maxEntry = cdflib.getAttrMaxEntry(cdfId,attrNum)` returns the number of the last entry for an attribute in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file.

`attrNum` is a numeric value that specifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Output Arguments

`maxEntry`

Entry number of the last entry in the attribute. Entry numbers are zero-based.

Examples

Open the example CDF and get the number of the last entry associated with an attribute with variable scope in the file:

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE


% Get the number of the last entry for this attribute.
entrynum = cdflib.getAttrMaxEntry(cdfid,3)

entrynum =

    3


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrMaxzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrMaxgEntry`

cdflib.getAttrMaxgEntry

Number of last entry for global attribute

Syntax

```
maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)
```

Description

`maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)` returns the last entry number of a global attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

Output Arguments

`maxEntry`

Entry number of the last entry in the attribute. Entry numbers are zero-based.

Examples

Open the example CDF and get the number of the last entry associated with a global attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attribute are of global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE


% Get the number of the last entry for this attribute.
entrynum = cdflib.getAttrMaxgEntry(cdfid,0)

entrynum =

     4

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrMaxgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrMaxEntry`

cdflib.getAttrName

Name of attribute, given attribute number

Syntax

```
name = cdflib.getAttrName(cdfId,attrNum)
```

Description

`name = cdflib.getAttrName(cdfId,attrNum)` returns the name of an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based.

Output Arguments

`name`

Text string specifying the name of the attribute.

Examples

Open the example CDF and get name of an attribute.

```
cdfid = cdflib.open('example.cdf');
```

```
% Get name of the first attribute in the file.
attrName = cdflib.getAttrName(cdfId,0)

attrName =

SampleAttribute

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr`

cdflib.getAttrNum

Attribute number, given attribute name

Syntax

```
attrNum = cdflib.getAttrNum(cdfId,attrName)
```

Description

`attrNum = cdflib.getAttrNum(cdfId,attrName)` returns the number of an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrName`

Text string specifying the name of an attribute.

Output Arguments

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based.

Examples

Open the example CDF and get the attribute number associated with the `SampleAttribute` attribute.

```
cdfid = cdflib.open('example.cdf');

attrNum = cdflib.getAttrNum(cdfid,'SampleAttribute')

attrNum =

    0

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrName`

cdflib.getAttrScope

Scope of attribute

Syntax

```
scope = cdflib.getAttrScope(cdfId,attrNum)
```

Description

`scope = cdflib.getAttrScope(cdfId,attrNum)` returns the scope of an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that specifies the attribute. Attribute numbers are zero-based.

Output Arguments

`scope`

Either of the following text strings, or its numeric equivalent.

Text String	Description
`'GLOBAL_SCOPE'`	Attribute applies to the CDF as a whole.
`'VARIABLE_SCOPE'`	Attribute applies only to the variable.

To get the numeric equivalent of these text string constants, use the `cdflib.getConstantValue` function.

Examples

Open example CDF and get the scope of the first attribute in the file:

```
cdfid = cdflib.open('example.cdf');

attrScope = cdflib.getAttrScope(cdfid,0)

attrScope =

GLOBAL_SCOPE

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrScope`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrName` | `cdflib.getConstantValue`

cdflib.getCacheSize

Number of cache buffers used

Syntax

```
numBuffers = cdflib.getCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getCacheSize(cdfId)` returns the number of cache buffers used for the Common Data Format (CDF) file identified by `cdfId`. For a discussion of cache schemes, see the *CDF User's Guide*.

Examples

Open the example CDF file and get the cache size:

```
cdfid = cdflib.open('example.cdf');

numBuf = cdflib.getCacheSize(cdfid)

numBuf =

   300

% Clean up
cdflib.close(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCacheSize`

cdflib.getChecksum

Checksum mode

Syntax

```
mode = cdflib.getChecksum(cdfId)
```

Description

`mode = cdflib.getChecksum(cdfId)` returns the checksum mode of the Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`mode`

Either of the following text strings or its numeric equivalent.

`'MD5_CHECKSUM'`	File uses MD5 checksum.
`'NO_CHECKSUM'`	File does not use a checksum.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Open the example CDF file, and then get the checksum mode of the file:

```
cdfid = cdflib.open('example.cdf');

checksummode = cdflib.getChecksum(cdfid)


checksummode =

NO_CHECKSUM

% Clean up
cdflib.close(cdfid);
clear cdfid;
```

References

This function corresponds to the CDF library C API routine `CDFgetChecksum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setChecksum` | `cdflib.getConstantValue`

cdflib.getCompression

Compression settings

Syntax

```
[ctype,cparms,cpercentage] = cdflib.getCompression(cdfId)
```

Description

`[ctype,cparms,cpercentage] = cdflib.getCompression(cdfId)` returns information about the compression settings of a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`ctype`

Text string specifying compression type, such as `'HUFF_COMPRESSION'`. If the CDF does not use compression, the function returns the string `'NO_COMPRESSION'`. For a list of supported compression types, see `cdflib.setCompression`.

`cparms`

The value of the parameter associated with the type of compression. For example, for the `'RLE_COMPRESSION'` compression type, the parameter specifies the style of run-length encoding. For a list of parameters supported by each compression type, see `cdflib.setCompression`.

`cpercentage`

The rate of compression, expressed as a percentage.

Examples

Open the example CDF file and check the compression settings in the file.

```
cdfId = cdflib.open('example.cdf');

[ctype, cparms, cpercentage] = cdflib.getCompression(cdfId)

ctype =

GZIP_COMPRESSION

cparms =

     7

cper =

    26

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompression` | `cdflib.getVarCompression` | `cdflib.setVarCompression`

cdflib.getCompressionCacheSize

Number of compression cache buffers

Syntax

```
numBuffers = cdflib.getCompressionCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getCompressionCacheSize(cdfId)` returns the number of cache buffers used for the compression scratch Common Data Format (CDF) file. `cdfId` identifies the CDF file. For a discussion of cache schemes, see the *CDF User's Guide*.

Examples

Open the example CDF file and check the compression cache size of the file:

```
cdfId = cdflib.open('example.cdf');

numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =

    80

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCompressionCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

See Also

`cdflib.setCompressionCacheSize`

cdflib.getConstantNames

Names of Common Data Format (CDF) library constants

Syntax

```
names = cdflib.getConstantNames()
```

Description

`names = cdflib.getConstantNames()` returns a cell array of text strings, where each text string is the name of a constant known to the Common Data Format (CDF) library.

Examples

Get a list of the names of CDF library constants.

```
names  = cdflib.getConstantNames()

names =

    'AHUFF_COMPRESSION'
    'ALPHAMVSD_ENCODING'
    'ALPHAMVSG_ENCODING'
    'ALPHAMVSI_ENCODING'
    'ALPHAOSF1_ENCODING'
    'CDF_BYTE'
    'CDF_CHAR'
     .
     .
     .
```

References

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue`

cdflib.getConstantValue

Numeric value corresponding to Common Data Format (CDF) library constant

Syntax

```
value = cdflib.getConstantValue(constantName)
```

Description

`value = cdflib.getConstantValue(constantName)` returns the numeric value of the CDF library constant specified by the text string *`constantName`*. To see a list of constant names, use `cdflib.getConstantNames`.

Examples

View the list of CDF library constants and get the numeric value corresponding to one of the constants.

```
% Retrieve a list of library constants
names = cdflib.getConstantNames();

value  = cdflib.getConstantValue(names{1})

value =

   3
```

References

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantNames`

cdflib.getCopyright

Copyright notice in Common Data Format (CDF) file

Syntax

```
copyright = cdflib.getCopyright(cdfId)
```

Description

`copyright = cdflib.getCopyright(cdfId)` returns the copyright notice in the CDF file identified by `cdfId`.

Examples

Create a CDF file, and then get the copyright notice in the file. To run this example, you must be in a writable folder.

```
cdfId = cdflib.create('your_file.cdf');

copyright  = cdflib.getCopyright(cdfId)

copyright =

Common Data Format (CDF)
(C) Copyright 1990-2009 NASA/GSFC
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 USA
(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)

% Clean up.
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCopyright`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getLibraryCopyright`

cdflib.getFileBackward

Return current backward compatibility mode setting

Syntax

```
mode = cdflib.getFileBackward()
```

Description

`mode = cdflib.getFileBackward()` returns the backward compatibility mode.

Output Arguments

`mode`

One of the following text strings:

`BACKWARDFILEon`	Backward compatibility mode is on.
`BACKWARDFILEoff`	Backward compatibility mode is off.

For more information about backward compatibility mode, see `cdflib.setFileBackward`.

Examples

```
mode = cdflib.getFileBackward

mode =

BACKWARDFILEoff
```

References

This function corresponds to the CDF library C API routine `CDFgetFileBackward`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setFileBackward` | `cdflib.getConstantValue`

cdflib.getFormat

Format of Common Data Format (CDF) file

Syntax

```
format = cdflib.setFormat(cdfId)
```

Description

`format = cdflib.setFormat(cdfId)` returns the format of the CDF file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`format`

Either of the following text strings, or its numeric equivalent.

`'SINGLE_FILE'`	The CDF is stored in a single file.
`'MULTI_FILE'`	The CDF is made up of multiple files.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Open the example CDF file and determine its file format:

```
cdfId = cdflib.open('example.cdf');
```

```
format = cdflib.getFormat(cdfId)

format =

   'SINGLE_FILE'

% Clean up.
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetFormat`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setFormat` | `cdflib.getConstantValue`

cdflib.getLibraryCopyright

Copyright notice of Common Data Format (CDF) library

Syntax

```
copyright = cdflib.getLibraryCopyright()
```

Description

`copyright = cdflib.getLibraryCopyright()` returns a text string containing the copyright notice of the CDF library.

Examples

Get the copyright of the CDF library.

```
copyright  = cdflib.getLibraryCopyright()

copyright =

Common Data Format (CDF)
(C) Copyright 1990-2008 NASA/GSFC
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 USA
(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)
```

References

This function corresponds to the CDF library C API routine `CDFgetLibraryCopyright`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCopyright`

cdflib.getLibraryVersion

Library version and release information

Syntax

```
[version,release,increment] = cdflib.getLibraryVersion()
```

Description

`[version,release,increment] = cdflib.getLibraryVersion()` returns information about the Common Data Format (CDF) library.

Output Arguments

`version`

Numeric value indicating the version number of the CDF library.

`release`

Numeric value indicating the release number of the CDF library.

`increment`

Numeric value indicating the increment number of the CDF library.

Examples

Get the version information of the CDF library:

```
[version, release, increment] = cdflib.getLibraryVersion()

version =

     3
```

```
release =

     3

increment =

     0
```

References

This function corresponds to the CDF library C API routine `CDFgetLibraryVersion`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVersion`

cdflib.getMajority

Majority of variables

Syntax

```
majority = cdflib.getMajority(cdfId)
```

Description

`majority = cdflib.getMajority(cdfId)` returns the majority of variables in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`majority`

Either of the following text strings, or its numeric equivalent.

`'ROW_MAJOR'`	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest. This is the default.
`'COLUMN_MAJOR'`	Fortran-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Open the example CDF file, and then determine the majority of variables in the file:

```
cdfId = cdflib.open('example.cdf');

majority = cdflib.getMajority(cdfId)

majority =

ROW_MAJOR

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetMajority`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setMajority` | `cdflib.getConstantValue`

cdflib.getName

Name of Common Data Format (CDF) file

Syntax

```
name = cdflib.getName(cdfId)
```

Description

`name = cdflib.getName(cdfId)` returns the name of the CDF file identified by `cdfId`.

Examples

Open the example CDF file and get the name of the file. The path name returned for your installation will be different.

```
cdfId = cdflib.open('example.cdf');

name = cdflib.getName(cdfId)

name =

yourinstallation\matlab\toolbox\matlab\demos\example

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.open` | `cdflib.create`

cdflib.getNumAttrEntries

Number of entries for attribute with variable scope

Syntax

```
nentries = cdflib.getNumAttrEntries(cdfId,attrNum)
```

Description

`nentries = cdflib.getNumAttrEntries(cdfId,attrNum)` returns the number of entries for the specified attribute in the Common Data Format (CDF) file.

`cdfId` identifies the CDF file.

`attrNum` is a numeric value that specifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Examples

Open the example CDF, find an attribute with variable scope, and determine how many entries are associated with the attribute:

```
cdfid = cdflib.open('example.cdf');

% Get the number of an attribute
% with variable scope
attrNum = cdflib.getAttrNum(cdfid,'Description');

% Check that scope of attribute is variable
attrScope = cdflib.getAttrScope(cdfid,attrNum)

VARIABLE_SCOPE

% Detemine the number of entries for the attribute
attrEntries = cdflib.getNumAttrEntries(cdfid,attrNum)

attrEntries =
```

```
    4

% Clean up
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFgetNumAttrzEntries`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrScope`

cdflib.getNumAttrgEntries

Number of entries for attribute with global scope

Syntax

```
nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)
```

Description

`nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)` returns the number of entries written for the specified global attribute in the Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `attrNum` is a numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

Examples

Open the example CDF and find out how many entries are associated with a global attribute in the file.

```
cdfid = cdflib.open('example.cdf');

% The first attribute is a global attribute.
attrgEntries = cdflib.getNumAttrgEntries(cdfid,0)

attrgEntries =

    3

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetNumAttrgEntries`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumAttrEntries`

cdflib.getNumAttributes

Number of attributes with variable scope

Syntax

```
numAtts = cdflib.getNumAttributes(cdfId)
```

Description

`numAtts = cdflib.getNumAttributes(cdfId)` returns the total number of attributes with variable scope in a Common Data Format (CDF) file. `cdfId` identifies the CDF file.

Examples

Open the example CDF and find out how many attributes in the file have variable scope:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of attributes with variable scope
numAttrs = cdflib.getNumAttributes(cdfid)

numAttrs =

    1

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetNumvAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumgAttributes`

cdflib.getNumgAttributes

Number of attributes with global scope

Syntax

```
ngatts = cdflib.getNumgAttributes(cdfId)
```

Description

`ngatts = cdflib.getNumgAttributes(cdfId)` returns the total number of global attributes in a Common Data Format (CDF) file. `cdfId` identifies the CDF file.

Examples

Open the example CDF and find out how many global attributes are in the file:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of global attributes in the file.
numgAttrs = cdflib.getNumgAttributes(cdfid)

numgAttrs =

    3

% Clean up
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFgetNumgAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumAttributes`

cdflib.getReadOnlyMode

Read-only mode

Syntax

```
mode = cdflib.getReadOnlyMode(cdfId)
```

Description

`mode = cdflib.getReadOnlyMode(cdfId)` returns the read-only mode of a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`mode`

Either of the following text strings or its numeric equivalent.

`'READONLYon'`	CDF is in read-only mode
`'READONLYoff'`	CDF can be modified.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Open the example CDF file and determine its current read-only status:

```
cdfId = cdflib.open('example.cdf');

mode = cdflib.getReadOnlyMode(cdfId)

mode =

READONLYoff

% Clean up.
cdflib.close(cdfId);
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetReadOnlyMode`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setReadOnlyMode` | `cdflib.getConstantValue`

cdflib.getStageCacheSize

Number of cache buffers for staging

Syntax

```
numBuffers = cdflib.getStageCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getStageCacheSize(cdfId)` returns the number of cache buffers used for the staging scratch file of the Common Data Format (CDF) file. For more information about cache buffers, see the *CDF User's Guide*.

`cdfId` identifies the CDF file.

Examples

Open the example CDF file and determine the number of cache buffers used for staging:

```
cdfId = cdflib.open('example.cdf');

numBuf = cdflib.getStageCacheSize(cdfId)

numBuf =

   125

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetStageCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setStageCacheSize`

cdflib.getValidate

Library validation mode

Syntax

```
mode = cdflib.getValidate()
```

Description

`mode = cdflib.getValidate()` returns the validation mode of the Common Data Format (CDF) library.

Output Arguments

`mode`

Either of the following text strings or its numeric equivalent.

`'VALIDATEFILEon'`	Validation mode is on. For information about validation mode, see `cdflib.setValidate`.
`'VALIDATEFILEoff'`	Validation mode is off.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Determine the current validation mode of the CDF library.

```
mode = cdflib.getValidate()
```

```
mode =
```

```
'VALIDATEFILEon'
```

References

This function corresponds to the CDF library C API routine `CDFgetValidate`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setValidate` | `cdflib.getConstantValue`

cdflib.getVarAllocRecords

Number of records allocated for variable

Syntax

```
numrecs = cdflib.getVarAllocRecords(cdfId, varNum)
```

Description

`numrecs = cdflib.getVarAllocRecords(cdfId, varNum)` returns the number of records allocated for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open example CDF and get the number of records allocated for a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records allocated for the
% first variable in the file.
numrecs = cdflib.getVarAllocRecords(cdfid,0)

numrecs =

    64


%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarAllocRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarAllocBlockRecords`

cdflib.getVarBlockingFactor

Blocking factor for variable

Syntax

```
blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)
```

Description

`blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)` returns the blocking factor for a variable in a Common Data Format (CDF) file. A variable's *blocking factor* specifies the minimum number of records the library allocates when you write to an unallocated record.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF and determine the blocking factor of a variable.

```
cdfid = cdflib.open('example.cdf');

cdflib.getVarBlockingFactor(cdfid,0)

ans =

     0

%  Clean up
cdflib.close(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarBlockingFactor`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarBlockingFactor`

cdflib.getVarCacheSize

Number of multifile cache buffers

Syntax

```
numBuffers = cdflib.getVarCacheSize(cdfId,varNum)
```

Description

`numBuffers = cdflib.getVarCacheSize(cdfId,varNum)` returns the number of cache buffers used for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable identifiers are zero-based.

This function applies only to multifile format CDFs. For more information about caching, see the *CDF User's Guide*.

Examples

Create a multifile CDF and retrieve the number of buffers being used for a variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*

your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
```

```
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarCacheSize`

cdflib.getVarCompression

Information about compression used by variable

Syntax

```
[ctype,cparams,percent] = cdflib.getVarCompression(cdfId, varNum)
```

Description

`[ctype,cparams,percent] = cdflib.getVarCompression(cdfId, varNum)` returns information about the compression used for a variable in a Common Data Format (CDF) File.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`ctype`

Text string identifying the type of compression. For a list of compression types, see `cdflib.setCompression`.

`cparams`

Any additional parameter required by the compression type.

`percent`

Numeric value indicating the level of compression, expressed as a percentage.

Examples

Open the example CDF file and check the compression settings of any variable.

```
cdfid = cdflib.open('example.cdf');

% Check the compression setting of any variable in the file
% The example checks the first variable (variable numbers are zero-based).
[ctype params percent] = cdflib.getVarCompression(cdfid,0)

ctype =

NO_COMPRESSION


params =

     []


percent =

   100

% Clean up
cdflib.close(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompression` | `cdflib.setVarCompression`

cdflib.getVarData

Single value from record in variable

Syntax

```
datum = cdflib.getVarData(cdfId,varNum,recNum,indices)
datum = cdflib.getVarData(cdfId,varNum,recNum)
```

Description

`datum = cdflib.getVarData(cdfId,varNum,recNum,indices)` returns a single value from a variable in a Common Data Format (CDF) file.

`datum = cdflib.getVarData(cdfId,varNum,recNum)` returns a single value from a variable with no dimensions in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying the variable containing the datum. Variable numbers are zero-based.

`recNum`

Numeric value identifying the location of the datum in the variable. In CDF terminology, this is called the *record number*. Record numbers are zero-based.

`indices`

Array of dimension indices within the record. Dimension indices are zero-based. If the variable has no dimensions, you can omit this parameter.

Output Arguments

datum

Value of the specified record.

Examples

Open the example CDF file and retrieve data associated with a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine how many variables are in the file.
info = cdflib.inquire(cdfid);

info.numVars

ans =

  5

% Determine if the first variable has dimensions.
varinfo = cdflib.inquireVar(cdfid,0);
vardims = varinfo.dims
vardims =

  []

% Get data from variable, without specifying dimensions.
datum = cdflib.getVarData(cdfid, varnum, recnum)

datum =

  6.3146e+013

% Get dimensions of another variable in file.
varinfo = cdflib.inquireVar(cdfid,3);
vardims = varinfo.dims
vardims =

  [4 2 2]
```

```
% Retrieve the first datum in the record. Indices are zero-based.
datum = cdflib.getVarData(cdfId,3,0,[0 0 0])

info =

    30

% Clean up.
cdflib.close(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putVarData` | `cdflib.getVarRecordData` | `cdflib.hyperGetVarData`

cdflib.getVarMaxAllocRecNum

Maximum allocated record number for variable

Syntax

```
maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)
```

Description

`maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)` returns the record number of the maximum allocated record for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers and record numbers are zero-based.

Examples

Open example CDF and get the maximum allocated record number for a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine maximum record number for variable in file.
maxRecNum = cdflib.getVarMaxAllocRecNum(cdfid,0)

maxRecNum =

    63

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarMaxAllocRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

cdflib.getVarMaxWrittenRecNum

Maximum written record number for variable

Syntax

```
maxrec = cdflib.getVarMaxwrittenRecNum(cdfId,varNum)
```

Description

`maxrec = cdflib.getVarMaxwrittenRecNum(cdfId,varNum)` returns the record number of the maximum record written for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers and record numbers are zero-based.

Examples

Open the example CDF, and then determine the maximum number of records written to a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number records written to variable.
numRecs = cdflib.getVarNumRecsWritten(cdfid,0)

numRecs =

    24


% Determine the maximum record number of the records written
maxRecNum = cdflib.getVarMaxWrittenRecNum(cdfid,0)

maxRecNum =

    23
```

```
%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarMaxWrittenRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxAllocRecNum` | `cdflib.getVarNumRecsWritten`

cdflib.getVarsMaxWrittenRecNum

Maximum written record number for CDF file

Syntax

```
maxrec = cdflib.getVarsMaxwrittenRecNum(cdfId)
```

Description

`maxrec = cdflib.getVarsMaxwrittenRecNum(cdfId)` returns the maximum record number written for all variables in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. Record numbers are zero-based.

Examples

Open the example CDF, and then determine the maximum number of records written to the file:

```
cdfid = cdflib.open('example.cdf');

% Determine the maximum record number of the records written
maxRecNum = cdflib.getVarsMaxWrittenRecNum(cdfid)

maxRecNum =

    23

%  Clean up
cdflib.close(cdfid)
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarsMaxWrittenRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

cdflib.getVarName

Variable name, given variable number

Syntax

```
name = cdflib.getVarName(cdfId,varNum)
```

Description

`name = cdflib.getVarName(cdfId,varNum)` returns the name of the variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.`name` is a text string specifying the name.

Examples

Open the example CDF, and then get the name of a variable in the file:

```
cdfid = cdflib.open('example.cdf');

name = cdflib.getVarName(cdfid,1)

name =

Longitude

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireVar`

cdflib.getVarNum

Variable number, given variable name

Syntax

```
varNum = cdflib.getVarNum(cdfId,varname)
```

Description

`varNum = cdflib.getVarNum(cdfId,varname)` returns the identifier (variable number) for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. *`varname`* is a text string that identifies the variable. Variable names are case-sensitive.

Examples

Open example CDF, and then get the number of a variable named `Longitude`:

```
cdfid = cdflib.open('example.cdf');

varNum = cdflib.getVarNum(cdfid,'Longitude')

varNum =

     1

%  Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarName`

cdflib.getVarNumRecsWritten

Number of records written to variable

Syntax

```
numrecs = cdflib.getVarNumRecsWritten(cdfId,varNum)
```

Description

`numrecs = cdflib.getVarNumRecsWritten(cdfId,varNum)` returns the total number of records written to a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF, and then determine the number of records written to a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records written to the variable.
numRecs = cdflib.getVarNumRecsWritten(cdfid,0)

numRecs =

    24

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarNumRecsWritten`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

cdflib.getVarPadValue

Pad value for variable

Syntax

```
padvalue = cdflib.getVarPadValue(cdfId,varNum)
```

Description

`padvalue = cdflib.getVarPadValue(cdfId,varNum)` returns the pad value used with a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF, and then determine the pad value for a variable:

```
cdfid = cdflib.open('example.cdf');

% Check pad value of variable in the file.
padval = cdflib.getVarPadValue(cdfid,0)

padval =

     0

%  Clean up.
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarPadValue`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarPadValue`

cdflib.getVarRecordData

Entire record for variable

Syntax

```
data = cdflib.getVarRecordData(cdfId,varNum,recNum)
```

Description

`data = cdflib.getVarRecordData(cdfId,varNum,recNum)` returns the data in a record associated with a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable in the CDF file. Variable numbers are zero-based.

`recNum`

Numeric value that identifies the record in the variable. Record numbers are zero-based.

Output Arguments

`data`

Data in the record.

Examples

Open the example CDF, and then get the data associated with a record in a variable:

```
cdfid = cdflib.open('example.cdf');

% Get data in first record in first variable in file.
recData = cdflib.getVarRecordData(cdfid,0,0)

recData =

    6.3146e+013

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarRecordData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putVarRecordData` | `cdflib.getVarData` | `cdflib.hyperGetVarData`

cdflib.getVarReservePercent

Compression reserve percentage for variable

Syntax

```
percent = cdflib.getVarReservePercent(cdfId,varNum)
```

Description

`percent = cdflib.getVarReservePercent(cdfId,varNum)` returns the compression reserve percentage for a variable in a Common Data Format (CDF) file. This operation only applies to compressed variables.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF file, get the number of a compressed variable, and then determine the reserve percent for the variable.

```
cdfid = cdflib.open('example.cdf');
varnum = cdflib.getVarNum(cdfid,'Temperature');
percent = cdflib.getVarReservePercent(cdfid,varnum);
cdflib.close(cdfid);
```

More About

reserve percentage

Specifies how much extra space to allocate for a compressed variable. This extra space allows the variable to expand when you write additional records to the variable. If you do not specify this room for growth, the library has to move the variable to the end of the file when the size expands and the space at the original location of the variable becomes wasted space.

By default, the reserve percent is `0` (no extra space is reserved). You can specify any percentage between `1` and `100` and values greater than `100`. The value specifies the percentage of the uncompressed size of the variable.

References

This function corresponds to the CDF library C API routine `CDFgetzVarReservePercent`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarReservePercent`

cdflib.getVarSparseRecords

Information about how variable handles sparse records

Syntax

```
stype = cdflib.getVarSparseRecords(cdfId,varNum)
```

Description

`stype = cdflib.getVarSparseRecords(cdfId,varNum)` returns information about how a variable in the Common Data Format (CDF) file handles sparse records.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable. Variable numbers are zero-based.

Output Arguments

`stype`

One of the following text strings, or its numeric equivalent, that specifies how the variable handles sparse records.

Text String	Description
`'NO_SPARSERECORDS'`	No sparse records.
`'PAD_SPARSERECORDS'`	For sparse records, the library uses the variable's pad value when reading values from a missing record.

Text String	Description
'PREV_SPARSERECORDS'	For sparse records, the library uses values from the previous existing record when reading values from a missing record. If there is no previous existing record, the library uses the variable's pad value.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Open the example CDF, and then get the sparse record type of a variable in the file:

```
cdfid = cdflib.open('example.cdf');

stype = cdflib.getVarSparseRecords(cdfid,0)

stype =

NO_SPARSERECORDS

%Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarSparseRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarSparseRecords`

cdflib.getVersion

Common Data Format (CDF) library version and release information

Syntax

```
[version,release,increment] = cdflib.getVersion(cdfId)
```

Description

`[version,release,increment] = cdflib.getVersion(cdfId)` returns information about the version of the Common Data Format (CDF) library used to create a CDF file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`version`

Numeric value indicating the version number of the CDF library.

`release`

Numeric value indicating the release number of the CDF library.

`increment`

Numeric value indicating the increment number of the CDF library.

Examples

Open the example CDF file, and then find out the version of the CDF library used to create it:

```
cdfId = cdflib.open('example.cdf');

[version, release, increment] = cdflib.getVersion(cdfId)

version =

     2

release =

     7

increment =

     8

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetVersion`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getLibraryVersion`

cdflib.hyperGetVarData

Read hyperslab of data from variable

Syntax

```
data = cdflib.hyperGetVarData(cdfId,varNum,recSpec,dimSpec)
data = cdflib.hyperGetVarData(cdfId,varNum,recSpec)
```

Description

`data = cdflib.hyperGetVarData(cdfId,varNum,recSpec,dimSpec)` reads a hyperslab of data from a variable in the Common Data Format (CDF) file. Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library.

`data = cdflib.hyperGetVarData(cdfId,varNum,recSpec)` reads a hyperslab of data for a zero-dimensional variable in the Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Number identifying the variable containing the datum.

`recSpec`

Three-element array, `[RSTART RCOUNT RSTRIDE]`, where `RSTART`, `RCOUNT`, and `RSTRIDE` are scalar values specifying the starting record, number of records to read, and the sampling interval or stride between records. Record numbers are zero-based.

`dimSpec`

Three-element cell array, `{DSTART DCOUNT DSTRIDE}`, where `DSTART`, `DCOUNT`, and `DSTRIDE` are *n*-element vectors that describe the start, number of values along each

dimension, and sampling interval along each dimension. If the hyperslab has zero dimensions, you can omit this parameter. Dimension indices are zero-based.

Examples

Open the example CDF file, and then get all the data associated with a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records allocated for the first variable in the file.
maxRecNum = cdflib.getVarMaxWrittenRecNum(cdfid,0);

% Retrieve all data in records for variable.
data = cdflib.hyperGetVarData(cdfid,0,[0 maxRecNum 1]);

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFhyperGetzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.hyperPutVarData`

cdflib.hyperPutVarData

Write hyperslab of data to variable

Syntax

```
cdflib.hyperPutVarData(cdfId,varNum,recSpec,dimSpec,data)
```

Description

`cdflib.hyperPutVarData(cdfId,varNum,recSpec,dimSpec,data)` writes a hyperslab of data to a variable in a Common Data Format (CDF) file. Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Specifies the variable containing the datum.

`recSpec`

Three-element array described by `[RSTART RCOUNT RSTRIDE]`, where `RSTART`, `RCOUNT`, and `RSTRIDE` are scalar values giving the start, number of records, and sampling interval (or stride) between records. Record indices are zero-based.

`dimSpec`

Three-element cell array described by `{DSTART DCOUNT DSTRIDE}`, where `DSTART`, `DCOUNT`, and `DSTRIDE` are n-element vectors that describe the start, number of values along each dimension, and sampling interval along each dimension. If the hyperslab has zero dimensions, you can omit this parameter. Dimension indices are zero-based.

`data`

Data to write to the variable.

Examples

Create a CDF, create a variable, and then write a slab of data to the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write data to the variable
cdflib.hyperPutVarData(cdfid,varNum,0,[],int8(98))


%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFhyperzPutVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.hyperGetVarData`

cdflib.inquire

Basic characteristics of Common Data Format (CDF) file

Syntax

```
info = cdflib.inquire(cdfId)
```

Description

`info = cdflib.inquire(cdfId)` returns basic information about a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`info`

A structure containing the following fields:

Field	Description
`encoding`	Encoding of the variable data and attribute entry data
`majority`	Majority of the variable data
`maxRec`	Maximum record number written to a CDF variable
`numVars`	Number of CDF variables
`numvAttrs`	Number of attributes with variable scope
`numgAttrs`	Number of attributes with global scope

Examples

Open the example CDF file, and then get basic information about the file:

```
cdfId = cdflib.open('example.cdf');

info = cdflib.inquire(cdfId)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: 23
      numVars: 5
    numvAttrs: 1
    numgAttrs: 3
```

References

This function corresponds to the CDF library C API routines `CDFinquireCDF` and `CDFgetNumgAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireVar`

cdflib.inquireAttr

Information about attribute

Syntax

```
info = cdflib.inquireAttr(cdfId,attrNum)
```

Description

`info = cdflib.inquireAttr(cdfId,attrNum)` returns information about an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute in the file. Attribute numbers are zero-based.

Output Arguments

`info`

Structure containing the following fields.

Field	Description
`name`	Attribute's name
`scope`	Either `'GLOBAL_SCOPE'` or `'VARIABLE_SCOPE'`
`maxgEntry`	The maximum entry number used for global attributes.

Field	Description
`maxEntry`	The maximum entry number used for attributes with variable scope.

Examples

Open the example CDF, and then get information about the first attribute in the file.

```
cdfid = cdflib.open('example.cdf');

% Get information about an attribute
info = cdflib.inquireAttr(cdfid,0)

info =

         name: 'SampleAttribute'
        scope: 'GLOBAL_SCOPE'
    maxgEntry: 4
     maxEntry: -1

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireAttrgEntry` | `cdflib.inquireAttrEntry`

cdflib.inquireAttrEntry

Information about entry in attribute with variable scope

Syntax

```
[datatype,numElements] =
cdflib.inquireAttrEntry(cdfId,attrNum,entryNum)
```

Description

`[datatype,numElements] = cdflib.inquireAttrEntry(cdfId,attrNum,entryNum)` returns the data type and the number of elements for an attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have variable scope.

`entryNum`

Numeric value identifying the entry in the attribute. Entry number are zero-based.

Output Arguments

`datatype`

Text string identifying a CDF data type. For a list of CDF data types, see `cdflib.putAttrEntry`

`numElements`

Numeric value indicating the number of elements in the entry.

Examples

Open example CDF, and then get information about entries associated with an attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrEntry(cdfid,3,0)

dtype =

cdf_char


numel =

    10


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireAttr` | `cdflib.getAttrScope`

cdflib.inquireAttrgEntry

Information about entry in attribute with global scope

Syntax

```
[datatype,numElements] =
cdflib.inquireAttrgEntry(cdfId,attrNum,entryNum)
```

Description

`[datatype,numElements] = cdflib.inquireAttrgEntry(cdfId,attrNum,entryNum)` returns the data type and the number of elements for a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Numeric value identifying the entry in the attribute. Entry number are zero-based.

Output Arguments

`datatype`

Text string identifying a CDF data type. For a list of CDF data types, see `cdflib.putAttrgEntry`

numElements

Numeric value indicating the number of elements in the entry.

Examples

Open the example CDF, and then get information about entries associated with a global attribute in the file.

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attributes have global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrgEntry(cdfid,0,0)

dtype =

cdf_char


numel =

    23


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

See Also

`cdflib.inquireAttr` | `cdflib.inquireAttrEntry`

cdflib.inquireVar

Information about variable

Syntax

```
info = cdflib.inquireVar(cdfId,varNum)
```

Description

`info = cdflib.inquireVar(cdfId,varNum)` returns information about a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable. Variable numbers are zero-based.

Output Arguments

`info`

Structure containing the following fields.

Field	Description
`name`	Name of the variable
`datatype`	Data type
`numElements`	Number of elements of the datatype
`dims`	Sizes of the dimensions

Field	Description
recVariance	Record variance
dimVariance	Dimension variances

Record and dimension variances affect how the library physically stores variable data. For example, if a variable has a record variance of VARY, the library physically stores each record. If the record variance is NOVARY, the library only stores one record.

Examples

Open the example CDF file and get information about a variable.

```
cdfid = cdflib.open('example.cdf');

% Determine if the file contains variables
info  = cdflib.inquireVar(cdfid,1)

info =

           name: 'Longitude'
       datatype: 'cdf_int1'
    numElements: 1
           dims: [2 2]
    recVariance: 0
    dimVariance: [1 0]
```

References

This function corresponds to the CDF library C API routine CDFinquirezVar.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the cdfcopyright.txt file.

See Also

cdflib.inquire

cdflib.open

Open existing Common Data Format (CDF) file

Syntax

```
cdfId = cdflib.open(filename)
```

Description

`cdfId = cdflib.open(filename)` opens an existing Common Data Format (CDF) file. *`filename`* is a text string that identifies the file.

This function returns a CDF file identifier, `cdfId`.

All CDF files opened this way have the `zMode` set to `zModeon2`. Refer to the *CDF User's Guide* for information about `zModes`.

Examples

Open the example CDF file:

```
cdfId = cdflib.open('example.cdf');

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFopenCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.close` | `cdflib.create`

cdflib.putAttrEntry

Write value to entry in attribute with variable scope

Syntax

```
cdflib.putAttrEntry(cdfId,attrNum,entryNum,CDFDataType,entryVal)
```

Description

`cdflib.putAttrEntry(cdfId,attrNum,entryNum,CDFDataType,entryVal)` writes a value to an attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Number identifying attribute. The attribute must have variable scope. Attribute numbers are zero-based.

`entryNum`

Number identifying entry. Entry numbers are zero-based.

`CDFdatatype`

One of the following text strings, or its numeric equivalent, that specify the data type of the attribute entry.

CDF Data Type	MATLAB Equivalent
`CDF_BYTE`	1-byte, signed integer
`CDF_CHAR`	1 byte, signed character data type that maps to the MATLAB `char` class

CDF Data Type	MATLAB Equivalent
CDF_INT1	1-byte, signed integer.
CDF_UCHAR	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
CDF_UINT1	1-byte, unsigned integer
CDF_INT2	2-byte, signed integer
CDF_UINT2	2-byte, unsigned integer.
CDF_INT4	4-byte, signed integer
CDF_UINT4	4-byte, unsigned integer
CDF_FLOAT	4-byte, floating point
CDF_REAL4	4-byte, floating point
CDF_REAL8	8-byte, floating point.
CDF_DOUBLE	8-byte, floating point
CDF_EPOCH	8-byte, floating point
CDF_EPOCH16	two 8-byte, floating point

`entryVal`

Data to be written to attribute entry.

Examples

Create a CDF and create an attribute with variable scope in the file. Write a value to an entry in the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
```

```
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create an attribute of variable scope in the file.
attrNum = cdflib.createAttr(cdfid,'Another Attribute','variable_scope');

% Write a value to an entry for the attribute
cdflib.putAttrEntry(cdfid,attrNum,0,'CDF_CHAR','My Variable Attribute Test');

% Get the value of the global attribute entry
value = cdflib.getAttrEntry(cdfid,attrNum,0)

value =

My Variable Attribute Test


% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrEntry` | `cdflib.putAttrgEntry` | `cdflib.getAttrgEntry` | `cdflib.getConstantValue`

cdflib.putAttrgEntry

Write value to entry in attribute with global scope

Syntax

```
cdflib.putAttrgEntry(cdfId,attrNum,entryNum,cdfDataType,entryVal)
```

Description

`cdflib.putAttrgEntry(cdfId,attrNum,entryNum,cdfDataType,entryVal)` writes a value to a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Number identifying attribute. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Number identifying entry. Entry numbers are zero-based.

`CDFdatatype`

One of the following text strings that specify the data type of the attribute entry, or its numeric equivalent.

CDF Data Type	MATLAB Equivalent
`CDF_BYTE`	1-byte, signed integer
`CDF_CHAR`	1 byte, signed character data type that maps to the MATLAB `char` class

CDF Data Type	MATLAB Equivalent
CDF_INT1	1-byte, signed integer.
CDF_UCHAR	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
CDF_UINT1	1-byte, unsigned integer
CDF_INT2	2-byte, signed integer
CDF_UINT2	2-byte, unsigned integer.
CDF_INT4	4-byte, signed integer
CDF_UINT4	4-byte, unsigned integer
CDF_FLOAT	4-byte, floating point
CDF_REAL4	4-byte, floating point
CDF_REAL8	8-byte, floating point.
CDF_DOUBLE	8-byte, floating point
CDF_EPOCH	8-byte, floating point
CDF_EPOCH16	two 8-byte, floating point

`entryVal`

Data to be written to global attribute entry.

Examples

Create a CDF and create a global attribute in the file. Write a value to an entry in the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
```

```
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a global attribute in the file.
attrNum = cdflib.createAttr(cdfid,'Purpose','global_scope');

% Write a value to an entry for the global attribute
cdflib.putAttrgEntry(cdfid,attrNum,0,'CDF_CHAR','My Test');

% Get the value of the global attribute entry
value = cdflib.getAttrgEntry(cdfid,attrNum,0)

value =

My Test

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrgEntry` | `cdflib.putAttrEntry` | `cdflib.getAttrEntry` | `cdflib.getConstantValue`

cdflib.putVarData

Write single value to variable

Syntax

```
cdflib.putVarData(cdfId,varNum,recNum,indices,datum)
```

Description

`cdflib.putVarData(cdfId,varNum,recNum,indices,datum)` writes a single value to a variable in a Common Data File (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable to which you want to write the datum. Variable numbers are zero-based.

`recNum`

Numeric value that identifies the record to which you want to write the datum. Record numbers are zero-based.

`dims`

Dimension indices within the record. Dimension indices are zero-based.

`datum`

Data to be written to the variable.

Examples

Create a CDF, create a variable in the CDF and write data to the variable. To run this example, you must have write permission in the current folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write some data to the variable
cdflib.putVarData(cdfid,varNum,0,[],int8(98))

% Read the value from the variable.
datum = cdflib.getVarData(cdfid,varNum,0)

datum =

   98

%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarData` | `cdflib.getVarRecordData` | `cdflib.hyperGetVarData`

cdflib.putVarRecordData

Write entire record to variable

Syntax

```
cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)
```

Description

`cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)` writes data to a record in a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable to which you want to write the datum. Variable numbers are zero-based.

`recNum`

Numeric value identifying the location of the datum in the variable. Record numbers are zero-based.

`recordData`

Data to be written to the variable.

Examples

Create a CDF, create a variable, and write an entire record of data to the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write some data to the variable
cdflib.putVarRecordData(cdfid,varNum,0,int8(98))

% Read the value from the variable.
datum = cdflib.getVarData(cdfid,varNum,0)

datum =

   98

%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputzVarRecordData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarRecordData` | `cdflib.putVarData` | `cdflib.hyperPutVarData`

cdflib.renameAttr

Rename existing attribute

Syntax

```
cdflib.renameAttr(cdfId,attrNum,newName)
```

Description

`cdflib.renameAttr(cdfId,attrNum,newName)` renames an attribute in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `attrNum` is a numeric value that identifies the attribute. Attribute numbers are zero-based. `newName` is a text string that specifies the name you want to assign to the attribute.

Examples

Create a CDF, create an attribute in the CDF, and then rename the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create an attribute
attrNum = cdflib.createAttr(cdfid,'Purpose','global_scope');

% Rename the attribute
cdflib.renameAttr(cdfid, attrNum, 'NewPurpose');

% Check the name of the attribute
attrName = cdflib.getAttrName(cdfid,anum)

attrName =

NewPurpose

% Clean up
```

```
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFrenameAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr`

cdflib.renameVar

Rename existing variable

Syntax

```
cdflib.renameVar(cdfId,varNum,newName)
```

Description

`cdflib.renameVar(cdfId,varNum,newName)` renames a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based. `newName` is a text string that specifies the name you want to assign to the variable.

Examples

Create a CDF, create a variable in the CDF, and then rename the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the name of the variable.
name = cdflib.getVarName(cdfid,varNum)

name =

Time

% Rename the variable
cdflib.renameVar(cdfid,varNum,'NewName');

% Check the new name.
```

```
name = cdflib.getVarName(cdfid,varNum)

name =

NewName

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFrenamezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

cdflib.setCacheSize

Specify number of dotCDF cache buffers

Syntax

```
cdflib.setCacheSize(cdfId,numBuffers)
```

Description

`cdflib.setCacheSize(cdfId,numBuffers)` specifies the number of cache buffers the CDF library uses for an open dotCDF file. A *dotCDF* file is a file with the `.cdf` file extension.

`cdfId` identifies an open CDF file. `numBuffers` is a numeric value that specifies the number of buffers.

For information about cache schemes, see the *CDF User's Guide*.

Examples

Create a CDF file and set the cache size. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('your_file.cdf');

% Get the default cache size
numBuf = cdflib.getCacheSize(cdfid)

numBuf =

   300

% Specify a cache size
cdflib.setCacheSize(cdfid,150)

% Check the cache size again
numBuf =  cdflib.getCacheSize(cdfid)
```

```
numBuf =

   150

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCacheSize`

cdflib.setChecksum

Specify checksum mode

Syntax

```
cdflib.setChecksum(cdfId,mode)
```

Description

`cdflib.setChecksum(cdfId,mode)` specifies the checksum mode of a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`mode`

Either of the following text strings, or its numeric equivalent. To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

`'MD5_CHECKSUM'`	Sets file checksum to MD5 checksum.
`'NO_CHECKSUM'`	File does not use a checksum.

Examples

Create a CDF file and set the checksum mode. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('mycdf.cdf');

% Check initial value of checksum.
```

```
mode = cdflib.getChecksum(cdfid)

NO_CHECKSUM

cdflib.setChecksum(cdfid,'MD5_CHECKSUM')

% Verify the setting
mode = cdflib.getChecksum(cdfid)

MD5_CHECKSUM
```

References

This function corresponds to the CDF library C API routine `CDFsetChecksum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getChecksum` | `cdflib.getConstantValue`

cdflib.setCompression

Specify compression settings

Syntax

```
cdflib.setCompression(cdfId,ctype,cparms)
```

Description

`cdflib.setCompression(cdfId,ctype,cparms)` specifies compression settings of a Common Data Format (CDF) file.

This function sets the compression for the CDF file itself, not that of any variables in the file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`ctype`

One of the following text strings, or its numeric equivalent, specifying compression type.

Text String	Compression Type
`'NO_COMPRESSION'`	No compression
`'RLE_COMPRESSION'`	Run-length encoding compression
`'HUFF_COMPRESSION'`	Huffman compression
`'AHUFF_COMPRESSION'`	Adaptive Huffman compression
`'GZIP_COMPRESSION'`	GNU's zip compression

To get the numeric equivalent, use `cdflib.getConstantValue`.

`cparms`

Optional parameter specifying any additional parameters required by the compression type. Currently, the only compression type that uses this parameter is `'GZIP_COMPRESSION'`. For this compression type, use `cparms` to specify the level of compression as a numeric value between 1 and 9.

Examples

Create a CDF file and set the compression setting of the file. To run this example, your current folder must be writable.

```
cdfId = cdflib.create('your_file.cdf');

% Determine the file's default compression setting
[ctype, cparms, cpercent ] = cdflib.getCompression(cdfId)

ctype =

NO_COMPRESSION

cparms =

     []

cpercent =

   100

% Specify new compression setting
cdflib.setCompression(cdfId,'HUFF_COMPRESSION');

% Check the file's compression setting.
[ctype, cparms, cpercent ] = cdflib.getCompression(cdfId)

ctype =

HUFF_COMPRESSION

cparms =

OPTIMAL_ENCODING_TREES
```

```
cpercent =

     0

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCompression` | `cdflib.getConstantValue`

cdflib.setCompressionCacheSize

Specify number of compression cache buffers

Syntax

```
cdflib.setCompressionCacheSize(cdfId,numBuffers)
```

Description

`cdflib.setCompressionCacheSize(cdfId,numBuffers)` specifies the number of cache buffers used for the compression scratch CDF file. For more information about CDF cache schemes, see the *CDF User's Guide*.

`cdfId` identifies the CDF file. `numBuffers` specifies the number of buffers.

Examples

Create a CDF file and specify the number of compression cache buffers used. To run this example you must be in a writable folder.

```
cdfId = cdflib.create('your_file.cdf');

% Get the current number of compression cache buffers
numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =

    80

% Set a new value
cdflib.setCompressionCacheSize(cdfId,100)

% Check the new value
numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =
```

```
   100

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCompressionCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCompressionCacheSize`

cdflib.setFileBackward

Set backward compatibility mode

Syntax

```
cdflib.setFileBackward(mode)
```

Description

`cdflib.setFileBackward(mode)` sets the backward compatibility mode to the value specified by `mode`.

Input Arguments

`mode`

One of the following text strings:

`BACKWARDFILEon`	Set backward compatibility mode on.
`BACKWARDFILEoff`	Set backward compatibility mode off.

Default: `BACKWARDFILEoff`

Examples

Set backward compatibility mode and then check the value.

```
cdflib.setFileBackward('BACKWARDFILEon');

mode = cdflib.getFileBackward

mode =

BACKWARDFILEon
```

More About

backward compatibility mode

When specified, ensures that any new CDF file created using CDF V3.0 (or later) will be readable by clients using version 2.7 of the CDF library. CDF 3.0 and later releases use a 64-bit file offset to allow for files greater than 2G bytes in size. CDF library versions released before CDF 3.0 use a 32-bit file offset.

Tips

- Setting backward compatibility mode affects only your current MATLAB session, or until you call `cdflib.setFileBackward` again.

References

This function corresponds to the CDF library C API routine `CDFsetFileBackward`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getFileBackward` | `cdflib.getConstantValue`

cdflib.setFormat

Specify format of Common Data Format (CDF) file

Syntax

```
cdflib.setFormat(cdfId,format)
```

Description

`cdflib.setFormat(cdfId,format)` specifies the format of a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`format`

Either of the following text strings, or its numeric equivalent.

`'SINGLE_FILE'`	The CDF consists of only one file. This is the default file format
`'MULTI_FILE'`	The CDF consists of one header file for control and attribute data and one additional file for each variable in the CDF.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Create a CDF file and specify its format. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('mycdffile.cdf');
```

```
% Specify multifile format.
cdflib.setFormat(cdfId, 'MULTI_FILE');

% Check format.
 format = cdflib.getFormat(cdfId)

format =

MULTI_FILE

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetFormat`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getFormat` | `cdflib.getConstantValue`

cdflib.setMajority

Specify majority of variables

Syntax

```
cdflib.setMajority(cdfId,majority)
```

Description

`cdflib.setMajority(cdfId,majority)` specifies the majority of variables in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`majority`

Either of the following text strings, or its numeric equivalent.

`'ROW_MAJOR'`	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest. This is the default.
`'COLUMN_MAJOR'`	Fortran-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Create a CDF file and specify the majority used by variables in the file. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('your_file.cdf')

% Specify the majority used by variables in the file
cdflib.setMajority(cdfId,'COLUMN_MAJOR');

% Check the majority value
majority = cdflib.getMajority(cdfId)

majority =

COLUMN_MAJOR

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetMajority`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getMajority`

cdflib.setReadOnlyMode

Specify read-only mode

Syntax

```
cdflib.setReadOnlyMode(cdfId,mode)
```

Description

`cdflib.setReadOnlyMode(cdfId,mode)` specifies the read-only mode of a Common Data Format (CDF) file.

After you open a CDF file, you can put the file into read-only mode to prevent accidental modification.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`mode`

Either of the following text strings or its numeric equivalent.

`'READONLYon'`	CDF file is read-only
`'READONLYoff'`	CDF file is modifiable.

To get the numeric equivalent of these mode values, use `cdflib.getConstantValue`.

Examples

Open the example CDF file and set the file to read-only mode.

```
cdfId = cdflib.open('example.cdf');

% Set the  file to READONLY mode
cdflib.setReadOnlyMode(cdfId,'READONLYon')

% Check read-only status of file again.
mode = cdflib.getReadOnlyMode(cdfId)

mode =

READONLYon

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetReadOnlyMode`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getReadOnlyMode` | `cdflib.getConstantValue`

cdflib.setStageCacheSize

Specify number of staging cache buffers for Common Data Format (CDF) file

Syntax

```
cdflib.setStageCacheSize(cdfId,numBuffers)
```

Description

`cdflib.setStageCacheSize(cdfId,numBuffers)` specifies the number of staging cache buffers for a Common Data Format (CDF) file. For information about CDF cache schemes, see the *CDF User's Guide*.

`cdfId` identifies the CDF file. `numBuffers` is a numeric value that specifies the number of buffers.

Examples

Open the example CDF file and specify the number of cache buffers used.

```
cdfId = cdflib.open('example.cdf');

% Get current number of staging cache buffers
size = cdflib.getStageCacheSize(cdfId)

size =

   125

% Specify new cache size value.
cdflib.setStageCacheSize(cdfId, 200)

% Get size again.
size = cdflib.getStageCacheSize(cdfId)

size =
```

```
   200

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetStageCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getStageCacheSize`

cdflib.setValidate

Specify library validation mode

Syntax

```
cdflib.setValidate(mode)
```

Description

`cdflib.setValidate(mode)` specifies the validation mode of the Common Data Format (CDF) library. Specify the validation mode before opening any files.

Input Arguments

`mode`

Either of the following text strings, or its numeric equivalent:

`'VALIDATEFILEon'`	Turns validation mode on. With validation mode on, the library performs sanity checks on the data fields in the CDF' file's internal data structures to make sure that the values are within valid ranges and consistent with the defined values/types/entries. This mode also ensures that variable and attribute associations within the file are valid. Note, however, that enabling this mode will, in most cases, slow down the file opening process, especially for large or very fragmented files.
`'VALIDATEFILEoff`	Turns validation mode off.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Set the validation mode of the CDF library.

```
cdflib.setValidate('VALIDATEFILEon');
```

References

This function corresponds to the CDF library C API routine `CDFsetValidate`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getValidate` | `cdflib.getConstantValue`

cdflib.setVarAllocBlockRecords

Specify range of records to be allocated for variable

Syntax

```
cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)
```

Description

`cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)` specifies a range of records you want to allocate (but not write) for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`firstRec`

Numeric value identifying the record at which to start allocating. Record numbers are zero-based.

`lastRec`

Numeric value identifying the record at which to stop allocating. Record numbers are zero-based.

Examples

Create a CDF, create a variable in the CDF, and then specify the number of records to allocate for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Specify the number of records to allocate.
cdflib.setVarAllocBlockRecords(cdfid,varNum,1,10);


%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarAllocBlockRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarAllocRecords`

cdflib.setVarBlockingFactor

Specify blocking factor for variable

Syntax

```
cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)
```

Description

`cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)` specifies the blocking factor for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable numbers are zero-based.

`blockingFactor`

Numeric value that specifies the number of records to allocate when writing to an unallocated record.

Examples

Create a CDF, create a variable in the CDF, and then set the blocking factor used with the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');
```

```
% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the current blocking factor used with the variable
bFactor = cdflib.getVarBlockingFactor(cdfid,varNum)

bFactor =

    0

% Change the blocking factor for the variable
cdflib.setVarBlockingFactor(cdfid,varNum,10);

% Check the new blocking factor .
bFactor = cdflib.getVarBlockingFactor(cdfid,varNum)

bFactor =

    10

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

More About

blocking factor

A variable's *blocking factor* specifies the minimum number of records the library allocates when you write to an unallocated record. If you specify a fractional blocking factor, the library rounds the value down.

References

This function corresponds to the CDF library C API routine `CDFsetzVarBlockingFactor`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

See Also

`cdflib.getVarBlockingFactor`

cdflib.setVarCacheSize

Specify number of multi-file cache buffers for variable

Syntax

```
cdflib.setVarCacheSize(cdfId,varNum,numBuffers)
```

Description

`cdflib.setVarCacheSize(cdfId,varNum,numBuffers)` specifies the number of cache buffers the CDF library uses for a variable in a Common Data Format (CDF) file.

This function is only used with multifile format CDF files. It does not apply to single-file format CDFs. For more information about caching, see the *CDF User's Guide*.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`numBuffers`

Numeric value identifying the number of cache buffers to use.

Examples

Create a multifile CDF, and then retrieve the number of buffers being used for a variable:

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*

your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

% Increase the number of cache buffers used.
cdflib.setVarCacheSize(cdfid,varNum,5)

% Check the number of cache buffers used with the variable.
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     5

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCacheSize` | `cdflib.setVarsCacheSize`

cdflib.setVarCompression

Specify compression settings used with variable

Syntax

```
cdflib.setVarCompression(cdfId,varNum,ctype,cparams)
```

Description

`cdflib.setVarCompression(cdfId,varNum,ctype,cparams)` configures the compression setting for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`ctype`

One of the following text strings, or its numeric equivalent, specifying the compression type.

Text String	Compression Type
`'NO_COMPRESSION'`	No compression.
`'RLE_COMPRESSION'`	Run-length encoding compression
`'HUFF_COMPRESSION'`	Huffman compression
`'AHUFF_COMPRESSION'`	Adaptive Huffman compression

Text String	Compression Type
'GZIP_COMPRESSION'	GNU's zip compression

cparams

Optional parameter specifying any additional parameters required by the compression type. Currently, the only compression type that uses this parameter is 'GZIP_COMPRESSION'. For this compression type, you use cparms to specify the level of compression as a numeric value between 1 and 9.

Examples

Create a CDF, create a variable, and then set the compression used by the variable. To run this example, you must be in a folder with execute permission.

```
cdfid = cdflib.create('mycdf.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Specify the compression used by the variable.
cdflib.setVarCompression(cdfid,0,'GZIP_COMPRESSION',8)

% Check the compression setting of the variable
[ctype params percent] = cdflib.getVarCompression(cdfid,0)

ctype =

GZIP_COMPRESSION

params =

     8

percent =

     0

% Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompression` | `cdflib.getVarCompression`

cdflib.setVarInitialRecs

Specify initial number of records written to variable

Syntax

```
cdflib.setVarInitialRecs(cdfId,varNum,numrecs)
```

Description

`cdflib.setVarInitialRecs(cdfId,varNum,numrecs)` specifies the initial number of records to write to a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable numbers are zero-based.

`numRecs`

Numeric value specifying the number of records to write.

Examples

Create a CDF, create a variable, and then specify the number of records to write for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);
```

```
% Specify the number of records to write for the variable
cdflib.setVarInitialRecs(cdfid,varNum,100);

recsWritten = cdflib.getVarNumRecsWritten(cdfid,varNum)

recsWritten =

   100

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarInitialRecs`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

cdflib.setVarPadValue

Specify pad value used with variable

Syntax

```
cdflib.setVarPadValue(cdfId,varNum,padvalue)
```

Description

`cdflib.setVarPadValue(cdfId,varNum,padvalue)` specifies the pad value used with a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable numbers are zero-based.

`padValue`

Value to use a pad value for the variable. The data type of the pad value must match the data type of the variable.

Examples

Create a CDF, create a variable in the CDF, and then set the pad value used with the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
```

```
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the current pad value used with the variable
padval = cdflib.getVarPadValue(cdfid,varNum)

padval =

    0

% Change the pad value for the variable
cdflib.setVarPadValue(cdfid,varNum,int8(1));

% Check the new pad value.
padval = cdflib.getVarPadValue(cdfid,varNum)

padval =

    1

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarPadValue`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

See Also

`cdflib.getVarPadValue`

cdflib.SetVarReservePercent

Specify reserve percentage for variable

Syntax

```
cdflib.setVarReservePercent(cdfId,varNum,percent)
```

Description

`cdflib.setVarReservePercent(cdfId,varNum,percent)` specifies the compression reserve percentage for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`percent`

Numeric value specifying the amount of extra space to allocate for a compressed variable, experssed as a percentage. You can specify values between`0` (no extra space is reserved) and `100`, or values greater than `100`. The value specifies the percentage of the uncompressed size of the variable. If you specify a fractional reserve percentages, the library rounds the value down.

Examples

Create a CDF, create a variable, set the compression of the variable, and then set the reserve percent for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Set the compression of the variable.
cdflib.setVarCompression(cdfid,varNum,'GZIP_COMPRESSION',8);

% Set the compression reserver percentage
cdflib.setVarReservePercent(cdfid,varNum, 80);

cdflib.close(cdfid);
```

More About

reserve percentage

Specifies how much extra space to allocate for a compressed variable. This extra space allows the variable to expand when you write additional records to the variable. If you do not specify this room for growth, the library has to move the variable to the end of the file when the size expands and the space at the original location of the variable becomes wasted space.

References

This function corresponds to the CDF library C API routine `CDFsetzVarReservePercent`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarReservePercent` | `cdflib.setVarCompression` | `cdflib.getVarCompression`

cdflib.setVarsCacheSize

Specify number of cache buffers used for all variables

Syntax

```
cdflib.setVarsCacheSize(cdfId,varNum,numBuffers)
```

Description

`cdflib.setVarsCacheSize(cdfId,varNum,numBuffers)` specifies the number of cache buffers the CDF library uses for all the variables in the multifile format Common Data Format (CDF) file.

This function is not applicable to single-file CDFs. For more information about caching, see the *CDF User's Guide*.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`numBuffers`

Numeric value specifying the cache buffers.

Examples

Create a multifile CDF and specify the number of buffers used for all variables. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*

your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

% Specify the number of cache buffers used by all variables in CDF.
cdflib.setVarsCacheSize(cdfid,6)

% Check the number of cache buffers used with the variable.
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     6

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarsCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCacheSize` | `cdflib.setVarCacheSize`

cdflib.setVarSparseRecords

Specify how variable handles sparse records

Syntax

```
cdflib.getVarSparseRecords(cdfId,varNum,stype)
```

Description

`cdflib.getVarSparseRecords(cdfId,varNum,stype)` specifies the sparse records type of a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Number that identifies the variable to be set. Variable numbers are zero-based.

`stype`

One of the following text strings, or its numeric equivalent, that specifies how the variable handles sparse records.

Text String	Description
`'NO_SPARSERECORDS'`	No sparse records
`'PAD_SPARSERECORDS'`	For sparse records, the library uses the variable's pad value when reading values from a missing record.
`'PREV_SPARSERECORDS'`	For sparse records, the library uses values from the previous existing record when reading values from a missing record. If there is no previous existing record, the library uses the variable's pad value.

To get the numeric equivalent of these text string constants, use the `cdflib.getConstantValue` function.

Examples

Open a multifile CDF and close a variable.

Create a CDF, create a variable, and set the sparse records type of the variable. To run this example you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Set the sparse records type of the variable
cdflib.setVarSparseRecords(cdfid,varNum,'PAD_SPARSERECORDS');

% Check the sparse records type of the variable
stype = cdflib.getVarSparseRecords(cdfid,varNum)

stype =

PAD_SPARSERECORDS


%Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarSparseRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarSparseRecords` | `cdflib.getConstantValue`

cdfread

Read data from Common Data Format (CDF) file

Syntax

```
data = cdfread(filename)
data = cdfread(filename, param1, val1, param2, val2, ...)
[data, info] = cdfread(filename, ...)
```

Description

`data = cdfread(filename)` reads all the data from the Common Data Format (CDF) file specified in the string `filename`. CDF data sets typically contain a set of variables, of a specific data type, each with an associated set of records. The variable might represent time values with each record representing a specific time that an observation was recorded. `cdfread` returns all the data in a cell array where each column represents a variable and each row represents a record associated with a variable. If the variables have varying numbers of associated records, `cdfread` pads the rows to create a rectangular cell array, using pad values defined in the CDF file.

Note Because `cdfread` creates temporary files, the current working directory must be writeable.

`data = cdfread(filename, param1, val1, param2, val2, ...)` reads data from the file, where *`param1`*, *`param2`*, and so on, can be any of the parameters listed in the following table.

`[data, info] = cdfread(filename, ...)` returns details about the CDF file in the `info` structure.

Parameter	Value
`'Records'`	A vector specifying which records to read. Record numbers are zero-based. `cdfread` returns a cell array with the same number of rows as the number of records read and as many columns as there are variables.

Parameter	Value
`'Variables'`	A 1-by-*n* or *n*-by-1 cell array specifying the names of the variables to read from the file. *n* must be less than or equal to the total number of variables in the file. `cdfread` returns a cell array with the same number of columns as the number of variables read, and a row for each record read.
`'Slices'`	An *m*-by-3 array, where each row specifies where to start reading along a particular dimension of a variable, the skip interval to use on that dimension (every item, every other item, etc.), and the total number of values to read on that dimension. *m* must be less than or equal to the number of dimensions of the variable. If *m* is less than the total number of dimensions, `cdfread` reads every value from the unspecified dimensions (`[0 1 n]`, where *n* is the total number of elements in the dimension. Note: Because the `'Slices'` parameter describes how to process a single variable, it must be used in conjunction with the `'Variables'` parameter.
`'ConvertEpochToDatenum'`	A Boolean value that determines whether `cdfread` automatically converts CDF epoch data types to MATLAB serial date numbers. If set to `false` (the default), `cdfread` wraps epoch values in MATLAB `cdfepoch` objects. Note: For better performance when reading large data sets, set this parameter to `true`.

Parameter	Value
'CombineRecords'	A Boolean value that determines how cdfread returns the CDF data sets read from the file. If set to false (the default), cdfread stores the data in an *m*-by-*n* cell array, where *m* is the number of records and *n* is the number of variables requested. If set to true, cdfread combines all records for a particular variable into one cell in the output cell array. In this cell, cdfread stores scalar data as a column array. cdfread extends the dimensionality of nonscalar and string data. For example, instead of creating 1000 elements containing 20-by-30 arrays for each record, cdfread stores all the records in one cell as a 1000-by-20-by-30 array Note: If you use the 'Records' parameter to specify which records to read, you cannot use the 'CombineRecords' parameter. Note: When using the 'Variable' parameter to read one variable, if the 'CombineRecords' parameter is true, cdfread returns the data as an M-by-N numeric or character array; it does not put the data into a cell array.

Note: To improve performance when working with large data files, use the 'ConvertEpochToDatenum' and 'CombineRecords' options.

Note: To improve performance, turn off the file validation which the CDF library does by default when opening files. For more information, see cdflib.setValidate.

Examples

Read all the data from a CDF file.

```
data = cdfread('example.cdf');
```

Read the data from the variable 'Time'.

```
data = cdfread('example.cdf', 'Variable', {'Time'});
```

Read the first value in the first dimension, the second value in the second dimension, the first and third values in the third dimension, and all values in the remaining dimension of the variable `'multidimensional'`.

```
data = cdfread('example.cdf', ...
               'Variable', {'multidimensional'}, ...
               'Slices', [0 1 1; 1 1 1; 0 2 2]);
```

This is similar to reading the whole variable into `data` and then using matrix indexing, as in the following.

```
data{1}(1, 2, [1 3], :)
```

Collapse the records from a data set and convert CDF epoch data types to MATLAB serial date numbers.

```
data = cdfread('example.cdf', ...
               'CombineRecords', true, ...
               'ConvertEpochToDatenum', true);
```

More About

- "Import CDF Files Using Low-Level Functions"

See Also

`cdfepoch` | `cdfinfo` | `cdflib.setValidate`

Introduced before R2006a

cdfwrite

Write data to Common Data Format (CDF) file

Note: `cdfwrite` is not recommended. Use the `cdflib` low-level functions instead.

Syntax

```
cdfwrite(filename,variablelist)
cdfwrite(...,'PadValues',padvals)
cdfwrite(...,'GlobalAttributes',gattrib)
cdfwrite(..., 'VariableAttributes', vattrib)
cdfwrite(...,'WriteMode',mode)
cdfwrite(...,'Format',format)
```

Description

`cdfwrite(filename,variablelist)` writes out a Common Data Format (CDF) file, specified in `filename`. The `filename` input is a string enclosed in single quotes. The `variablelist` argument is a cell array of ordered pairs, each of which comprises a CDF variable name (a string) and the corresponding CDF variable value. To write out multiple records for a variable, put the values in a cell array where each element in the cell array represents a record.

Note Because `cdfwrite` creates temporary files, both the destination directory for the file and the current working directory must be writeable.

`cdfwrite(...,'PadValues',padvals)` writes out pad values for given variable names. `padvals` is a cell array of ordered pairs, each of which comprises a variable name (a string) and a corresponding pad value. Pad values are the default values associated with the variable when an out-of-bounds record is accessed. Variable names that appear in `padvals` must appear in `variablelist`.

`cdfwrite(...,'GlobalAttributes',gattrib)` writes the structure `gattrib` as global metadata for the CDF file. Each field of the structure is the name of a global attribute. The value of each field contains the value of the attribute. To write out multiple values for an attribute, put the values in a cell array where each element in the cell array represents a record.

Note To specify a global attribute name that is invalid in your MATLAB application, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `GlobalAttributes` structure, and the corresponding name of the attribute to be written to the CDF file.

`cdfwrite(..., 'VariableAttributes', vattrib)` writes the structure `vattrib` as variable metadata for the CDF. Each field of the struct is the name of a variable attribute. The value of each field should be an M-by-2 cell array where M is the number of variables with attributes. The first element in the cell array should be the name of the variable and the second element should be the value of the attribute for that variable.

Note To specify a variable attribute name that is illegal in MATLAB, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `VariableAttributes` struct, and the corresponding name of the attribute to be written to the CDF file. If you are specifying a variable attribute of a CDF variable that you are renaming, the name of the variable in the `VariableAttributes` structure must be the same as the renamed variable.

`cdfwrite(...,'WriteMode',`*`mode`*`)`, where *`mode`* is either `'overwrite'` or `'append'`, indicates whether or not the specified variables should be appended to the CDF file if the file already exists. By default, `cdfwrite` overwrites existing variables and attributes.

`cdfwrite(...,'Format',`*`format`*`)`, where *`format`* is either `'multifile'` or `'singlefile'`, indicates whether or not the data is written out as a multifile CDF. In a multifile CDF, each variable is stored in a separate file with the name `*.vN`, where `N` is the number of the variable that is written out to the CDF. By default, `cdfwrite` writes out a single file CDF. When `'WriteMode'` is set to `'Append'`, the `'Format'` option is ignored, and the format of the preexisting CDF is used.

Examples

Write out a file `'example.cdf'` containing a variable `'Longitude'` with the value `[0:360]`.

```
cdfwrite('example', {'Longitude', 0:360});
```

Write out a file `'example.cdf'` containing variables `'Longitude'` and `'Latitude'` with the variable `'Latitude'` having a pad value of 10 for all out-of-bounds records that are accessed.

```
cdfwrite('example', {'Longitude', 0:360, 'Latitude', 10:20}, ...
         'PadValues', {'Latitude', 10});
```

Write out a file `'example.cdf'`, containing a variable `'Longitude'` with the value `[0:360]`, and with a variable attribute of `'validmin'` with the value 10.

```
varAttribStruct.validmin = {'Longitude' [10]};
cdfwrite('example', {'Longitude' 0:360}, 'VariableAttributes', ...
         varAttribStruct);
```

See Also

`cdfread` | `cdfinfo` | `cdfepoch`

Introduced before R2006a

ceil

Round toward positive infinity

Syntax

```
Y = ceil(X)
```

```
Y = ceil(t)
Y = ceil(t,unit)
```

Description

`Y = ceil(X)` rounds each element of `X` to the nearest integer greater than or equal to that element.

`Y = ceil(t)` rounds each element of the `duration` array `t` to the nearest number of seconds greater than or equal to that element.

`Y = ceil(t,unit)` rounds each element of `t` to the nearest number of the specified unit of time greater than or equal to that element.

Examples

Round Matrix Elements Toward Positive Infinity

```
X = [-1.9 -0.2 3.4; 5.6 7 2.4+3.6i];
Y = ceil(X)
```

```
Y =

  -1.0000 + 0.0000i   0.0000 + 0.0000i   4.0000 + 0.0000i
   6.0000 + 0.0000i   7.0000 + 0.0000i   3.0000 + 4.0000i
```

Round Duration Values Toward Positive Infinity

Round each value in a duration array to the nearest number of seconds greater than or equal to that value.

```
t = hours(8) + minutes(29:31) + seconds(1.23);
t.Format = 'hh:mm:ss.SS'
```

```
t =

   08:29:01.23   08:30:01.23   08:31:01.23
```

```
Y1 = ceil(t)
```

```
Y1 =

   08:29:02.00   08:30:02.00   08:31:02.00
```

Round each value in t to the nearest number of hours greater than or equal to that value.

```
Y2 = ceil(t,'hours')
```

```
Y2 =

   09:00:00.00   09:00:00.00   09:00:00.00
```

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex `X`, `ceil` treats the real and imaginary parts independently.

`ceil` converts logical and `char` elements of `X` into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

t — Input duration

`duration` array

Input duration, specified as a `duration` array.

`unit` — Unit of time

`'seconds'` (default) | `'minutes'` | `'hours'` | `'days'`

Unit of time, specified as `'seconds'`, `'minutes'`, `'hours'`, or `'days'`.

More About

- "Integers"
- "Floating-Point Numbers"

See Also

`fix` | `floor` | `round`

Introduced before R2006a

cell

Create cell array

Syntax

```
C = cell(n)
C = cell(sz1,...,szN)
C = cell(sz)

D = cell(obj)
```

Description

`C = cell(n)` returns an `n`-by-`n` cell array of empty matrices.

`C = cell(sz1,...,szN)` returns a `sz1`-by-...-by-`szN` cell array of empty matrices where `sz1,...,szN` indicate the size of each dimension. For example, `cell(2,3)` returns a 2-by-3 cell array.

`C = cell(sz)` returns a cell array of empty matrices where size vector `sz` defines `size(C)`. For example, `cell([2 3])` returns a 2-by-3 cell array.

`D = cell(obj)` converts a Java array or .NET `System.String` or `System.Object` array into a MATLAB cell array.

Examples

Square Cell Array

Create a 3-by-3 cell array of empty matrices.

```
C = cell(3)
```

```
C =

    []    []    []
```

```
    []    []    []
    []    []    []
```

3-D Cell Array

Create a 3-by-4-by-2 cell array of empty matrices.

```
C = cell(3,4,2);
size(C)
```

```
ans =

     3     4     2
```

Clone Size from Existing Array

Create a cell array of empty matrices that is the same size as an existing array.

```
A = [7 9; 2 1; 8 3];
sz = size(A);
C = cell(sz)
```

```
C =

    []    []
    []    []
    []    []
```

It is a common pattern to combine the previous two lines of code into a single line.

```
C = cell(size(A));
```

Cell Array from Java Array

Convert an array of `java.lang.String` objects into a cell array.

```
X = java_array('java.lang.String', 3);
X(1) = java.lang.String('one');
X(2) = java.lang.String('two');
X(3) = java.lang.String('three');
```

```
D = cell(X)

D =

    'one'
    'two'
    'three'
```

Cell Array from .NET Array

Convert a .NET array of `System.Double` objects into a cell array.

```
N = NET.createArray('System.Double[]',2);
N(1) = [13 7 30];
N(2) = 42;
D = cell(N)

D =

    [1x3 double]    [42]
```

- "Create a Cell Array"
- "Access Data in a Cell Array"
- "Use Python list Type in MATLAB"

Input Arguments

n — Size of square cell array

integer value

Size of a square cell array, specified as an integer value.

- If `n` is `0`, then `C` is an empty cell array.
- If `n` is negative, then it is treated as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN — Size of each dimension (as separate arguments)

integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `C` is an empty cell array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `cell` ignores trailing dimensions with a size of 1. For example, `cell([3,1,1,1])` produces a 3-by-1 cell array of empty matrices.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)

integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is `0`, then `C` is an empty cell array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `cell` ignores trailing dimensions with a size of 1. For example, `cell([3,1,1,1])` produces a 3-by-1 cell array of empty matrices.

Example: `sz = [2, 3, 4]` creates a 2-by-3-by-4 cell array of empty matrices.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`obj` — Input array

Java array or object | .NET array of type `System.String` or `System.Object` | Python® sequence type

Input array, specified as:

- Java array or object
- .NET array of type `System.String` or `System.Object`
- Python sequence type

Output Arguments

`C` — Output array

cell array

Output array, returned as a cell array. Each cell contains an empty, 0-by-0 array of type `double`.

`D` — Converted array

cell array

Converted array, returned as a cell array.

Each cell contains a MATLAB type closest to the Java or .NET type. For more information, see:

- "Conversion of Java Return Types"
- ".NET Type to MATLAB Type Mapping"

More About

Tips

- Creating an empty array with the `cell` function, such as

  ```
  C = cell(3,4,2);
  ```

  is equivalent to assigning an empty array to the last index of a new cell array:

  ```
  C{3,4,2} = [];
  ```

See Also

`num2cell` | `table2cell`

Introduced before R2006a

cell2mat

Convert cell array to ordinary array of the underlying data type

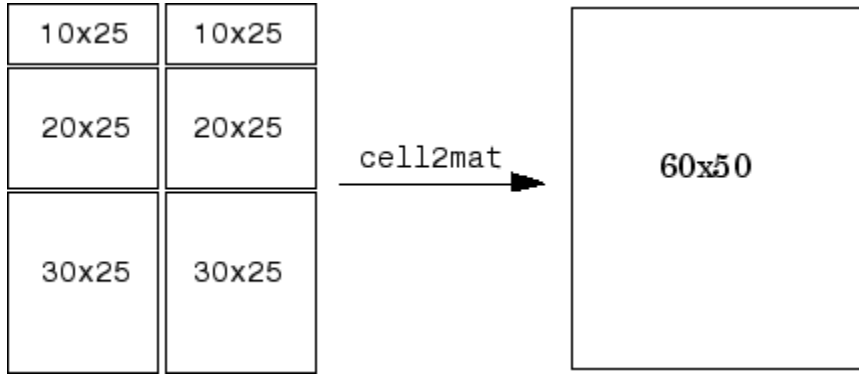
Syntax

```
A = cell2mat(C)
```

Description

`A = cell2mat(C)` converts a cell array into an ordinary array. The elements of the cell array must all contain the same data type, and the resulting array is of that data type.

The contents of `C` must support concatenation into an N-dimensional rectangle. Otherwise, the results are undefined. For example, the contents of cells in the same column must have the same number of columns, although they need not have the same number of rows (see figure).



Examples

Convert Cell Array to Numeric Array

Convert numeric arrays in four cells of a cell array into one numeric array.

```
C = {[1],    [2 3 4];
     [5; 9], [6 7 8; 10 11 12]}

C =

    [         1]    [1x3 double]
    [2x1 double]    [2x3 double]

A = cell2mat(C)

A =

     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Convert Cell Array of Structures to Array

Convert structures in a cell array into one structure array. The structures must have the same fields.

```
s1.a = [1 2 3 4];
s1.b = 'Good';
s2.a = [5 6; 7 8];
s2.b = 'Morning';
c = {s1,s2};
d = cell2mat(c)

d =

1x2 struct array with fields:

    a
    b
```

Display the first field of structure `d(1)`.

```
d(1).a

ans =

     1     2     3     4
```

Display the second field of `d(2)`.

```
d(2).b
```

```
ans =

Morning
```

Input Arguments

C — Input cell array

cell array

Input cell array, in which all cells contain the same data type. `cell2mat` accepts numeric or character data within cells of `C`, or structures with the same field names and data types. `cell2mat` does not accept objects or nested cells within `C`.

See Also

`cell` | `cell2struct` | `cell2table` | `iscell` | `mat2cell` | `num2cell` | `struct2cell` | `table2cell`

Introduced before R2006a

cell2struct

Convert cell array to structure array

Syntax

```
structArray = cell2struct(cellArray, fields, dim)
```

Description

`structArray = cell2struct(cellArray, fields, dim)` creates a structure array, *`structArray`*, from the information contained within cell array *`cellArray`*.

The *`fields`* argument specifies field names for the structure array. This argument is an array of strings or a cell array of strings.

The *`dim`* argument tells MATLAB which axis of the cell array to use in creating the structure array. Use a numeric `double` to specify *`dim`*.

To create a structure array with fields derived from `N` rows of a cell array, specify `N` field names in the *`fields`* argument, and the number 1 in the *`dim`* argument. To create a structure array with fields derived from `M` columns of a cell array, specify `M` field names in the *`fields`* argument and the number 2 in the *`dim`* argument.

The *`structArray`* output is a structure array with `N` fields, where `N` is equal to the number of fields in the *`fields`* input argument. The number of fields in the resulting structure must equal the number of cells along dimension *`dim`* that you want to convert.

Examples

Create the following table for use with the examples in this section. The table lists information about the employees of a small Engineering company. Reading the table by rows shows the names of employees by department. Reading the table by columns shows the number of years each employee has worked at the company.

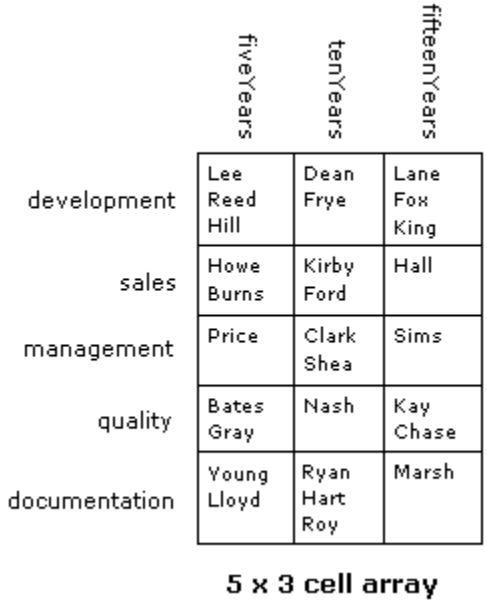
	5 Years	10 Years	15 Years
Development	Lee, Reed, Hill	Dean, Frye	Lane, Fox, King

	5 Years	10 Years	15 Years
Sales	Howe, Burns	Kirby, Ford	Hall
Management	Price	Clark, Shea	Sims
Quality	Bates, Gray	Nash	Kay, Chase
Documentation	Lloyd, Young	Ryan, Hart, Roy	Marsh

Enter the following commands to create the initial cell array `employees`:

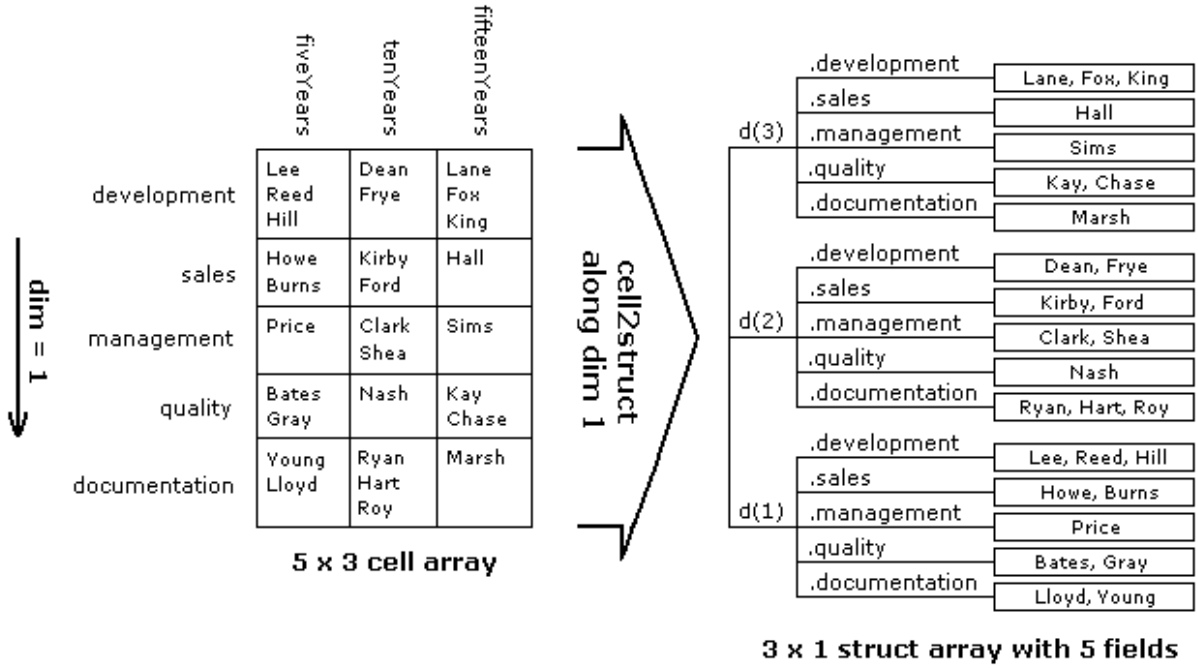
```
devel = {{'Lee','Reed','Hill'}, {'Dean','Frye'}, ...
   {'Lane','Fox','King'}};
sales = {{'Howe','Burns'}, {'Kirby','Ford'}, {'Hall'}};
mgmt = {{'Price'}, {'Clark','Shea'}, {'Sims'}};
qual = {{'Bates','Gray'}, {'Nash'}, {'Kay','Chase'}};
docu = {{'Lloyd','Young'}, {'Ryan','Hart','Roy'}, {'Marsh'}};

employees = [devel; sales; mgmt; qual; docu]
employees =

    {1x3 cell}    {1x2 cell}    {1x3 cell}
    {1x2 cell}    {1x2 cell}    {1x1 cell}
    {1x1 cell}    {1x2 cell}    {1x1 cell}
    {1x2 cell}    {1x1 cell}    {1x2 cell}
    {1x2 cell}    {1x3 cell}    {1x1 cell}
```

This is the resulting cell array:



Convert the cell array to a struct along dimension 1:

1 Convert the 5-by-3 cell array along its first dimension to construct a 3-by-1 struct array with 5 fields. Each of the rows along dimension 1 of the cell array becomes a field in the struct array:



Traversing the first (i.e., vertical) dimension, there are 5 rows with row headings that read as follows:

```
rowHeadings = {'development', 'sales', 'management', ...
   'quality', 'documentation'};
```

2 Convert the cell array to a struct array, `depts`, in reference to this dimension:

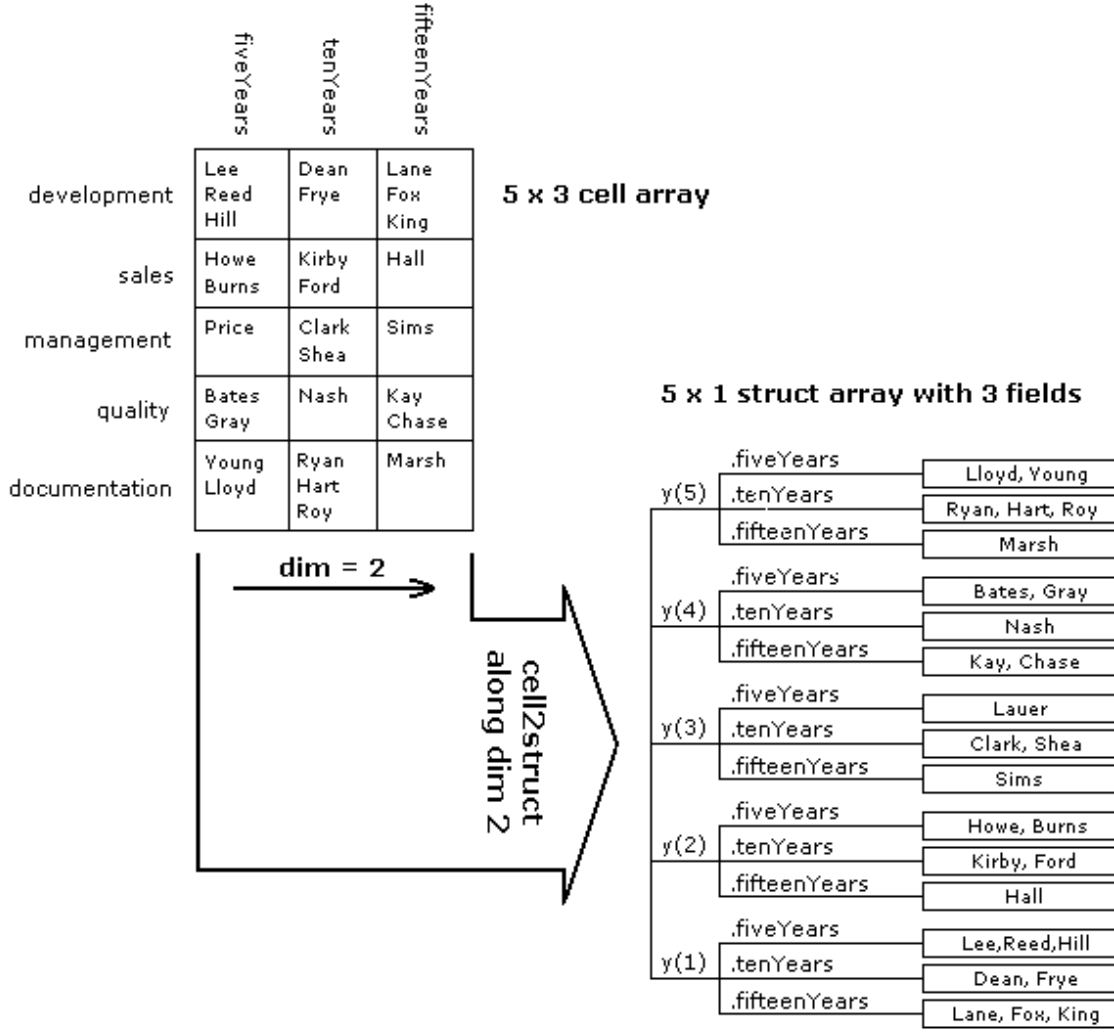
```
depts = cell2struct(employees, rowHeadings, 1)
depts =
3x1 struct array with fields:
    development
    sales
    management
    quality
    documentation
```

3 Use this row-oriented structure to find the names of the Development staff who have been with the company for up to 10 years:

```
depts(1:2).development
ans =
    'Lee'    'Reed'    'Hill'
ans =
    'Dean'    'Frye'
```

Convert the same cell array to a struct along dimension 2:

1. Convert the 5-by-3 cell array along its second dimension to construct a 5-by-1 struct array with 3 fields. Each of the columns along dimension 2 of the cell array becomes a field in the struct array:



2 Traverse the cell array along the second (or horizontal) dimension. The column headings become fields of the resulting structure:

```
colHeadings = {'fiveYears' 'tenYears' 'fifteenYears'};
```

```
years = cell2struct(employees, colHeadings, 2)
years =
5x1 struct array with fields:
    fiveYears
    tenYears
    fifteenYears
```

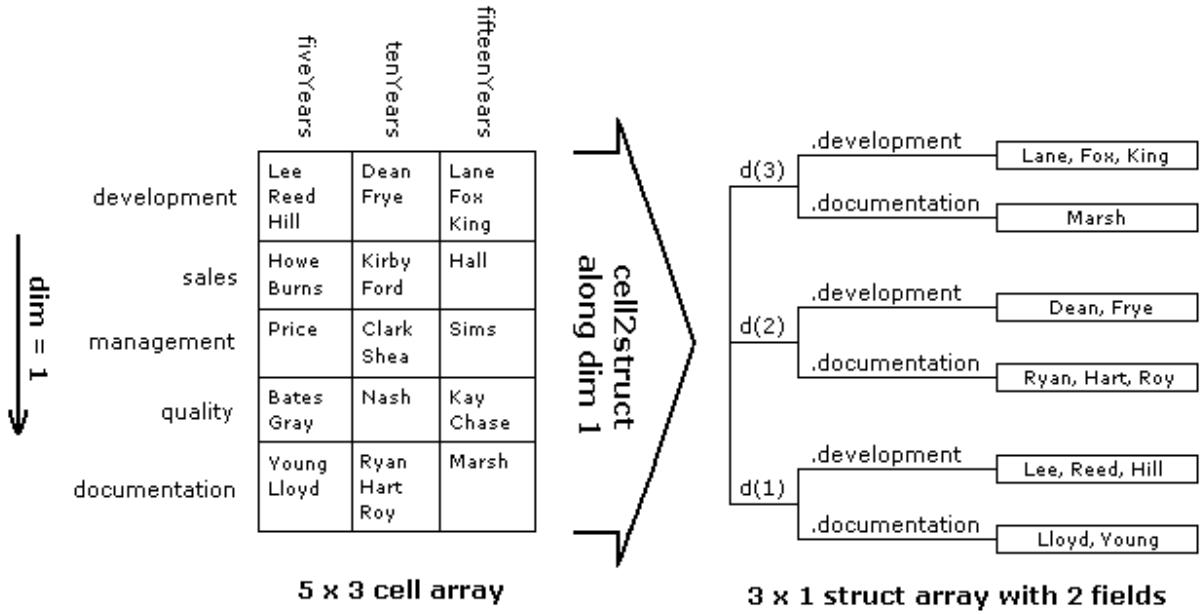
3 Using the column-oriented structure, show how many employees from the Sales and Documentation departments have worked for the company for at least 5 years:

```
[~, sales_5years, ~, ~, docu_5years] = years.fiveYears
sales_5years =
    'Howe'    'Burns'
docu_5years =
    'Lloyd'    'Young'
```

Convert only part of the cell array to a struct:

1 Convert only the first and last rows of the cell array. This results in a 3-by-1 struct array with 2 fields:

```
rowHeadings = {'development', 'documentation'};

depts = cell2struct(employees([1,5],:), rowHeadings, 1)
depts =
3x1 struct array with fields:
    development
    documentation
```



2 Display those employees who belong to these departments for all three periods of time:

```
for k=1:3
   depts(k,:)
end

ans =
      development: {'Lee'  'Reed'  'Hill'}
    documentation: {'Lloyd'  'Young'}
ans =
      development: {'Dean'  'Frye'}
    documentation: {'Ryan'  'Hart'  'Roy'}
ans =
      development: {'Lane'  'Fox'  'King'}
    documentation: {'Marsh'}
```

More About

- dynamic field names

See Also

`struct2cell` | `cell2table` | `table2struct` | `cell` | `iscell` | `struct` | `isstruct` | `fieldnames`

Introduced before R2006a

cell2table

Convert cell array to table

Syntax

```
T = cell2table(C)
T = cell2table(C,Name,Value)
```

Description

`T = cell2table(C)` converts the contents of an `m`-by-`n` cell array, `C`, to an `m`-by-`n` table, `T`. Each variable in the table, `T`, is numeric, with a data type `double`, or a cell array of strings.

`cell2table` uses the input array name appended with the column number for the variable names in the table. If these names are not valid MATLAB identifiers, `cell2table` uses strings of the form `'Var1',...,'VarN'` where *N* is the number of columns in `C`.

`T = cell2table(C,Name,Value)` creates a table from a cell array, `C`, with additional options specified by one or more Name,Value pair arguments.

For example, you can specify row names or variable names to include in the table.

Examples

Convert Cell Array to Table

Create a cell array containing strings and numeric data.

```
C = {5 'cereal' 110 'C+'; 12 'pizza' 140 'B';...
    23 'salmon' 367 'A'; 2 'cookies' 160 'D'}
```

```
C =

    [ 5]    'cereal'    [110]    'C+'
```

```
    [12]    'pizza'      [140]    'B'
    [23]    'salmon'     [367]    'A'
    [ 2]    'cookies'    [160]    'D'
```

Convert the cell array, `C`, to a table and specify variable names.

```
T = cell2table(C,...
    'VariableNames',{'Age' 'FavoriteFood' 'Calories' 'NutritionGrade'})
```

```
T =

    Age    FavoriteFood    Calories    NutritionGrade
    ___    ____________    ________    ______________

     5     'cereal'        110         'C+'
    12     'pizza'         140         'B'
    23     'salmon'        367         'A'
     2     'cookies'       160         'D'
```

The variables `T.Age` and `T.Calories` are numeric while the variables `T.FavoriteFood` and `T.NutritionGrade` are cell arrays of strings.

Convert Column Headings to Variable Names

Convert a cell array to a table, and then include the first row from the cell array as variable names for the table.

Create a cell array where the first row contains strings to identify column headings.

```
Patients = {'Gender' 'Age' 'Height' 'Weight' 'Smoker';...
    'M' 38 71 176 true;...
    'M' 43 69 163 false;...
    'M' 38 64 131 false;...
    'F' 38 64 131 false;...
    'F' 40 67 133 false;...
    'F' 49 64 119 false}
```

```
Patients =

    'Gender'    'Age'    'Height'    'Weight'    'Smoker'
    'M'         [ 38]    [    71]    [   176]    [     1]
    'M'         [ 43]    [    69]    [   163]    [     0]
    'M'         [ 38]    [    64]    [   131]    [     0]
    'F'         [ 38]    [    64]    [   131]    [     0]
    'F'         [ 40]    [    67]    [   133]    [     0]
```

```
    'F'         [ 49]    [    64]    [   119]    [     0]
```

Exclude the columns headings and convert the contents of the cell array to a table.

```
C = Patients(2:end,:);
T = cell2table(C)
```

```
T =

    C1     C2    C3    C4      C5
    ___    __    __    ___    _____

    'M'    38    71    176    true
    'M'    43    69    163    false
    'M'    38    64    131    false
    'F'    38    64    131    false
    'F'    40    67    133    false
    'F'    49    64    119    false
```

The table, `T`, has variable names `C1,...,C5`.

Change the variable names by setting the table property, `T.Properties.VariableNames`, to the first row of the cell array.

```
T.Properties.VariableNames = Patients(1,:)
```

```
T =

    Gender    Age    Height    Weight    Smoker
    ______    ___    ______    ______    ______

    'M'       38     71        176       true
    'M'       43     69        163       false
    'M'       38     64        131       false
    'F'       38     64        131       false
    'F'       40     67        133       false
    'F'       49     64        119       false
```

- “Access Data in a Table”

Input Arguments

`C` — Input cell array

2-D cell array

Input cell array, specified as a 2-D cell array.

If the cell contents have compatible sizes and types, then `cell2table` vertically concatenates the contents of the cells in each column of `C` to create each variable in `T`. If the cell contents have different sizes or incompatible types, then the corresponding variable in the table, `T`, is a column of cells.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `RowNames',{'row1','row2','row3'}` uses the row names, `row1`, `row2`, and `row3` for the table, `T`.

`'RowNames'` — Row names for `T`
`{}` (default) | cell array of nonempty, distinct strings

Row names for `T`, specified as the comma-separated pair consisting of `'RowNames'` and a cell array of nonempty, distinct strings. The number of strings must equal the number of rows, `size(C,1)`.

`'VariableNames'` — Variable names for `T`
cell array of nonempty, distinct strings

Variable names for `T`, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of nonempty, distinct strings. The number of strings must equal the number of variables, `size(C,2)`.

Furthermore, the strings must be valid MATLAB identifiers. If valid MATLAB identifiers are not available for use as variable names, MATLAB uses a cell array of `N` strings of the form `{'Var1' ... 'VarN'}`, where *N* is the number of variables. You can determine valid MATLAB variable names using the function `isvarname`.

Output Arguments

`T` — Output table
table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

See Also

`array2table` | `isvarname` | `struct2table` | `table` | `table2cell`

celldisp

Display cell array contents

Syntax

```
celldisp(C)
celldisp(C, name)
```

Description

`celldisp(C)` recursively displays the contents of a cell array.

`celldisp(C, name)` uses the string *`name`* for the display instead of the name of the first input (or `ans`).

Examples

Use `celldisp` to display the contents of a 2-by-3 cell array:

```
C = {[1 2] 'Tony' 3+4i; [1 2;3 4] -5 'abc'};
celldisp(C)

C{1,1} =
     1     2

C{2,1} =
     1     2
     3     4

C{1,2} =
Tony

C{2,2} =
    -5

C{1,3} =
   3.0000+ 4.0000i
```

```
C{2,3} =
abc
```

More About

- “Export Cell Array to Text File”

See Also

`cellplot`

Introduced before R2006a

cellfun

Apply function to each cell in cell array

Syntax

```
[A1,...,Am] = cellfun(func,C1,...,Cn)
[A1,...,Am] = cellfun(func,C1,...,Cn,Name,Value)
```

Description

`[A1,...,Am] = cellfun(func,C1,...,Cn)` calls the function specified by function handle `func` and passes elements from cell arrays `C1,...,Cn`, where `n` is the number of inputs to function `func`. Output arrays `A1,...,Am`, where `m` is the number of outputs from function `func`, contain the combined outputs from the function calls. The `i`th iteration corresponds to the syntax `[A1(i),...,Am(i)] = func(C1{i},...,Cn{i})`. The `cellfun` function does not perform the calls to function `func` in a specific order.

`[A1,...,Am] = cellfun(func,C1,...,Cn,Name,Value)` calls function `func` with additional options specified by one or more Name,Value pair arguments. Possible values for `Name` are `'UniformOutput'` or `'ErrorHandler'`.

Input Arguments

`func`

Handle to a function that accepts `n` input arguments and returns `m` output arguments.

If function `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

Backward Compatibility

`cellfun` accepts function name strings for function `func`, rather than a function handle, for these function names: `isempty`, `islogical`, `isreal`, `length`, `ndims`, `prodofsize`, `size`, `isclass`. Enclose the function name in single quotes.

If you specify a function name string rather than a function handle:

- `cellfun` does not call any overloaded versions of the function.
- The `size` and `isclass` functions require additional inputs to the `cellfun` function:

  `A = cellfun('size', C, k)` returns the size along the `k`th dimension of each element of `C`.

  `A = cellfun('isclass', C, classname)` returns logical `1` (`true`) for each element of `C` that matches the `classname` string. This syntax returns logical 0 (`false`) for objects that are a subclass of `classname`.

`C1,...,Cn`

Cell arrays that contain the `n` inputs required for function `func`. Each cell array must have the same dimensions.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'UniformOutput'`

Logical value, as follows:

`true (1)`	Indicates that for all inputs, each output from function `func` is a cell array or a scalar value that is always of the same type and size. The `cellfun` function combines the outputs in arrays `A1,...,Am`, where `m` is the number of function outputs. Each output array is of the same type as the individual function outputs.
`false (0)`	Requests that the `cellfun` function combine the outputs into cell arrays `A1,...,Am`. The outputs of function `func` can be of any size or type.

Default: `true`

`'ErrorHandler'`

Handle to a function that catches any errors that occur when MATLAB attempts to execute function `func`. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Linear index corresponding to the element of the input cell array at the time of the error.

- The set of input arguments to function `func` at the time of the error.

Output Arguments

`A1,...,Am`

Arrays that collect the `m` outputs from function `func`. Each array `A` is the same size as each of the inputs `C1,...,Cn`.

Function `func` can return output arguments of different classes. However, if `UniformOutput` is `true` (the default):

- The individual outputs from function `func` must be scalar values (numeric, logical, character, or structure) or cell arrays.
- The class of a particular output argument must be the same for each set of inputs. The class of the corresponding output array is the same as the class of the outputs from function `func`.

Examples

Compute the mean of each vector in cell array `C`.

```
C = {1:10, [2; 4; 6], []};
```

```
averages = cellfun(@mean, C)
```

This code returns

```
averages =
    5.5000    4.0000       NaN
```

Compute the size of each array in `C`, created in the previous example.

```
[nrows, ncols] = cellfun(@size, C)
```

This code returns

```
nrows =
     1     3     0
ncols =
    10     1     0
```

Create a cell array that contains strings, and abbreviate those strings to the first three characters. Because the output strings are nonscalar, set `UniformOutput` to `false`.

```
days = {'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'};

abbrev = cellfun(@(x) x(1:3), days, 'UniformOutput', false)
```

The syntax `@(x)` creates an anonymous function. This code returns

```
abbrev =
    'Mon'    'Tue'    'Wed'    'Thu'    'Fri'
```

Compute the covariance between arrays in two cell arrays `C` and `D`. Because the covariance output is nonscalar, set `UniformOutput` to `false`.

```
c1 = rand(5,1);  c2 = rand(10,1);  c3 = rand(15,1);
d1 = rand(5,1);  d2 = rand(10,1);  d3 = rand(15,1);
C = {c1, c2, c3};
D = {d1, d2, d3};

covCD = cellfun(@cov, C, D, 'UniformOutput', false)
```

This code returns

```
covCD =
    [2x2 double]    [2x2 double]    [2x2 double]
```

Define and call a custom error handling function.

```
function result = errorfun(S, varargin)
   warning(S.identifier, S.message);
   result = NaN;
end

A = {rand(3)};
B = {rand(5)};
AgtB = cellfun(@(x,y) x > y, A, B, 'ErrorHandler', @errorfun, ...
               'UniformOutput', false)
```

See Also

arrayfun | spfun | function_handle | cell2mat

Introduced before R2006a

cellplot

Graphically display structure of cell array

Syntax

```
cellplot(c)
cellplot(c, 'legend')
handles = cellplot(c)
```

Description

`cellplot(c)` displays a figure window that graphically represents the contents of `c`. Filled rectangles represent elements of vectors and arrays, while scalars and short text strings are displayed as text.

`cellplot(c, 'legend')` places a colorbar next to the plot labelled to identify the data types in `c`.

`handles = cellplot(c)` displays a figure window and returns a vector of surface handles.

Limitations

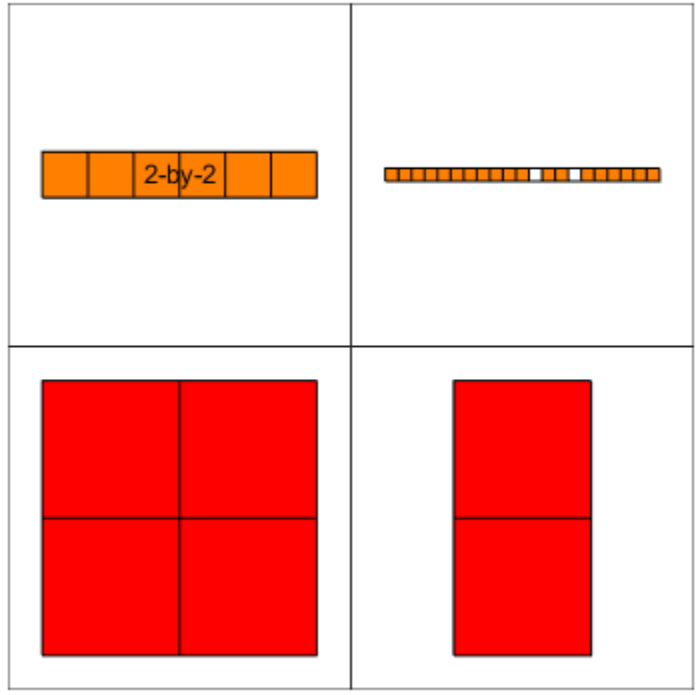
The `cellplot` function can display only two-dimensional cell arrays.

Examples

Consider a 2-by-2 cell array containing a matrix, a vector, and two text strings:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces



More About

- “Export Cell Array to Text File”

See Also

`celldisp`

Introduced before R2006a

cellstr

Convert to cell array of strings

Syntax

```
C = cellstr(S)

S = cellstr(D)
S = cellstr(D,fmt)
S = cellstr(D,fmt,locale)
```

Description

`C = cellstr(S)` converts array `S` to a cell array.

- If `S` is a character array, then each row of `S` is a cell of `C`. Any trailing spaces in the rows of `S` are removed. Use the `char` function to convert back into a character array.
- If `S` is a cell array of strings, then `cellstr` returns `S` unaltered.
- If `S` is a categorical array, then `cellstr` returns a cell array of the same size.

`S = cellstr(D)` converts a datetime, duration, or calendar duration array into a cell array of strings in the format specified by the `Format` property of `D`. The output has the same dimensions as `D`.

`S = cellstr(D,fmt)` represents dates or durations in the specified format, such as `'HH:mm:ss'`.

`S = cellstr(D,fmt,locale)` represents dates or durations in the specified locale, such as `'en_US'`. The locale affects the language used to represent character strings such as month and day names.

Examples

Convert Character Array to Cell Array of Strings

Create a character array. Include trailing spaces so that each row has the same length, resulting in a 3-by-4 array.

```
S = ['abc ';'defg';'hi  ']
```

```
S =

abc
defg
hi
```

```
whos S
```

```
 Name      Size            Bytes  Class    Attributes

  S         3x4                24  char
```

Convert the character array to a 3-by-1 cell array of strings.

```
C = cellstr(S)
```

```
C =

    'abc'
    'defg'
    'hi'
```

```
whos C
```

```
 Name      Size            Bytes  Class    Attributes

  C         3x1               354  cell
```

Convert Calendar Duration Array to Cell Array of Strings

Create a `calendarDuration` array.

```
D = calmonths(15:17) + caldays(8) + hours(1.2345)
```

```
D =

   1y 3mo 8d 1h 14m 4.2s   1y 4mo 8d 1h 14m 4.2s   1y 5mo 8d 1h 14m 4.2s
```

Convert the array to a cell array of strings.

```
C = cellstr(D)
```

```
C =

    '1y 3mo 8d 1h 14m ...'    '1y 4mo 8d 1h 14m ...'    '1y 5mo 8d 1h 14m ...'
```

```
whos C
```

```
  Name      Size            Bytes  Class    Attributes

  C         1x3               462  cell
```

Input Arguments

`S` — Input array

character array | cell array of strings | categorical array

Input array, specified as a character array, a cell array of strings, or a categorical array.

If `S` is a character array, then each row of `S` becomes a separate cell of `C`, with trailing spaces removed.

If `S` is a cell array of strings, then `cellstr` returns `S` unaltered.

Data Types: `char` | `cell`

`D` — Input date and time

date or duration array

Input date and time, specified as a date or duration array.

Data Types: `datetime` | `duration` | `calendarDuration`

`fmt` — Date format

string

Date format, specified as a string. The supported formats depend on the data type of input `D`.

- `datetime` formats can include combinations of units and delimiters, such as `'yyyy-MMM-dd HH:mm:ss.SSS'`. For details, see the `Format` property for datetime arrays.
- `duration` formats are either single characters (`'y'`, `'d'`, `'h'`, `'m'`, or `'s'`) or one of these combinations:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`
- Any of the above, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`

- `calendarDuration` formats can include combinations of the characters `'y'`, `'q'`, `'m'`, `'w'`, `'d'`, and `'t'` in order from largest to smallest unit of time, such as `'ym'`.

For more information on the `duration` and `calendarDuration` formats, see "Set Date and Time Display Format".

`locale` — Locale of the character strings to create

string

Locale of the character strings to create, specified as a string.

`locale` can be:

- `'system'`, to specify your system locale.
- a string in the form *`xx`*`_`*`YY`*, where *`xx`* is a lowercase ISO 639-1 two-letter code that specifies a language, and *`YY`* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

These are the same strings accepted by the `'Locale'` name-value pair argument for the `datetime` function. The locale affects the language used in the output array.

Example: `'en_US'`

Example: `'ja_JP'`

See Also

`char` | `iscellstr` | `isstrprop` | `strsplit`

Introduced before R2006a

cgs

Conjugate gradients squared method

Syntax

```
x = cgs(A,b)
cgs(A,b,tol)
cgs(A,b,tol,maxit)
cgs(A,b,tol,maxit,M)
cgs(A,b,tol,maxit,M1,M2)
cgs(A,b,tol,maxit,M1,M2,x0)
[x,flag] = cgs(A,b,...)
[x,flag,relres] = cgs(A,b,...)
[x,flag,relres,iter] = cgs(A,b,...)
[x,flag,relres,iter,resvec] = cgs(A,b,...)
```

Description

`x = cgs(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `cgs` converges, a message to that effect is displayed. If `cgs` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`cgs(A,b,tol)` specifies the tolerance of the method, `tol`. If `tol` is `[]`, then `cgs` uses the default, `1e-6`.

`cgs(A,b,tol,maxit)` specifies the maximum number of iterations, `maxit`. If `maxit` is `[]` then `cgs` uses the default, `min(n,20)`.

`cgs(A,b,tol,maxit,M)` and `cgs(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `cgs` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`cgs(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess `x0`. If `x0` is `[]`, then `cgs` uses the default, an all-zero vector.

`[x,flag] = cgs(A,b,...)` returns a solution `x` and a flag that describes the convergence of `cgs`.

Flag	Convergence
0	`cgs` converged to the desired tolerance `tol` within `maxit`iterations.
1	`cgs` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`cgs` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `cgs` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = cgs(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres <= tol`.

`[x,flag,relres,iter] = cgs(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = cgs(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Using cgs with a Matrix Input

```
A = gallery('wilk',21);
```

```
b = sum(A,2);
tol = 1e-12;  maxit = 15;
M1 = diag([10:-1:1 1 1:10]);
x = cgs(A,b,tol,maxit,M1);
```

displays the message

```
cgs converged at iteration 13 to a solution with
relative residual 2.4e-016.
```

Using cgs with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in the file `run_cgs` that

- Calls `cgs` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_cgs` are available to `afun` and `myfun`.

The following shows the code for `run_cgs`:

```
function x1 = run_cgs
n = 21;
b = afun(ones(n,1));
tol = 1e-12;  maxit = 15;
x1 = cgs(@afun,b,tol,maxit,@mfun);

    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_cgs
```

MATLAB software returns

```
cgs converged at iteration 13 to a solution with
relative residual 2.4e-016.
```

Using cgs with a Preconditioner.

This example demonstrates the use of a preconditioner.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

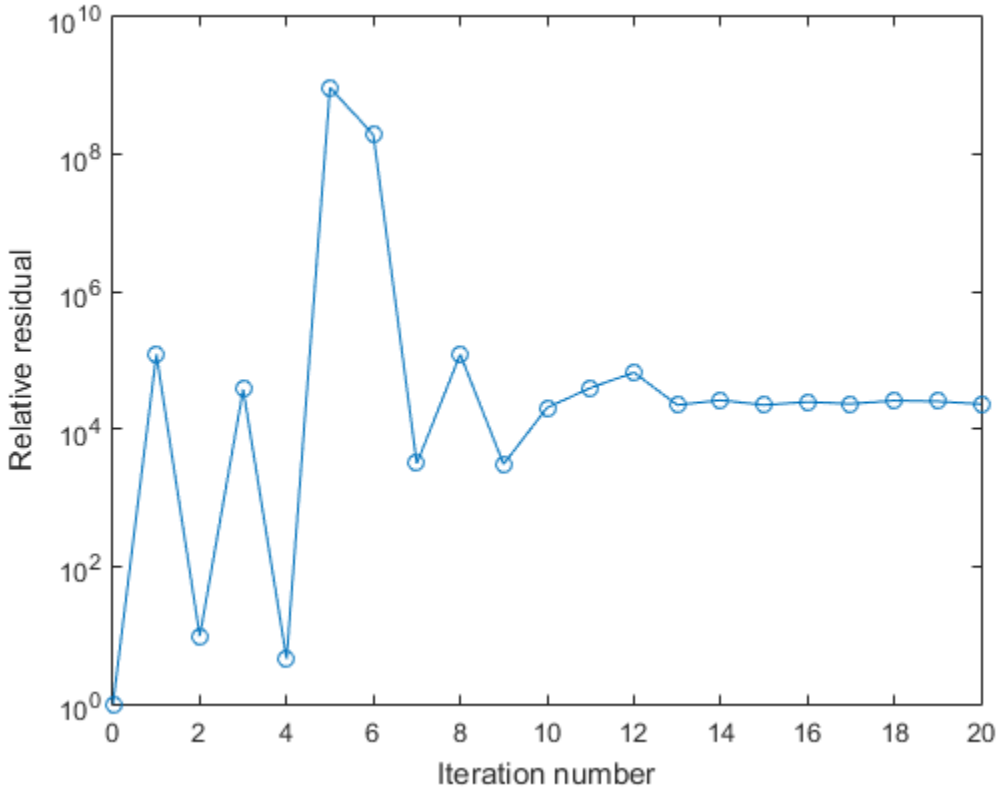
Use `cgs` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = cgs(A,b,tol,maxit);
```

`fl0` is 1 because `cgs` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `cgs` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB stores the residual history in `rv0`.

Plot the behavior of `cgs`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create a preconditioner with `ilu`, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero.  Consider decreasing the
drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

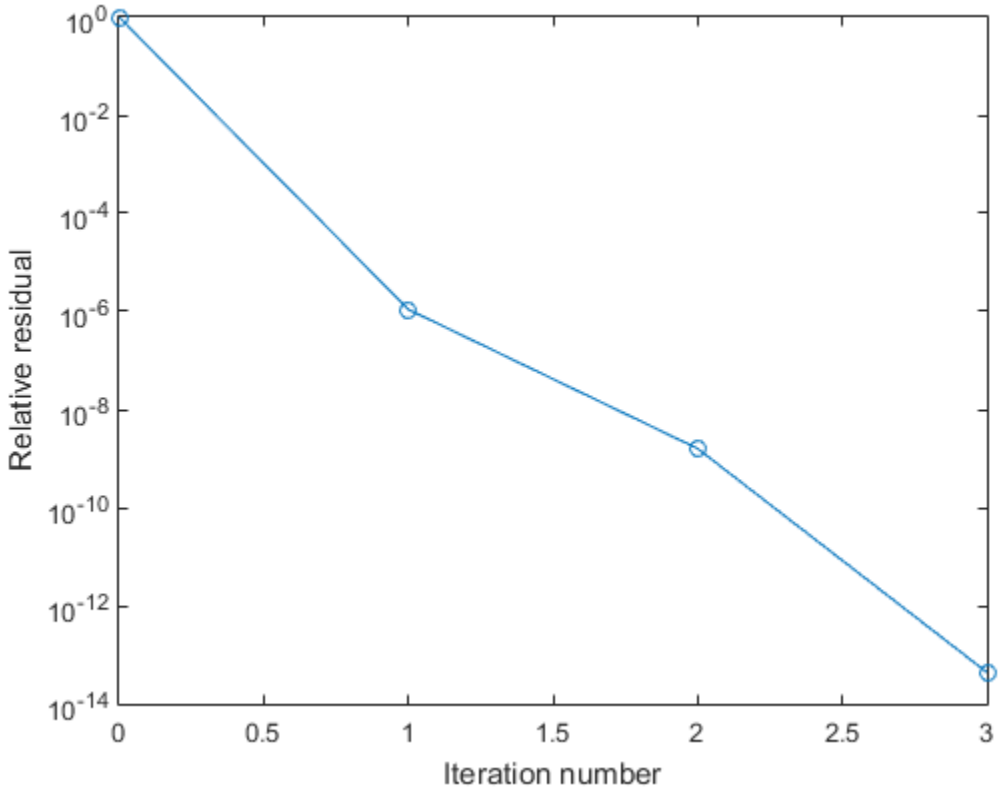
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = cgs(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `cgs` drives the relative residual to `4.3851e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the third iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)` and the output `rv1(14)` is `norm(b-A*x2)`.

You can follow the progress of `cgs` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Sonneveld, Peter, "CGS: A fast Lanczos-type solver for nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, January 1989, Vol. 10, No. 1, pp. 36–52.

See Also

`bicg` | `bicgstab` | `function_handle` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Introduced before R2006a

char

Convert to character array (string)

Syntax

```
S = char(A)
S = char(A1,...,AN)

S = char(D)
S = char(D,fmt)
S = char(D,fmt,locale)
```

Description

`S = char(A)` converts array `A` into a character array.

- If `A` is a numeric array, then `char` converts numbers into characters. Valid numeric values range from 0 to 65535, with 0 to 127 corresponding to 7-bit ASCII characters. Values from 128 to 65535 produce characters that depend upon your locale setting.
- If `A` is a character array, then `char` returns `A` unaltered.

  To convert characters into a numeric array, use a function that converts to a numeric type (for example, `double`, `int32`, or `cast`).
- If `A` is a cell array of strings, then `char` converts the cell array into a character array. The strings from the cell array become rows in `S` (automatically padded with blanks as needed).

  To convert `S` back into a cell array, use the `cellstr` function.
- If `A` is a categorical array, then `char` converts each element of `A` into a row of a character array, in column order.

`S = char(A1,...,AN)` converts the arrays `A1,...,AN` into a single character array. After conversion to characters, the input arrays become rows in `S`. Each row is automatically padded with blanks as needed. An empty string becomes a row of blanks.

The input arrays `A1,...,AN` cannot be cell arrays or categorical arrays.

`A1,...,AN` can be of different sizes and shapes.

`S = char(D)` converts a datetime, duration, or calendar duration array into a character array in the format specified by the `Format` property of `D`. The output contains one date or duration in each row.

`S = char(D,fmt)` represents dates or durations in the specified format, such as `'HH:mm:ss'`.

`S = char(D,fmt,locale)` represents dates or durations in the specified locale, such as `'en_US'`. The locale affects the language used to represent character strings such as month and day names.

Examples

Convert Integers to Printable ASCII Character Array

Convert the integers 32–127 into a 3-by-32 array of the printable ASCII characters.

```
A = reshape(32:127,32,3)';
S = char(A)
```

```
S =

 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~#
```

Convert Multiple Arrays to Character Array

Convert multiple arrays into a single character array. The input arrays need not have the same shape.

```
A1 = [65 66; 67 68];
A2 = 'abcd';
S = char(A1,A2)
```

```
S =

AB
CD
abcd
```

Because the input arrays do not have the same number of columns, `char` pads the rows from `A1` with blanks.

```
whos S
```

```
  Name      Size            Bytes  Class    Attributes

  S         3x4                24  char
```

Convert Cell Array of Strings to Character Array

Convert a cell array of strings into a character array.

```
A = {'ABCD','efghijk','','LM'};
S = char(A)
```

```
S =

ABCD
efghijk

LM
```

Because the input strings from cell array `A` have different lengths, `char` pads the strings with blanks. `char` pads the third row with seven blanks, because the third cell of `A` contains an empty string.

```
whos S
```

```
  Name      Size            Bytes  Class    Attributes

  S         4x7                56  char
```

Convert Duration Array to Character Array

Create a `duration` array.

```
D = hours(23:25) + minutes(8) + seconds(1.2345)
```

```
D =

   23.134 hrs   24.134 hrs   25.134 hrs
```

Convert `D` to a character array.

```
S = char(D)
```

```
S =

23.134 hrs
24.134 hrs
25.134 hrs
```

S is a character array with one duration value per row.

Specify the format of the duration values in S.

```
S = char(D,'hh:mm')
```

```
S =

23:08
24:08
25:08
```

Input Arguments

A — Input array

numeric array | character array | cell array of strings | categorical array

Input array, specified as an array of real numbers or characters, a cell array of strings, or a categorical array.

If A is a numeric array, then:

- Nonintegers are rounded towards zero.
- Values less than 0 are treated as 0.
- Values greater than 65535 are treated as 65535.

Example: `char(65)` converts the integer 65 into the character A.

D — Input date and time

date or duration array

Input date and time, specified as a date or duration array.

Data Types: `datetime` | `duration` | `calendarDuration`

`fmt` — Date format

string

Date format, specified as a string. The supported formats depend on the data type of input `D`.

- `datetime` formats can include combinations of units and delimiters, such as `'yyyy-MMM-dd HH:mm:ss.SSS'`. For details, see the `Format` property for datetime arrays.
- `duration` formats are either single characters (`'y'`, `'d'`, `'h'`, `'m'`, or `'s'`) or one of these combinations:
  - `'dd:hh:mm:ss'`
  - `'hh:mm:ss'`
  - `'mm:ss'`
  - `'hh:mm'`
  - Any of the above, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`
- `calendarDuration` formats can include combinations of the characters `'y'`, `'q'`, `'m'`, `'w'`, `'d'`, and `'t'` in order from largest to smallest unit of time, such as `'ym'`.

For more information on the `duration` and `calendarDuration` formats, see "Set Date and Time Display Format".

`locale` — Locale of the character strings to create

string

Locale of the character strings to create, specified as a string.

`locale` can be:

- `'system'`, to specify your system locale.
- a string in the form *`xx_YY`*, where *`xx`* is a lowercase ISO 639-1 two-letter code that specifies a language, and *`YY`* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

These are the same strings accepted by the `'Locale'` name-value pair argument for the `datetime` function. The locale affects the language used in the output array.

Example: `'en_US'`

Example: `'ja_JP'`

More About

- "How the MATLAB Process Uses Locale Settings"

See Also

`cellstr` | `iscellstr` | `ischar` | `isletter` | `isspace` | `isstrprop` | `text`

Introduced before R2006a

checkcode

Check MATLAB code files for possible problems

Alternatives

For information on using the graphical user interface for checking code, see “Check Code for Errors and Warnings”.

Syntax

```
checkcode('filename')
checkcode('filename','-config=settings.txt')
checkcode('filename','-config=factory')
inform=checkcode('filename','-struct')
msg=checkcode('filename','-string')
[inform,filepaths]=checkcode('filename')
inform=checkcode('filename','-id')
inform=checkcode('filename','-fullpath')
inform=checkcode('filename','-notok')
checkcode('filename','-cyc')
checkcode('filename','-codegen')
checkcode('filename','-eml')
```

Description

`checkcode('filename')` displays messages, sometimes referred to as Code Analyzer messages, about `filename`, where the message reports potential problems and opportunities for code improvement. The line number in the message is a hyperlink that opens the file in the Editor, scrolled to that line. If `filename` is a cell array, information is displayed for each file. For `checkcode(F1,F2,F3,...)`, where each input is a character array, MATLAB software displays information about each input file name. You cannot combine cell arrays and character arrays of file names. Note that the exact text of the `checkcode` messages is subject to some change between versions.

`checkcode('filename','-config=settings.txt')` overrides the default active settings file with the settings that enable or suppress messages as indicated in the specified `settings.txt` file.

Note: If used, you must specify the full path to the `settings.txt` file specified with the `-config` option.

For information about creating a `settings.txt` file, see "Save and Reuse Code Analyzer Message Settings". If you specify an invalid file, `checkcode` returns a message indicating that it cannot open or read the file you specified. In that case, `checkcode` uses the factory default settings.

`checkcode('filename','-config=factory')` ignores all settings files and uses the factory default preference settings.

`inform=checkcode('filename','-struct')` returns the information in a structure array whose length is the number of messages found. The structure has the fields that follow.

Field	Description
`message`	Message describing the suspicious construct that code analysis caught.
`line`	Vector of file line numbers to which the message refers.
`column`	Two-column array of file columns (column extents) to which the message applies. The first column of the array specifies the column in the Editor where the message begins. The second column of the array specifies the column in the Editor where the message ends. There is one row in the two-column array for each occurrence of a message.

If you specify multiple file names as input, or if you specify a cell array as input, `inform` contains a cell array of structures.

`msg=checkcode('filename','-string')` returns the information as a string to the variable `msg`. If you specify multiple file names as input, or if you specify a cell array as input, `msg` contains a string where each file's information is separated by 10 equal sign characters (=), a space, the file name, a space, and 10 equal sign characters.

If you omit the **-struct** or **-string** argument and you specify an output argument, the default behavior is **-struct**. If you omit the argument and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepaths]=checkcode('filename')` additionally returns `filepaths`, the absolute paths to the file names, in the same order as you specified them.

`inform=checkcode('filename','-id')` requests the message ID, where ID is a string of the form `ABC...`. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=checkcode('filename','-fullpath')` assumes that the input file names are absolute paths, so that `checkcode` does not try to locate them.

`inform=checkcode('filename','-notok')` runs `checkcode` for all lines in `filename`, even those lines that end with the `checkcode` suppression directive, `%#ok`.

`checkcode('filename','-cyc')` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`checkcode('filename','-codegen')` enables code generation messages for display in the Command Window.

`checkcode('filename','-eml')` `'-eml'` is not recommended. Use `'-codegen'` instead.

Examples

The following examples use `lengthofline.m`, which is a sample file with MATLAB code that can be improved. You can find it in *`matlabroot`*`/help/techdoc/matlab_env/examples`. If you want to run the examples, save a copy of `lengthofline.m` to a location on your MATLAB path.

Running checkcode on a File with No Options

To run `checkcode` on the example file, `lengthofline.m`, run

```
checkcode('lengthofline')
```

MATLAB displays the Code Analyzer messages for `lengthofline.m` in the Command Window:

```
L 21 (C 1-9): The value assigned to variable 'nothandle' might be unused.
L 22 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 23 (C 5-11): The variable 'notline' appears to change size on every loop iteration. Consider preallocating for speed.
L 23 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
L 27 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 33 (C 13-16): The variable 'data' appears to change size on every loop iteration. Consider preallocating for speed.
L 33 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
L 37 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 38 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Consider preallocating for speed.
L 44 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Consider preallocating for speed.
L 47 (C 52): Invalid syntax at ';'. Possibly, a ), }, or ] is missing.
L 47 (C 53): Invalid syntax at ')'. Possibly, a ), }, or ] is missing.
L 47 (C 54): Parse error at ']': usage might be invalid MATLAB syntax.
L 48 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 48 (C 23): Use of brackets [] is unnecessary. Use parentheses to group, if needed.
```

For details about these messages and how to improve the code, see “Changing Code Based on Code Analyzer Messages” in the MATLAB Desktop Tools and Development Environment documentation.

Running checkcode with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=checkcode('lengthofline','-id')
```

MATLAB returns

```
inform =

17x1 struct array with fields:

    id
    message
    fix
    line
    column
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans =

         id: 'NASGU'
    message: 'The value assigned to variable 'nothandle' might be unused.'
        fix: 0
       line: 21
     column: [1 9]
```

Here, the message is for the value that appears on line 21 that extends from column 1–9 in the file.NASGU is the ID for the message `'The value assigned to variable 'nothandle' might be unused.'`.

Displaying McCabe Complexity with checkcode

To display the McCabe complexity of a MATLAB code file, run `checkcode` with the `-cyc` option, as shown in the following example (assuming you have saved `lengthofline.m` to a local folder).

```
checkcode lengthofline.m -cyc
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the Code Analyzer messages, as shown here:

```
L 1 (C 23-34): The McCabe complexity of 'lengthofline' is 12.
L 21 (C 1-9): The value assigned to variable 'nothandle' might be unused.
L 22 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 23 (C 5-11): The variable 'notline' appears to change size on every loop iteration. Consider preallocating for speed.
L 23 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
L 27 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 33 (C 13-16): The variable 'data' appears to change size on every loop iteration. Consider preallocating for speed.
L 33 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
L 37 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 38 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Consider preallocating for speed.
L 44 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Consider preallocating for speed.
L 47 (C 52): Invalid syntax at ';'. Possibly, a ), }, or ] is missing.
L 47 (C 53): Invalid syntax at ')'. Possibly, a ), }, or ] is missing.
L 47 (C 54): Parse error at ']': usage might be invalid MATLAB syntax.
L 48 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 48 (C 23): Use of brackets [] is unnecessary. Use parentheses to group, if needed.
```

See Also

`mlintrpt`, `profile`

How To

- For information on the suppression directive, `%#ok`, and suppressing messages from within your program, see “Adjust Code Analyzer Message Indicators and Messages ”.

checkin

(To be removed) Check files into source control system (UNIX platforms)

Note: `checkin` will be removed in a future release.

Syntax

```
checkin('filename','comments','comment_text')
checkin({'filename1','filename2'},'comments','comment_text')
checkin('filename','comments', 'comment_text','option','value')
```

Description

`checkin('filename','comments','comment_text')` checks in the file named `filename` to the source control system. Use the full path for `filename` and include the file extension. You must save the file before checking it in, but the file can be open or closed. The `comment_text` is a MATLAB string containing checkin comments for the source control system. You must supply **`comments`** and `comment_text`.

`checkin({'filename1','filename2'},'comments','comment_text')` checks in the files `filename1` through `filenamen` to the source control system. Use the full paths for the files and include file extensions. Comments apply to all files checked in.

`checkin('filename','comments', 'comment_text','option','value')` provides additional `checkin` options. For multiple file names, use an array of strings instead of `filename`, that is, `{'filename1','filename2',...}`. Options apply to all file names. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	`filename` is checked in even if the file has not changed since it was checked out.
`'force'`	`'off'` (default)	`filename` is not checked in if there were no changes since checkout.

option Argument	value Argument	Purpose
`'lock'`	`'on'`	`filename` is checked in with comments, and is automatically checked out.
`'lock'`	`'off'` (default)	`filename` is checked in with comments but does not remain checked out.

Examples

Check In a File

Check the file `/myserver/myfiles/clock.m` into the source control system, with the comment `Adjustment for leapyear`:

```
checkin('/myserver/myfiles/clock.m','comments',...
'Adjustment for leapyear')
```

Check In Multiple Files

Check two files into the source control system, using the same comment for each:

```
checkin({'/myserver/myfiles/clock.m', ...
'/myserver/myfiles/calendar.m'},'comments',...
'Adjustment for leapyear')
```

Check In a File and Keep It Checked Out

Check the file `/myserver/myfiles/clock.m` into the source control system and keep the file checked out:

```
checkin('/myserver/myfiles/clock.m','comments',...
'Adjustment for leapyear','lock','on')
```

Introduced before R2006a

checkout

(To be removed) Check files out of source control system (UNIX platforms)

Note: `checkout` will be removed in a future release.

Syntax

```
checkout('filename')
checkout({'filename1','filename2', ...})
checkout('filename','option','value',...)
```

Description

`checkout('filename')` checks out the file named `filename` from the source control system. Use the full path for `filename` and include the file extension. The file can be open or closed when you use `checkout`.

`checkout({'filename1','filename2', ...})` checks out the files named `filename1` through `filenamen` from the source control system. Use the full paths for the files and include the file extensions.

`checkout('filename','option','value',...)` provides additional `checkout` options. For multiple file names, use an array of strings instead of `filename`, that is, `{'filename1','filename2', ...}`. Options apply to all file names. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	The checkout is forced, even if you already have the file checked out. This is effectively an `undocheckout` followed by a `checkout`.

option Argument	value Argument	Purpose
`'force'`	`'off'` (default)	Prevents you from checking out the file if you already have it checked out.
`'lock'`	`'on'` (default)	The checkout gets the file, allows you to write to it, and locks the file so that access to the file for others is read only.
`'lock'`	`'off'`	The checkout gets a read-only version of the file, allowing another user to check out the file for updating. You do not have to check the file in after checking it out with this option.
`'revision'`	`'version_num'`	Checks out the specified revision of the file.

If you end the MATLAB session, the file remains checked out. You can check in the file from within the MATLAB desktop during a later session, or directly from your source control system.

Examples

Check Out a File

Check out the file `/myserver/myfiles/clock.m` from the source control system:

```
checkout('/myserver/myfiles/clock.m')
```

Check Out Multiple Files

Check out `/matlab/myfiles/clock.m` and `/matlab/myfiles/calendar.m` from the source control system:

```
checkout({'/myserver/myfiles/clock.m',...
'/myserver/myfiles/calendar.m'})
```

Force a Checkout, Even If File Is Already Checked Out

Check out `/matlab/myfiles/clock.m` even if `clock.m` is already checked out to you:

```
checkout('/myserver/myfiles/clock.m','force','on')
```

Check Out Specified Revision of File

Check out revision `1.1` of `clock.m`:

```
checkout('/matlab/myfiles/clock.m','revision','1.1')
```

Introduced before R2006a

chol

Cholesky factorization

Syntax

```
R = chol(A)
L = chol(A,'lower')
R = chol(A,'upper')
[R,p] = chol(A)
[L,p] = chol(A,'lower')
[R,p] = chol(A,'upper')
[R,p,S] = chol(A)
[R,p,s] = chol(A,'vector')
[L,p,s] = chol(A,'lower','vector')
[R,p,s] = chol(A,'upper','vector')
```

Description

`R = chol(A)` produces an upper triangular matrix `R` from the diagonal and upper triangle of matrix `A`, satisfying the equation `R'*R=A`. The `chol` function assumes that `A` is (complex Hermitian) symmetric. If it is not, `chol` uses the (complex conjugate) transpose of the upper triangle as the lower triangle. Matrix `A` must be positive definite.

`L = chol(A,'lower')` produces a lower triangular matrix `L` from the diagonal and lower triangle of matrix `A`, satisfying the equation `L*L'=A`. The `chol` function assumes that `A` is (complex Hermitian) symmetric. If it is not, `chol` uses the (complex conjugate) transpose of the lower triangle as the upper triangle. When `A` is sparse, this syntax of `chol` is typically faster. Matrix `A` must be positive definite. `R = chol(A,'upper')` is the same as `R = chol(A)`.

`[R,p] = chol(A)` for positive definite `A`, produces an upper triangular matrix `R` from the diagonal and upper triangle of matrix `A`, satisfying the equation `R'*R=A` and `p` is zero. If `A` is not positive definite, then `p` is a positive integer and MATLAB does not generate an error. When `A` is full, `R` is an upper triangular matrix of order `q=p-1` such that `R'*R=A(1:q,1:q)`. When `A` is sparse, `R` is an upper triangular matrix of size `q`-by-`n`

so that the L-shaped region of the first q rows and first q columns of R'*R agree with those of A.

[L,p] = chol(A,'lower') for positive definite A, produces a lower triangular matrix L from the diagonal and lower triangle of matrix A, satisfying the equation L*L'=A and p is zero. If A is not positive definite, then p is a positive integer and MATLAB does not generate an error. When A is full, L is a lower triangular matrix of order q=p-1 such that L*L'=A(1:q,1:q). When A is sparse, L is a lower triangular matrix of size q-by-n so that the L-shaped region of the first q rows and first q columns of L*L' agree with those of A. [R,p] = chol(A,'upper') is the same as [R,p] = chol(A).

The following three-output syntaxes require sparse input A.

[R,p,S] = chol(A), when A is sparse, returns a permutation matrix S. Note that the preordering S may differ from that obtained from amd since chol will slightly change the ordering for increased performance. When p=0, R is an upper triangular matrix such that R'*R=S'*A*S. When p is not zero, R is an upper triangular matrix of size q-by-n so that the L-shaped region of the first q rows and first q columns of R'*R agree with those of S'*A*S. The factor of S'*A*S tends to be sparser than the factor of A.

[R,p,s] = chol(A,'vector'), when A is sparse, returns the permutation information as a vector s such that A(s,s)=R'*R, when p=0. You can use the 'matrix' option in place of 'vector' to obtain the default behavior.

[L,p,s] = chol(A,'lower','vector'), when A is sparse, uses only the diagonal and the lower triangle of A and returns a lower triangular matrix L and a permutation vector s such that A(s,s)=L*L', when p=0. As above, you can use the 'matrix' option in place of 'vector' to obtain a permutation matrix. [R,p,s] = chol(A,'upper','vector') is the same as [R,p,s] = chol(A,'vector').

Note Using chol is preferable to using eig for determining positive definiteness.

Examples

Example 1

The gallery function provides several symmetric, positive, definite matrices.

```
A=gallery('moler',5)
```

```
A =

     1    -1    -1    -1    -1
    -1     2     0     0     0
    -1     0     3     1     1
    -1     0     1     4     2
    -1     0     1     2     5

C=chol(A)

ans =

     1    -1    -1    -1    -1
     0     1    -1    -1    -1
     0     0     1    -1    -1
     0     0     0     1    -1
     0     0     0     0     1
isequal(C'*C,A)

ans =

     1
```

For sparse input matrices, `chol` returns the Cholesky factor.

```
N = 100;
A = gallery('poisson', N);
```

`N` represents the number of grid points in one direction of a square `N`-by-`N` grid. Therefore, `A` is N^2 by N^2.

```
L = chol(A, 'lower');
D = norm(A - L*L', 'fro');
```

The value of `D` will vary somewhat among different versions of MATLAB but will be on order of 10^{-14}.

Example 2

The binomial coefficients arranged in a symmetric array create a positive definite matrix.

```
n = 5;
```

```
X = pascal(n)
X =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70
```

This matrix is interesting because its Cholesky factor consists of the same coefficients, arranged in an upper triangular matrix.

```
R = chol(X)
R =
     1     1     1     1     1
     0     1     2     3     4
     0     0     1     3     6
     0     0     0     1     4
     0     0     0     0     1
```

Destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element.

```
X(n,n) = X(n,n)-1

X =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    69
```

Now an attempt to find the Cholesky factorization of `X` fails.

```
chol(X)
Error using chol
Matrix must be positive definite.
```

See Also

`ichol` | `cholupdate`

Introduced before R2006a

cholupdate

Rank 1 update to Cholesky factorization

Syntax

```
R1 = cholupdate(R,x)
R1 = cholupdate(R,x,'+')
R1 = cholupdate(R,x,'-')
[R1,p] = cholupdate(R,x,'-')
```

Description

`R1 = cholupdate(R,x)` where `R = chol(A)` is the original Cholesky factorization of `A`, returns the upper triangular Cholesky factor of `A + x*x'`, where `x` is a column vector of appropriate length. `cholupdate` uses only the diagonal and upper triangle of `R`. The lower triangle of `R` is ignored.

`R1 = cholupdate(R,x,'+')` is the same as `R1 = cholupdate(R,x)`.

`R1 = cholupdate(R,x,'-')` returns the Cholesky factor of `A - x*x'`. An error message reports when R is not a valid Cholesky factor or when the downdated matrix is not positive definite and so does not have a Cholesky factorization.

`[R1,p] = cholupdate(R,x,'-')` will not return an error message. If `p` is `0`, `R1` is the Cholesky factor of `A - x*x'`. If `p` is greater than `0`, `R1` is the Cholesky factor of the original `A`. If `p` is `1`, `cholupdate` failed because the downdated matrix is not positive definite. If `p` is `2`, `cholupdate` failed because the upper triangle of `R` was not a valid Cholesky factor.

Examples

```
A = pascal(4)
A =

     1     1     1     1
     1     2     3     4
```

```
     1     3     6    10
     1     4    10    20

R = chol(A)
R =

     1     1     1     1
     0     1     2     3
     0     0     1     3
     0     0     0     1
x = [0 0 0 1]';
```

This is called a rank one update to `A` since `rank(x*x')` is `1`:

```
A + x*x'
ans =


     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    21
```

Instead of computing the Cholesky factor with `R1 = chol(A + x*x')`, we can use `cholupdate`:

```
R1 = cholupdate(R,x)
R1 =

    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    1.4142
```

Next destroy the positive definiteness (and actually make the matrix singular) by subtracting `1` from the last element of `A`. The downdated matrix is:

```
A - x*x'
ans =

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    19
```

Compare `chol` with `cholupdate`:

```
R1 = chol(A-x*x')
Error using chol
Matrix must be positive definite.
R1 = cholupdate(R,x,'-')
Error using cholupdate
Downdated matrix must be positive definite.
```

However, subtracting `0.5` from the last element of `A` produces a positive definite matrix, and we can use `cholupdate` to compute its Cholesky factor:

```
x = [0 0 0 1/sqrt(2)]';
R1 = cholupdate(R,x,'-')
R1 =
    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    0.7071
```

More About

Tips

`cholupdate` works only for full matrices.

Algorithms

`cholupdate` uses the algorithms from the LINPACK subroutines `ZCHUD` and `ZCHDD`. `cholupdate` is useful since computing the new Cholesky factor from scratch is an $O(N^3)$ algorithm, while simply updating the existing factor in this way is an $O(N^2)$ algorithm.

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

See Also

`chol` | `qrupdate`

Introduced before R2006a

circshift

Shift array circularly

Compatibility

The default behavior of `circshift(A,K)`, where `K` is a scalar, will change in a future release. The new default behavior will be to operate along the first array dimension of `A` whose size does not equal 1. Use `circshift(A,[K 0])` to retain current behavior.

Syntax

```
Y = circshift(A,K)
Y = circshift(A,K,dim)
```

Description

`Y = circshift(A,K)` circularly shifts the elements in array `A` by `K` positions. Specify `K` as an integer to shift the rows of `A`, or as a vector of integers to specify the shift amount in each dimension.

`Y = circshift(A,K,dim)` circularly shifts the values in array `A` by `K` positions along dimension `dim`. Inputs `K` and `dim` must be scalars.

Examples

Shift Column Vector Elements

Create a numeric column vector.

```
A = (1:10)'
```

```
A =

     1
     2
```

```
     3
     4
     5
     6
     7
     8
     9
    10
```

Use `circshift` to shift the elements by three positions.

```
Y = circshift(A,3)
```

```
Y =

     8
     9
    10
     1
     2
     3
     4
     5
     6
     7
```

The result, `Y`, has the same elements as `A` but they are in a different order.

Move Characters in Array

Create an array of characters.

```
A = 'racecar'
```

```
A =

racecar
```

Use `circshift` to shift the characters by three positions in the second dimension.

```
Y = circshift(A,3,2)
```

```
Y =

carrace
```

The characters are in a different order in Y.

Move Matrix Elements

Create a numeric array with a cluster of ones in the top left.

```
A = [1 1 0 0; 1 1 0 0; 0 0 0 0; 0 0 0 0]
```

```
A =

     1     1     0     0
     1     1     0     0
     0     0     0     0
     0     0     0     0
```

Use `circshift` to shift the elements of A by one position in each dimension.

```
Y = circshift(A,[1 1])
```

```
Y =

     0     0     0     0
     0     1     1     0
     0     1     1     0
     0     0     0     0
```

The cluster of ones is now in the center of the matrix.

To move the cluster back to its original position, use `circshift` on Y with negative shift values.

```
X = circshift(Y,[-1 -1])
```

```
X =

     1     1     0     0
     1     1     0     0
     0     0     0     0
     0     0     0     0
```

The matrix X is equivalent to A.

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`
Complex Number Support: Yes

K — Shift amount

integer scalar | vector of integers

Shift amount, specified as an integer scalar or vector of integers. If the shift amount is larger than the length of the corresponding dimension in A, then the shift circularly wraps to the beginning of that dimension. For example, shifting a 3-element vector by +3 positions will bring its elements back to their original positions.

- If you specify K as an integer and do not specify `dim`, then `circshift` shifts the rows of A down (positive integer) or up (negative integer).
- If you specify K as a vector of integers, each element specifies the shift amount for the *N*th dimension in A. If the *N*th element in K is positive, then the values of A are shifted towards the end (positive integer) or beginning (negative integer) of the *N*th dimension.

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is `dim = 1`. If you specify `dim`, then K must be an integer scalar. Specify `dim = 1` to exchange rows, `dim = 2` to exchange columns, and so on.

See Also

`fftshift` | `permute` | `reshape` | `shiftdim`

Introduced before R2006a

circumcenters

Class: TriRep

(Will be removed) Circumcenters of specified simplices

Note: `circumcenters(TriRep)` will be removed in a future release. Use `circumcenter(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
CC = circumcenters(TR, SI)
[CC RCC] = circumcenters(TR, SI)
```

Description

`CC = circumcenters(TR, SI)` returns the coordinates of the circumcenter of each specified simplex `SI`. `CC` is an `m`-by-`n` matrix, where `m` is of length `length(SI)`, the number of specified simplices, and `n` is the dimension of the space where the triangulation resides.

`[CC RCC] = circumcenters(TR, SI)` returns the circumcenters and the corresponding radii of the circumscribed circles or spheres.

Input Arguments

`TR`	Triangulation object.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the circumcenter information

for the entire triangulation is returned, where the circumcenter associated with simplex `i` is the `i`'th row of `CC`.

Output Arguments

`CC`	`m`-by-`n` matrix. `m` is the number of specified simplices and `n` is the dimension of the space where the triangulation resides. Each row `CC(i,:)` represents the coordinates of the circumcenter of simplex `SI(i)`.
`RCC`	Vector of length `length(SI)`, the number of specified simplices containing radii of the circumscribed circles or spheres.

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Examples

Example 1

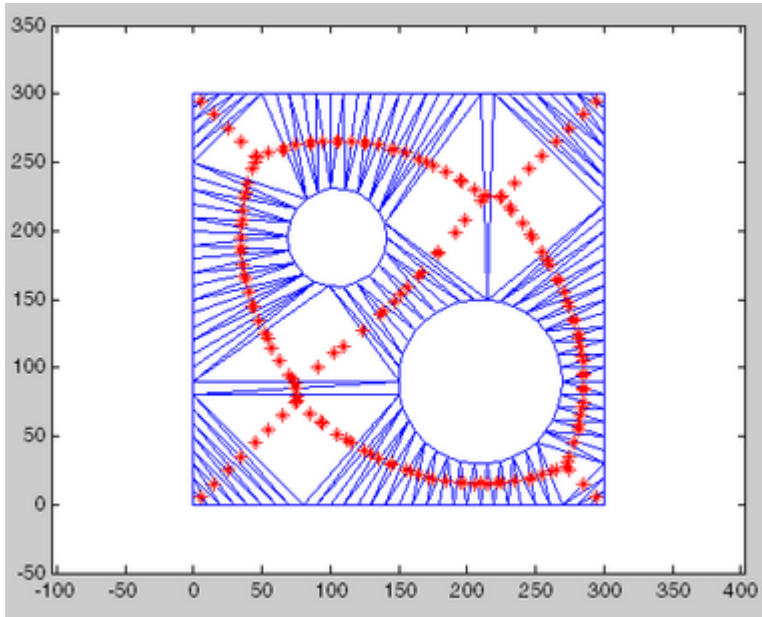
Load a 2-D triangulation.

```
load trimesh2d
trep = TriRep(tri, x,y)
```

Compute the circumcenters.

```
cc = circumcenters(trep);
triplot(trep);
axis([-50 350 -50 350]);
axis equal;
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;
```

The circumcenters represent points on the medial axis of the polygon.



Example 2

Query a 3-D triangulation created with `DelaunayTri`. Compute the circumcenters of the first five tetrahedra.

```
X = rand(10,3);
dt = DelaunayTri(X);
cc = circumcenters(dt, [1:5]')
```

See Also

`delaunayTriangulation` | `incenter` | `triangulation`

cla

Clear axes

Syntax

```
cla
cla reset

cla(ax)
cla(ax,'reset')
```

Description

`cla` deletes all graphics objects that have visible handles from the current axes. The handle is visible if the `HandleVisibility` property of the object is set to `'on'`. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyleOrder` properties of the axes.

`cla reset` deletes graphics objects from the current axes regardless of their handle visibility. It also resets axes properties to their default values, with the exception of the `Position` and `Units` properties.

`cla(ax)` deletes graphics objects from the axes specified by `ax` instead of the current axes.

`cla(ax,'reset')` resets properties for the specified axes.

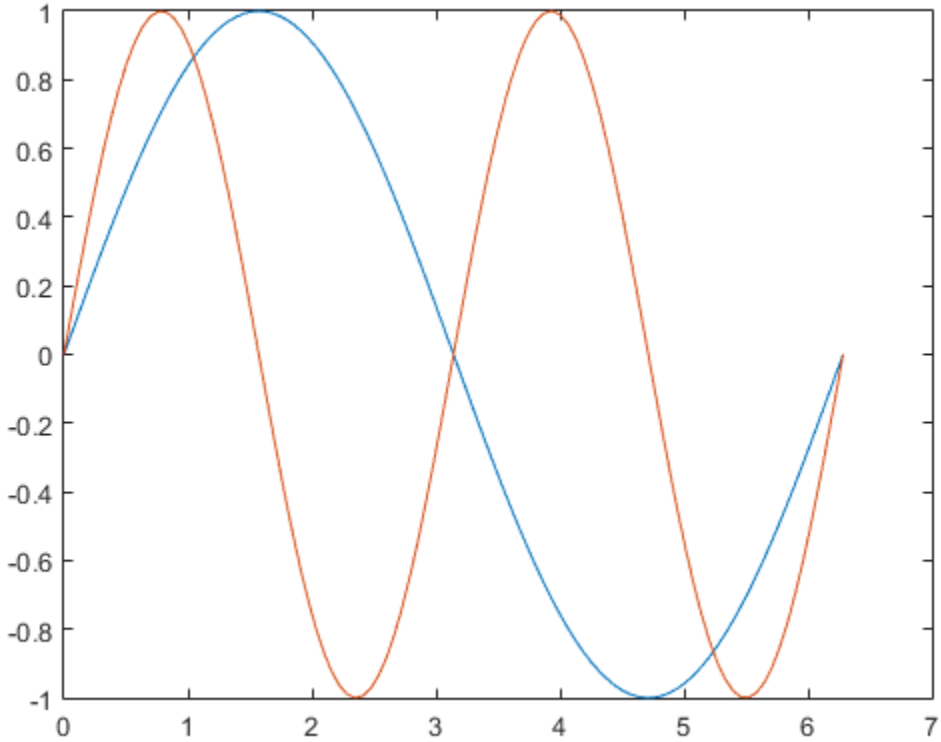
Examples

Clear Current Axes

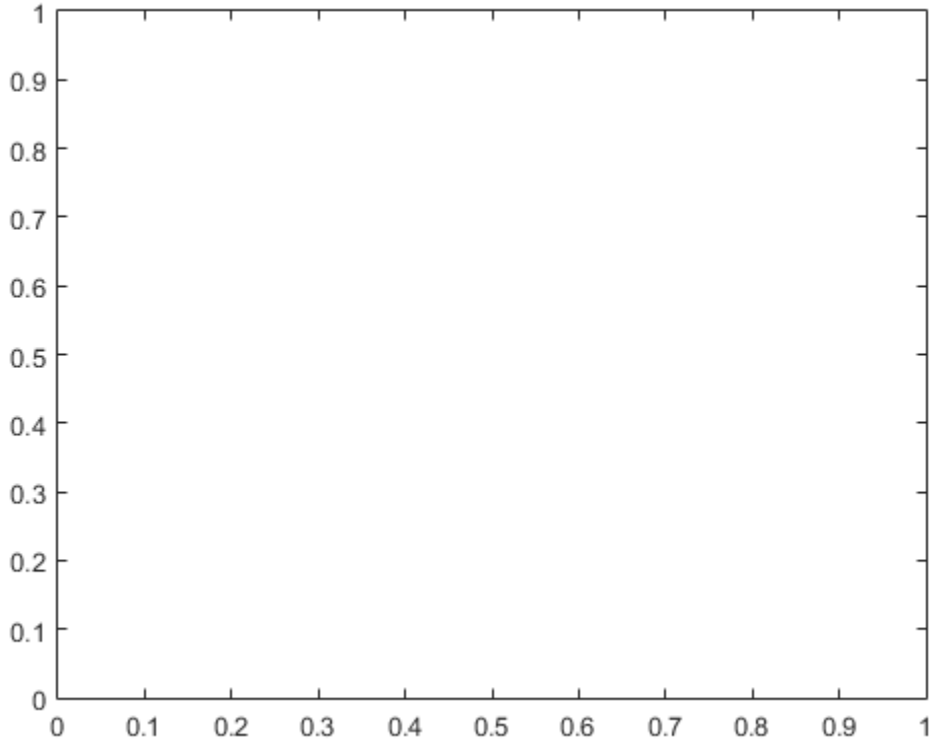
Plot two sine waves. Then, clear the line plots from the axes.

```
x = linspace(0,2*pi);
```

```
y1 = sin(x);
y2 = sin(2*x);

plot(x,y1)
hold on
plot(x,y2)
```

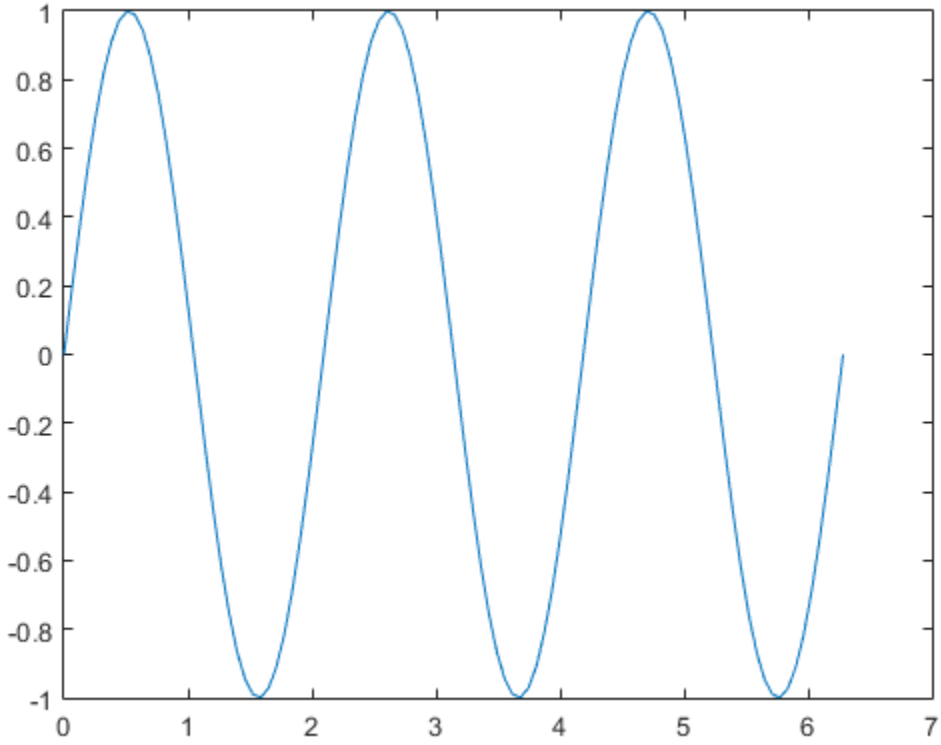



```
cla
```



`cla` clears the line plots and resets the `ColorIndex` and `LineStyleIndex` properties of the axes to 1. Subsequent plots start from the beginning of the color order and line style order. For example, plot another sine wave.

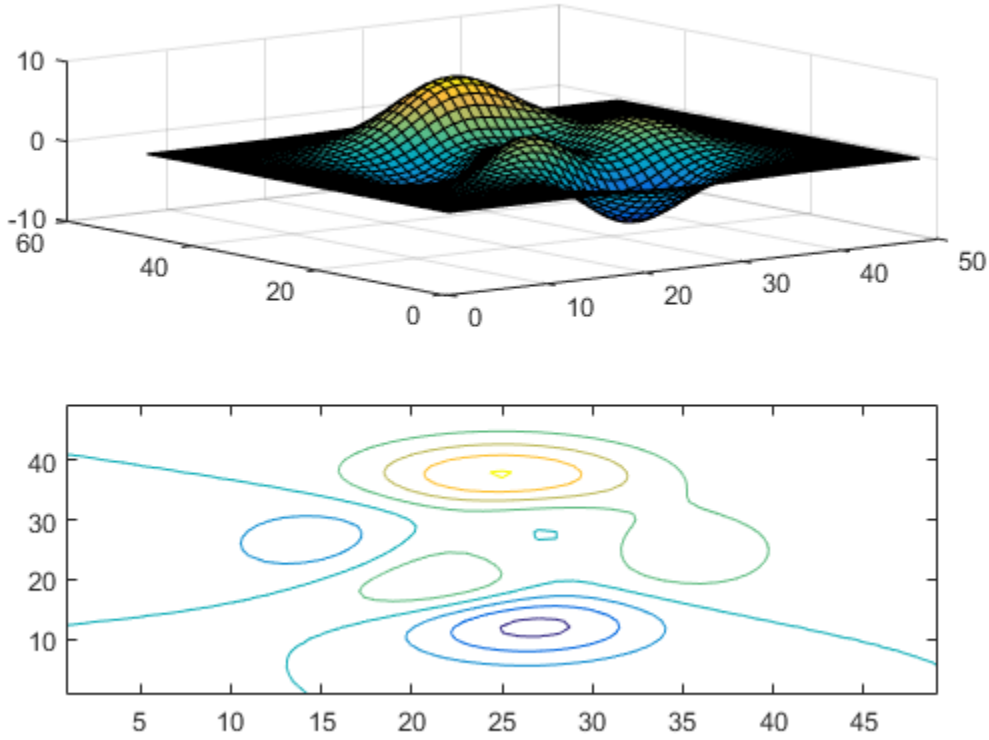
```
y3 = sin(3*x);
plot(x,y3)
hold off
```



Clear Objects from Specific Axes

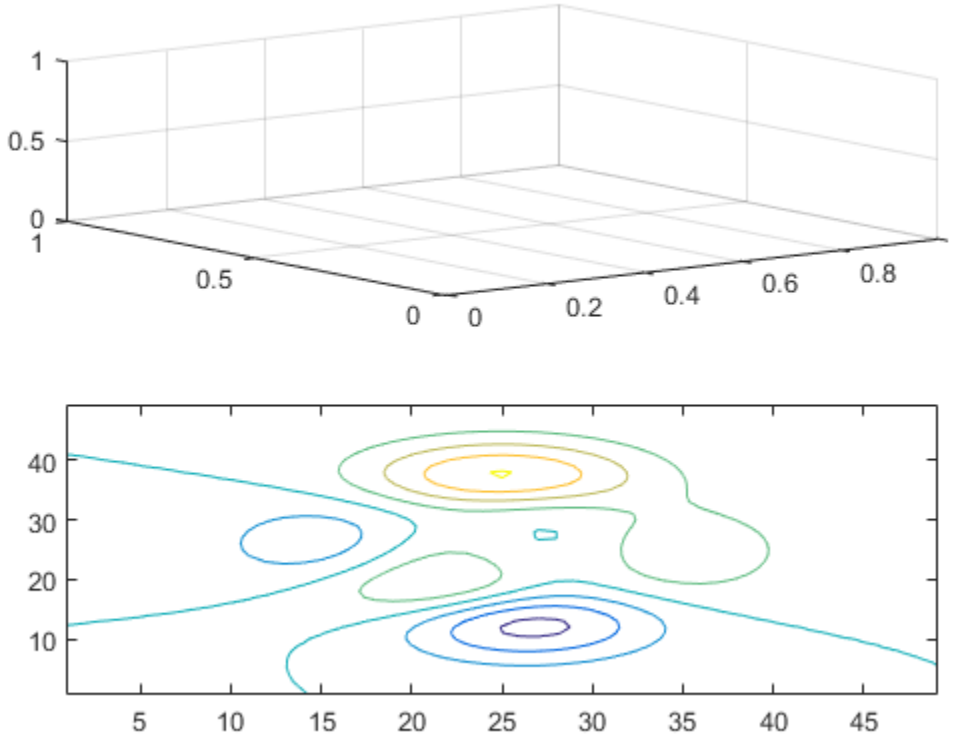
Create a figure with two subplots and add plots to both axes.

```
ax1 = subplot(2,1,1);
surf(peaks)

ax2 = subplot(2,1,2);
contour(peaks)
```



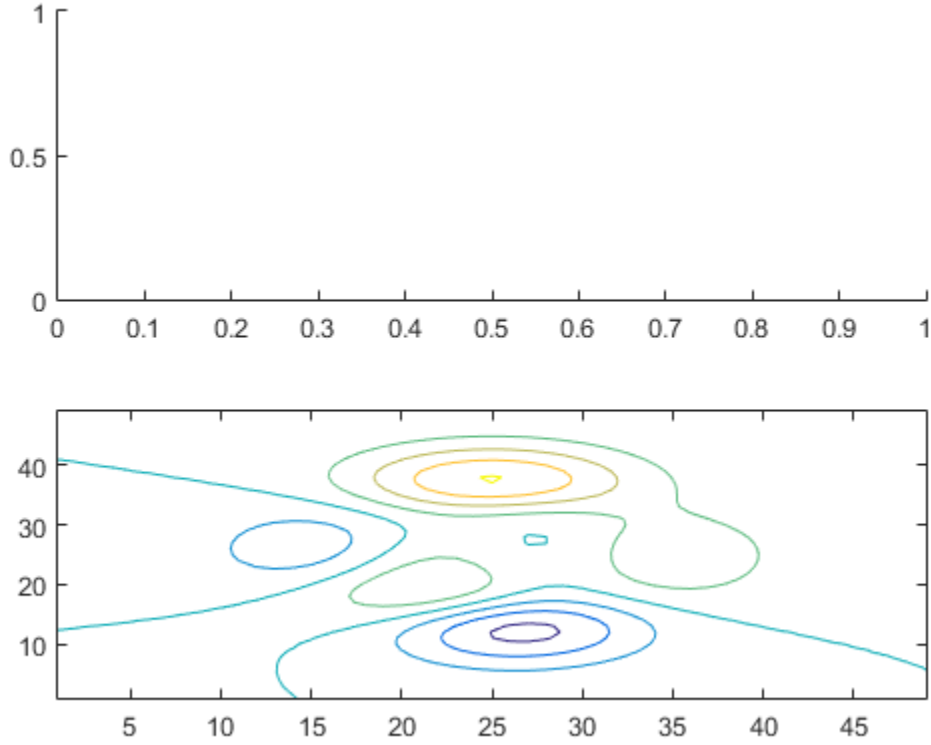
Clear the surface plot from the upper subplot.

```
cla(ax1)
```



Now, reset all axes properties for the upper subplot, including the camera properties that control the view, by using the optional input argument `'reset'`.

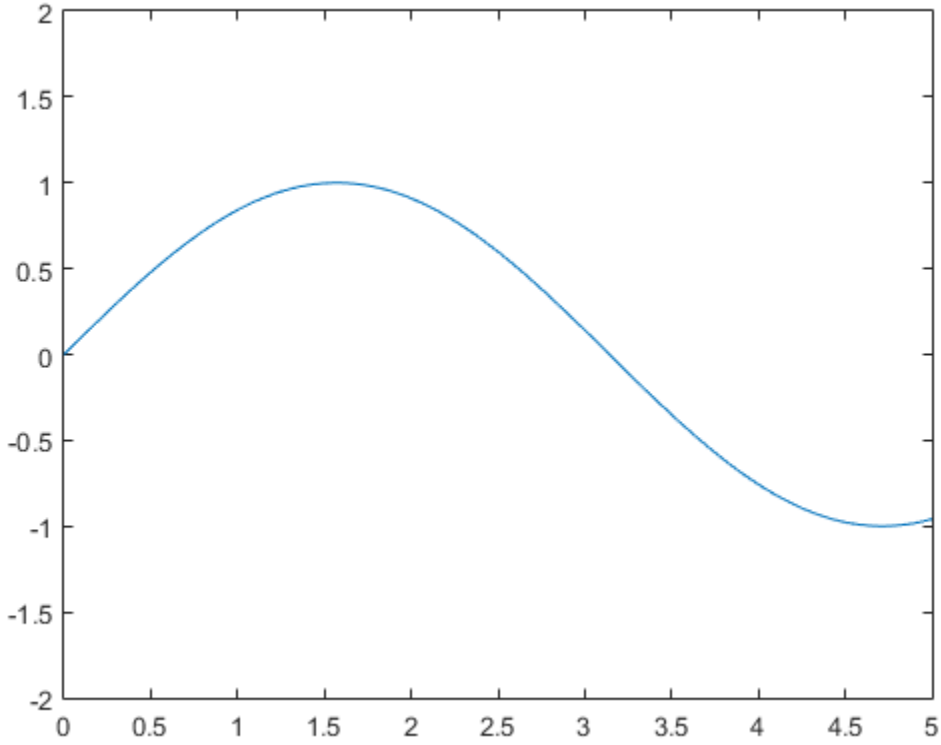
```
cla(ax1,'reset')
```



Clear Axes and Reset All Axes Properties

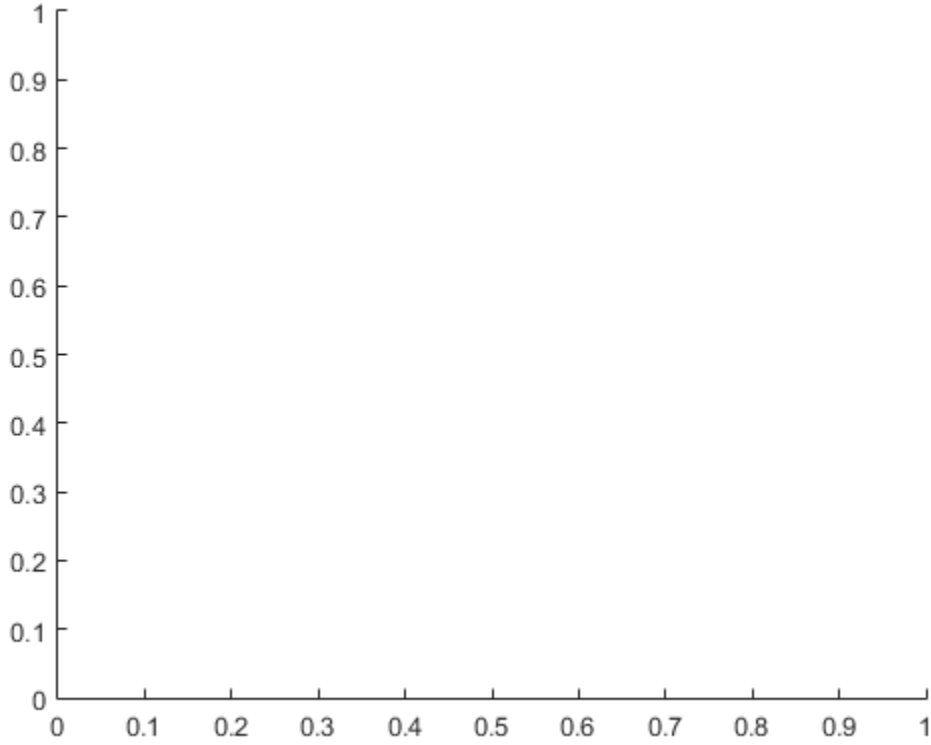
Create a line plot and set the axis limits.

```
x = linspace(0,2*pi);
y = sin(x);

plot(x,y)
axis([0,5,-2,2])
```



Clear the line plot from the axes and reset all the axes properties to their default values. `cla reset` resets all properties of the current axes, except for the `Position` and `Units` properties.

```
cla reset
```



Input Arguments

ax — Axes object

axes object

Axes object. Use `ax` to clear a specific axes, instead of the current axes.

More About

Tips

- If an axes does not exist, then `cla` creates one.
- The `cla` command resets the `ColorOrderIndex` and `LineStyleOrderIndex` properties of the current axes to `1`.
- `cla` only deletes objects with a `HandleVisibility` property set to `'on'`. Therefore, if the `HandleVisibility` of an object is set to `'callback'` and you are in a callback, then `cla` does not delete it.

See Also

Functions
`clf` | `hold` | `newplot` | `reset`

Properties
Axes Properties

Introduced before R2006a

clabel

Contour plot elevation labels

Syntax

```
clabel(C)
clabel(C,v)
clabel(C,'manual')
clabel( ___ ,Name,Value)
tl = clabel( ___ )

clabel(C,h)
clabel(C,h,v)
clabel(C,h,'manual')
clabel( ___ ,'LabelSpacing',space)
```

Description

`clabel(C)` labels all contours displayed in the current contour plot. Labels are upright and displayed with `'+'` symbols. `clabel` randomly selects label positions.

`clabel(C,v)` labels only the contour levels specified by the vector, `v`.

`clabel(C,'manual')` places contour labels at locations you select with a mouse. Click the mouse or press the space bar to label the contour closest to the center of the crosshair. Press the **Return** key while the cursor is within the figure window to terminate labeling.

`clabel( ___ ,Name,Value)` specifies text properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes. For example, `'FontSize',14` sets the font size to 14 points.

`tl = clabel( ___ )` returns the text and line objects created.

`clabel(C,h)` inserts rotated labels into each contour line. The contour line must be long enough to fit the label, otherwise `clabel` does not insert a label. If you do not have the contour matrix `C`, then replace `C` with `[]`.

`clabel(C,h,v)` labels only the contour levels specified by vector `v`.

`clabel(C,h,'manual')` places contour labels at locations you select with a mouse. Press the **Return** key while the cursor is within the figure window to terminate labeling.

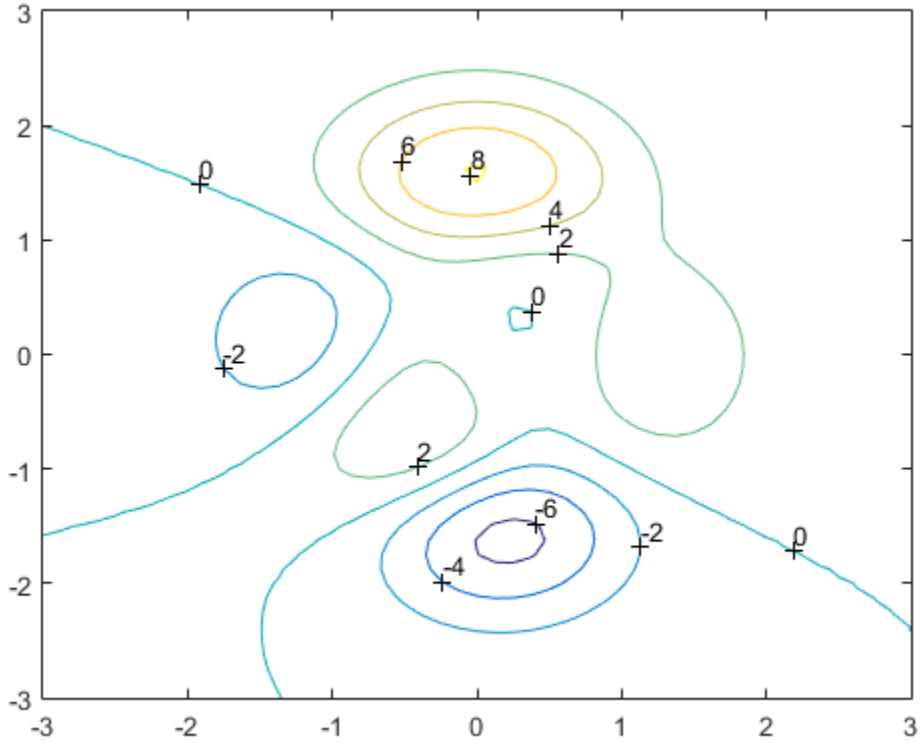
`clabel( ___ ,'LabelSpacing',space)` specifies the spacing between labels as a scalar value. Use this option with any of the input argument combinations that use both the contour matrix `C` and the contour object `h`. For example, `clabel(C,h,'LabelSpacing',72)` spaces the labels 72 points apart (1 inch).

Examples

Label Contour Plot with Vertical Text

Create a contour plot and return the contour matrix, `C`. Then, label the contours.

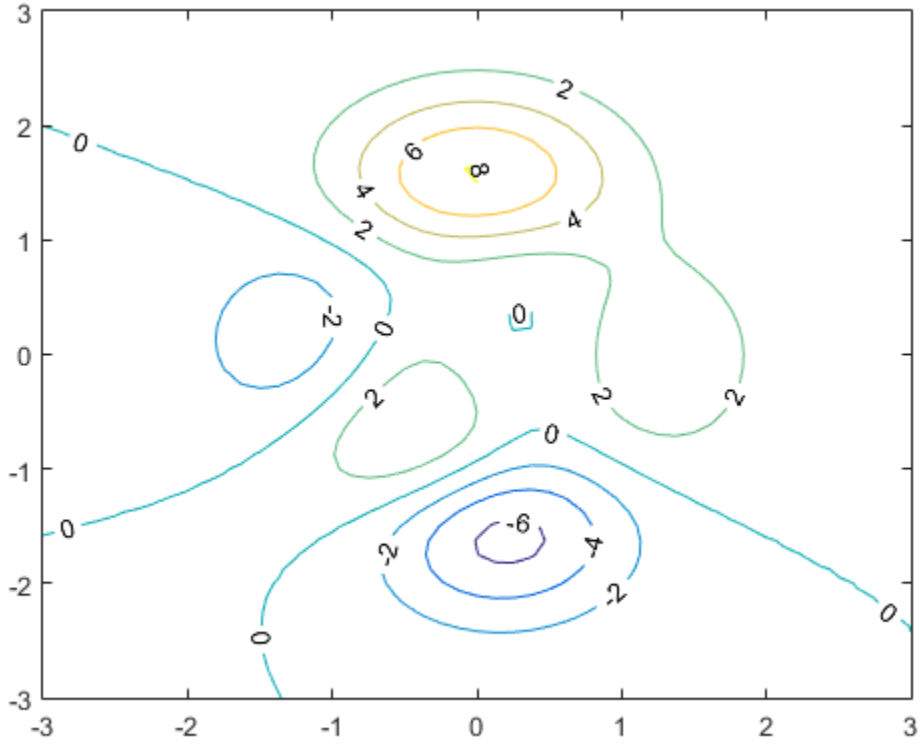
```
[x,y,z] = peaks;
C = contour(x,y,z);
clabel(C)
```



Label Contour Plot with Rotated Text

Create a contour plot and obtain the contour matrix, `C`, and the contour object, `h`. Then, label the contour plot.

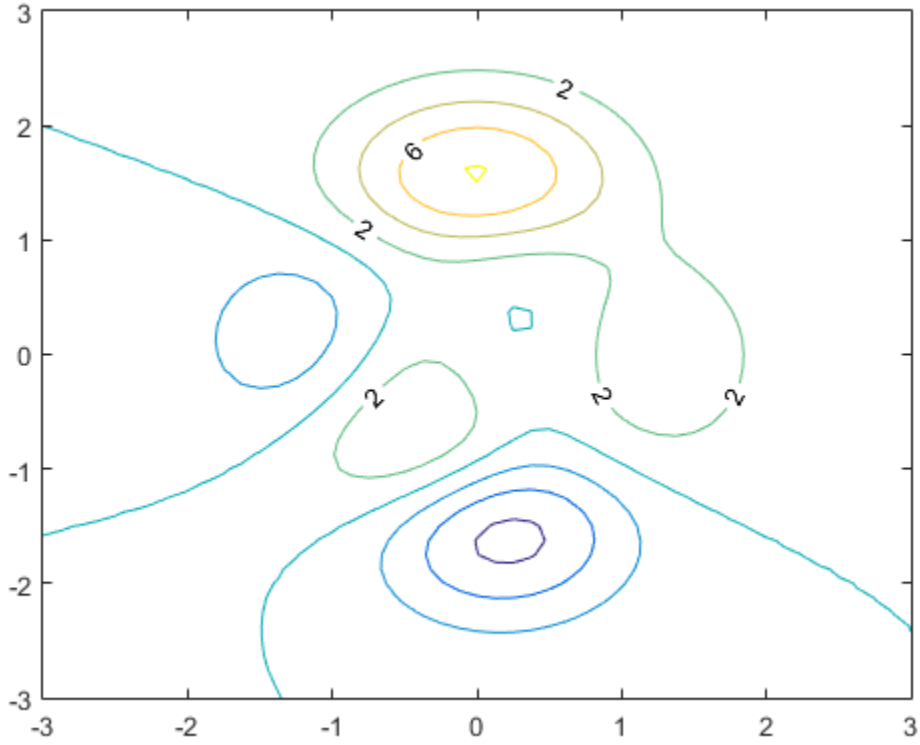
```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h)
```



Label Specific Contour Levels

Label only the contours with contour levels 2 or 6.

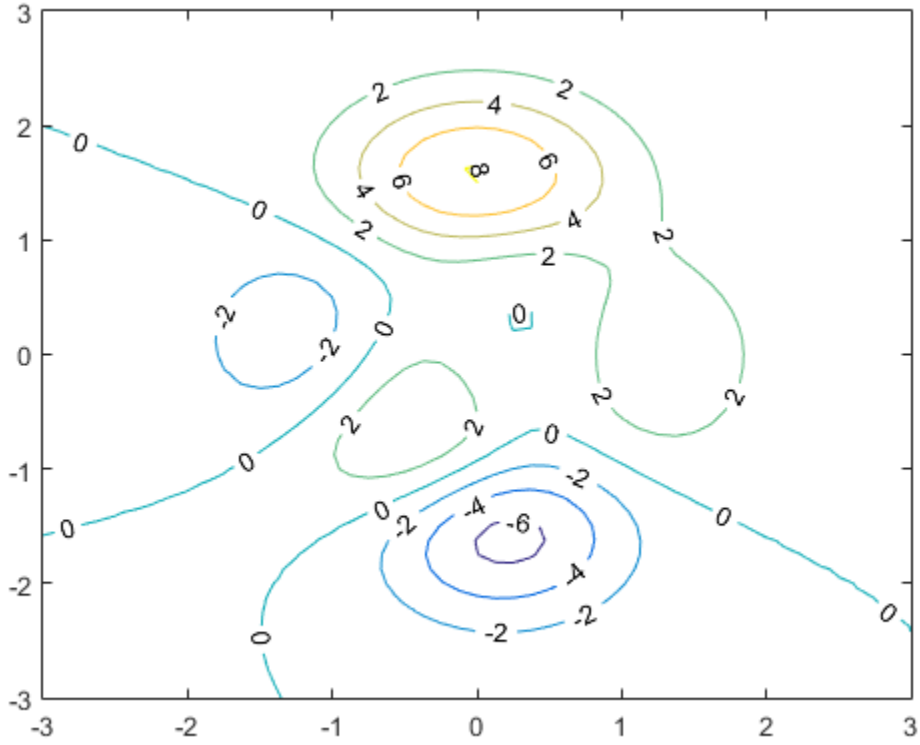
```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
v = [2,6];
clabel(C,h,v)
```



Set Contour Label Spacing

Set the label spacing to 72 points (1 inch).

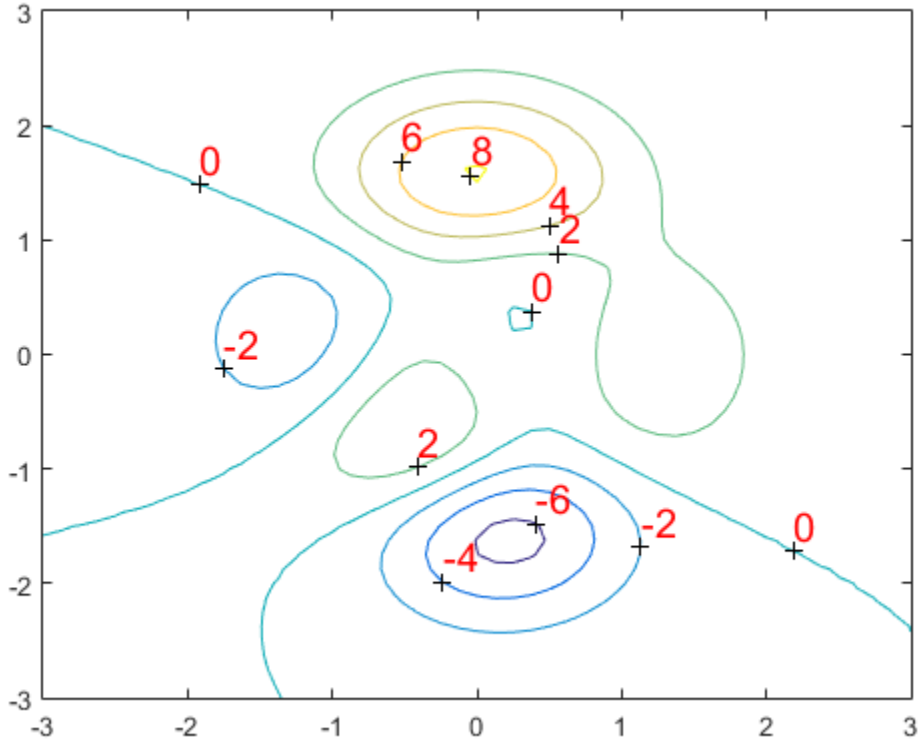
```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'LabelSpacing',72)
```



Set Contour Label Text Properties

Use `Name,Value` arguments to set the font size, font color, and text orientation of the labels.

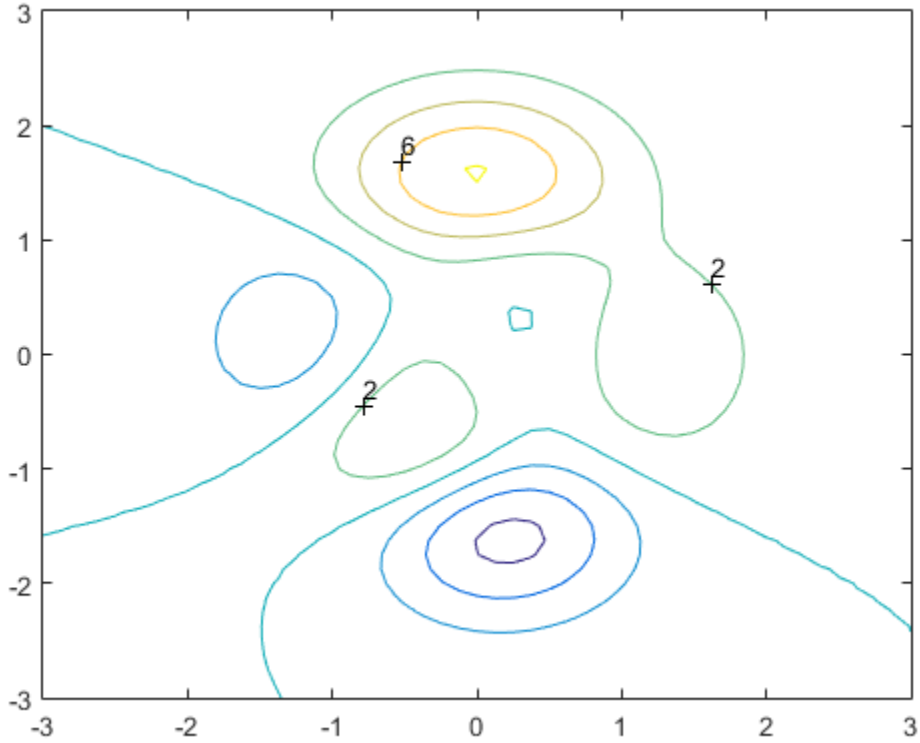
```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,'FontSize',15,'Color','r','Rotation',0)
```



Set Contour Label Text Properties After Creation

Label the contour lines at level 2 and 6. Return the text and line objects created.

```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
tl = clabel(C,[2,6]);
```



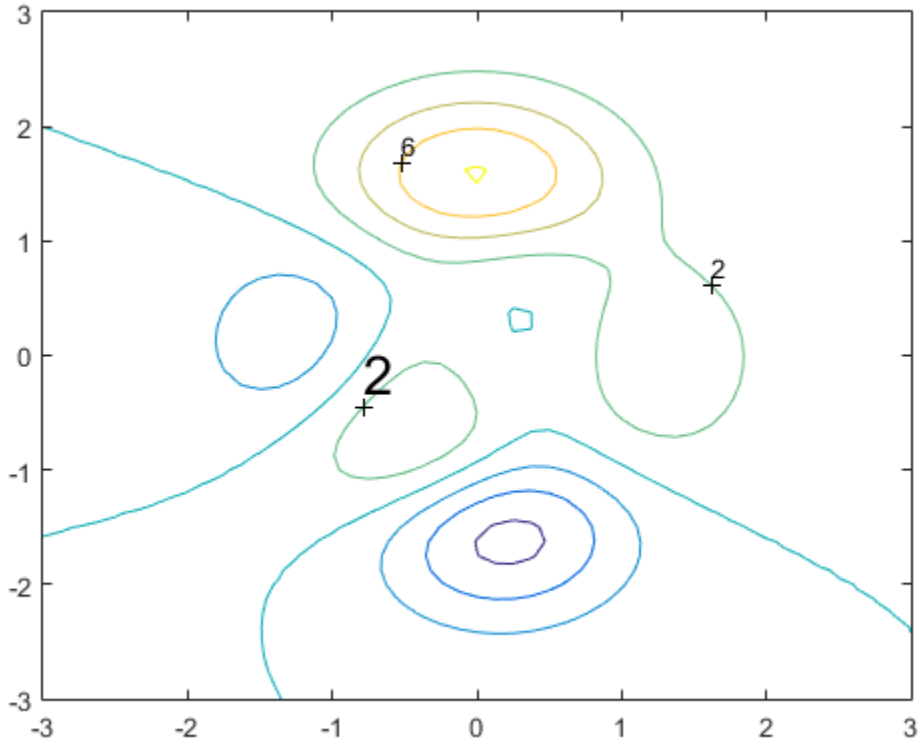
The output `tl` is a vector that contains one text object and one line object for each contour label.

```
disp(tl)
```

```
  6x1 graphics array:

  Line
  Text    (2)
  Line
  Text    (2)
  Line
  Text    (6)
```

Change the font size for one of the labels. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
tl(2).FontSize = 20;
```



Input Arguments

C — Contour matrix

two-row matrix

Contour matrix returned by the `contour`, `contour3`, or `contourf` function. `C` contains the data that defines the contour lines.

Note: If you pass the contour object h to the `clabel` function, then you can replace `C` with `[]`. For example, use `clabel([],h)`.

h — Contour object
contour object

Contour object returned by the `contour`, `contour3`, or `contourf` function.

v — Contour level values
vector

Contour level values, specified as a row or column vector of individual values.

Example: `[0 10 20]`

space — Space between labels
scalar

Space between labels, specified as a scalar in point units.

Example: `72`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','Rotation',45` specifies red labels rotated 45 degrees.

Note: Starting in R2014b, you can no longer set text properties if you pass the contour object `h` as an input argument to the function. For example, `clabel(C,h,'Fontsize',14)` does not set the font size. Use `clabel(C,'FontSize',14)` instead.

The text properties listed here are only a subset, for a complete list, see Text Properties.

'FontSize' — Font size
10 (default) | scalar value greater than 0

Font size, specified as a scalar value greater than 0 in point units. One point equals 1/72 inch. To change the font units, use the FontUnits property.

Example: `12`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`'FontName'` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`'FontWeight'` — Thickness of text characters

`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

`'Color'` — Text color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. The default color is black with an RGB triplet value of `[0 0 0]`. If you set the color to `'none'`, then the text is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

Output Arguments

`tl` — Text and line objects

vector

Text and line objects, returned as a vector. The `String` properties of the text objects contain the contour values displayed. The line objects correspond to the `'+'` symbols.

Note: Starting in R2014b, you can no longer return the output argument `tl` if you pass the contour object `h` as an input argument to the function. For example, `tl = clabel(C,h)` warns and returns empty. Use `tl = clabel(C)` instead.

See Also

Functions
`contour` | `contour3` | `contourc` | `contourf`

Properties
Text Properties

Introduced before R2006a

class

Determine class of object

Syntax

```
ClassName = class(object)
obj = class(s,'class_name')
obj = class(s,'class_name',parent1,parent2,...)
obj = class(struct([]),'class_name',parent1,parent2,...)
obj_struct = class(struct_array,'class_name',parent_array)
```

Description

`ClassName = class(object)` returns a string specifying the class of `object`. See "Fundamental MATLAB Classes" for more information on MATLAB classes.

Note: Before MATLAB 7.6 (classes defined without a `classdef` statement), class constructors called the `class` function to create the object. The following `class` function syntaxes apply only within classes defined before Version 7.6.

`obj = class(s,'class_name')` creates an array of class `class_name` objects using the `struct` `s` as a pattern to determine the size of `obj`.

`obj = class(s,'class_name',parent1,parent2,...)` inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. The size of the parent objects must match the size of `s` or be a scalar (1–by-1), in which case, MATLAB performs scalar expansion.

`obj = class(struct([]),'class_name',parent1,parent2,...)` constructs object containing only fields that it inherits from the parent objects. All parents must have the same, nonzero size, which determines the size of the returned object `obj`.

`obj_struct = class(struct_array,'class_name',parent_array)` maps every element of the `parent_array` to a corresponding element in the `struct_array` to produce the output array of objects, `obj_struct`.

All arrays must be of the same size. If either the `struct_array` or the `parent_array` is of size 1–by–1, then MATLAB performs scalar expansion to match the array sizes.

To create an object array of size 0–by-0, set the size of the `struct_array` and `parent_array` to 0–by-0.

Examples

Return the class of Java object `obj`:

```
import java.lang.*;
obj = String('mystring');
class(obj)

ans =

java.lang.String
```

Return class of any MATLAB variable:

```
h = @sin;
class(h)

ans =

function_handle
```

See Also

`isa` | `isobject` | `metaclass`

Introduced before R2006a

classdef

Class definition keywords

Syntax

```
classdef classname
   properties
      PropName
   end
   methods
      methodName
   end
   events
      EventName
   end
   enumeration
      EnumName
   end
end
```

Description

`classdef` *`classname`* begins the class definition and an `end` keyword terminates the `classdef` block. Only blank lines and comments can precede `classdef`. Enter a class definition in a file having the same name as the class, with a filename extension of `.m`.

Class definition files can be in folders on the MATLAB path or in class folders whose parent folder is on the MATLAB path. Class folder names begin with the `'@'` character followed by the class name (for example, `@MyClass`). For more information on class folders, see “Class Files and Folders” .

For more information on classes, see “Classdef Block” and “Class Definition” .

`properties` begins a property definition block, an `end` keyword terminates the `properties` block. Class definitions can contain multiple property definition blocks, each specifying different attribute settings that apply to the properties in that particular block.

For more information on properties, see “Defining Properties”.

Note: Properties cannot have the same name as the class.

`methods` begins a methods definition block, an `end` keyword terminates the `methods` block. This block contains functions that implement class methods. Class definitions can contain multiple method blocks, each specifying different attribute settings that apply to the methods in that particular block. It is possible to define method functions in separate files.

For more information on methods, see “How to Use Methods”.

`events` begins an events definition block, an `end` keyword terminates the `events` block. This block contains event names defined by the class. Class definitions can contain multiple event blocks, each specifying different attribute settings that apply to the events in that particular block.

For more information on events, see “Events and Listeners — Syntax and Techniques”.

`enumeration` begins an enumeration definition block, an `end` keyword terminates the `enumeration` block.

For more information on enumerations, see “Enumerations”.

`properties`, `methods`, `events`, and `enumeration` are also the names of MATLAB functions used to query the respective class members for a given object or class name.

To see the attributes of all class components in a popup window, click this link: Attribute Tables

Examples

Use these keywords to define classes.

```
classdef (Attributes) ClassName
   properties (Attributes)
      PropertyName
   end
   methods (Attributes)
```

```
      function obj = methodName(obj,arg2,...)
         ...
      end
   end
   events (Attributes)
      EventName
   end
   enumeration
      EnumName
   end
end
```

See Also

properties | methods | events

clc

Clear Command Window

Syntax

```
clc
```

Description

`clc` clears all input and output from the Command Window display, giving you a "clean screen."

After using `clc`, you cannot use the scroll bar to see the history of functions, but you still can use the up arrow key, ↑, to recall statements from the command history.

Examples

Use `clc` in a MATLAB code file to always display output in the same starting position on the screen.

See Also

`clear` | `clf` | `close` | `home`

Introduced before R2006a

clear

Remove items from workspace, freeing up system memory

Syntax

```
clear
clear name1 ... nameN
clear -regexp expr1 ... exprN
clear ItemType
```

Description

`clear` removes all variables from the current workspace, releasing them from system memory.

`clear name1 ... nameN` removes the variables, scripts, functions, or MEX-functions `name1 ... nameN` from memory.

- If `name` is a function name, then `clear name` reinitializes any persistent variables in the function.
- If function `name` is locked by `mlock`, then it remains in memory.
- If variable `name` is global, then `clear` removes it from the current workspace, but it remains in the global workspace.

`clear -regexp expr1 ... exprN` clears all variables that match any of the regular expressions listed. This option only clears variables.

`clear ItemType` clears the types of items indicated by `ItemType`, such as `all`, `functions`, or `classes`.

Examples

Clear a Single Variable

Define two variables `a` and `b`, and then clear `a`.

```
a = 1;
```

```
b = 2;
clear a
whos
```

```
Name      Size            Bytes  Class     Attributes

b         1x1                 8  double
```

Only variable `b` remains in the workspace.

Clear Specific Variables by Name

Using regular expressions, clear those variables with names that begin with `Mon`, `Tue`, or `Wed`.

```
clear -regexp ^Mon ^Tue ^Wed;
```

Clear List of Variables

Create a cell array, `varlist`, that contains the names of variables to clear. Then, clear those variables.

```
varlist = {'v1','v2','time'};
clear(varlist{:})
```

Clear All Compiled Scripts, Functions, and MEX-functions

```
clear functions
```

If a function is locked, it will not be cleared from memory.

Input Arguments

`name1 ... nameN` — Names of variables, scripts, functions, or MEX-functions to clear

string

Names of variables, scripts, functions, or MEX-functions to clear, specified as strings.

Use a partial path to distinguish between different overloaded versions of a function. For example, `clear polynom/display` clears only the `display` method for `polynom` objects, leaving any other implementations in memory.

`expr1 ... exprN` — Regular expressions matching names of variables to clear

string

Regular expressions matching names of variables to clear, specified as strings.

`ItemType` — Type of items to clear
`all` | `classes` | `functions` | `global` | `import` | `java` | `mex` | `variables`

Type of items to clear, specified as one of the following strings. Click the linked strings below for additional information.

Value of `ItemTyp`	Items Cleared								
	Variables in scope	Scripts and function	Class definition	Persistent variables	Debuggir breakpoi	MEX functions	Global variable	Import list	Java classes on the dynamic path
`all`	✓	✓		✓	✓	✓	✓	From comman prompt only	
`classes`	✓	✓	✓	✓	✓	✓	✓		
`functio`		✓		✓	✓	✓			
`global`							✓		
`import`								✓	
`java`	✓	✓		✓	✓	✓	✓		✓
`mex`						✓			
`variabl`	✓								

The following applies to clearing types of items.

`all`:

- `clear all` also removes the import list when issued from the command prompt.

`classes`:

- `clear classes` issues a warning and does not clear a class of objects if any of those objects still exists after the workspace is cleared. For example, objects can still exist in persistent variables of functions or figure windows.

- `clear classes` does not clear a class if its file is locked using the `mlock` command. No warning is issued in this case.

`global`:

- `clear global` removes all global variables in the base and global workspaces. If called from a function, `clear global` also removes all global variables in the function workspace.
- Use `clear global name1 ... nameN` to clear the global variables with the specified names.
- Use `clear global –regexp expr1 ... exprN` to clear all global variables that match any of the regular expressions listed.

`import`:

- Call `clear import` only from the command prompt. Calling `clear import` in a function returns an error.

`java`:

- `clear java` issues a warning and does not remove the Java class definition if any Java objects exist outside the workspace (for example, in user data or persistent variables in a locked code file).
- Issue a `clear java` command after modifying any files on the dynamic Java path.

`mex`:

- `clear mex` does not clear locked MEX functions or functions that are currently in use.

If the name of a variable is a value of `ItemType`, then calling `clear` followed by that name deletes the variable with that name. `clear` does not interpret the name as a keyword in this context. For example, if the workspace contains variables `a`, `all`, `b`, and `ball`, `clear all` removes the variable `all` only.

More About

Tips

- The `clear` function can remove variables that you specify. To remove all but some specified variables, use `clearvars` instead.

- You can clear the handle of a figure or graphics object, but the object itself is not removed. Use `delete` to remove objects. Deleting an object does not delete the variable (if any) used for storing its handle.
- The `clear` function does not clear Simulink® models. Use `bdclose` instead.
- On UNIX systems, `clear` does not affect the amount of memory allocated to the MATLAB process.

- "Base and Function Workspaces"
- "Strategies for Efficient Use of Memory"
- "Automatic Updates for Modified Classes"
- "Java Class Path"
- "Regular Expressions"

See Also

`clc` | `clearvars` | `close` | `delete` | `import` | `inmem` | `load` | `mlock` | `persistent` | `whos` | `workspace`

Introduced before R2006a

clearpoints

Clear points from animated line

To use this function, you must first create an animated line with the `animatedline` function. For more information on line animations, see Using Animated Line Objects.

Syntax

```
clearpoints(h)
```

Description

`clearpoints(h)` clears all points from the animated line specified by `h`. If you want to display the update on the screen, use `drawnow` after using `clearpoints`.

Examples

Clear Points from Animated Line

Create an animated line with 10 points. Then, clear the points stored in the animated line.

```
h = animatedline(1:10,1:10);
clearpoints(h)
```

The animated line still exists, but has no data.

Input Arguments

h — Animated line object

animated line object

Animated line object. Create an animated line object using the `animatedline` function.

See Also

Functions
`addpoints` | `animatedline` | `getpoints`

Using Objects
Using Animated Line Objects

clearvars

Clear variables from memory

Syntax

```
clearvars
clearvars variables
clearvars -except keepVariables
clearvars variables -except keepVariables

clearvars -global ___
```

Description

`clearvars` removes all variables from the currently active workspace.

`clearvars variables` removes the variables specified by `variables`. If any of the variables are global, `clearvars` removes these variables from the current workspace only, leaving them accessible to any functions that declare them as global.

`clearvars -except keepVariables` removes all variables, except for those specified by `keepVariables`. Use this syntax to keep specific variables and remove all others.

`clearvars variables -except keepVariables` removes the variables specified by `variables`, and does not remove the variables specified by `keepVariables`. This syntax allows you to use a combination of variable names, wild card characters, or regular expressions to specify variables to remove or keep.

`clearvars -global ___` removes the specified global variables from the workspace, including those made global within functions, using any of the input arguments in the preceding syntaxes. The `-global` flag must be first in the argument list.

Examples

Clear Named Variables

Define three variables, `a`, `b`, and `c`. Then, clear `a` and `c`.

```
a = 1;
b = 2;
c = 3;
clearvars a c
whos
```

```
  Name      Size            Bytes  Class     Attributes

  b         1x1                 8  double
```

Only variable `b` remains in the workspace.

Clear All Variables Except Specified

Remove all variables from the workspace except for the variables `C` and `D`.

```
clearvars -except C D
```

Clear Variables Using Regular Expressions and Name Variables to Exclude

Clear variables with names that start with `b` and are followed by 3 digits, except for the variable `b106`.

```
clearvars -regexp ^b\d{3}$ -except b106
```

Name Variables to Clear and Preserve Variables Using Regular Expressions

Clear variables with names that start with `a` and do not end with `a`.

```
clearvars a* -except -regexp a$
```

Clear Global Variables Except Specified

Clear all global variables, except those with names that start with `x`.

```
clearvars -global -except x*
```

Clear List of Variables

Clear a list of variables used for intermediate calculations.

Create two variables in the workspace.

```
cashOnHand = 20;
cost = 12.99;
```

Store a list of the names of all the variables currently in the workspace.

```
initialVars = who;
```

Specify or calculate additional variables, `taxRate` and `tax`.

```
taxRate = 0.0625;
tax = round(100*cost*taxRate)/100;
```

Update the initial variables, `cost` and `cashOnHand`.

```
cost = cost + tax;
cashOnHand = cashOnHand - cost;
```

Clear all variables except the initial variables, using the function form of `clearvars`. When using the function form of a syntax, enclose input strings in single quotes, and separate them with commas.

```
clearvars('-except',initialVars{:})
```

`clearvars` clears the variables, `initialVars`, `taxRate`, and `tax`.

Input Arguments

`variables` — Names of variables to remove

strings

Names of variables to remove, specified as one or more strings in one of the following forms.

Form of `Variables` Input	Variables to Remove
`var1 ... varN`	Named variables, specified as individual strings. Use the `'*'` wildcard to match patterns. For example, `clearvars A*` clears all variables in the workspace with names that start with `A`.
`-regexp` *`expr1 ... exprN`*	Variables with names that match the regular expressions, specified as strings. For example, `clearvars -regexp ^Mon ^Tues` clears only the variables in the workspace with names that begin with `Mon` or `Tues`.

`keepVariables` — Names of variables to keep
strings

Names of variables to keep, specified as one or more strings in one of the following forms.

Form of `Variables` Input	Variables to Keep
`var1 ... varN`	Named variables, specified as individual strings. Use the `'*'` wildcard to match patterns. For example, `clearvars -except A*` clears all variables in the workspace, except those with names that start with `A`.
`-regexp` *`expr1 ... exprN`*	Variables with names that match the regular expressions, specified as strings. For example, `clearvars -except -regexp ^Mon ^Tues` clears all the variables in the workspace, except those with names that begin with `Mon` or `Tues`.

More About

- "What Is the MATLAB Workspace?"
- "Regular Expressions"
- "Command vs. Function Syntax"

See Also

`clear` | `exist` | `global` | `persistent` | `save` | `who` | `whos`

clear (serial)

Remove serial port object from MATLAB workspace

Syntax

```
clear obj
```

Description

`clear obj` removes `obj` from the MATLAB workspace, where `obj` is a serial port object or an array of serial port objects.

Examples

This example creates the serial port object `s` on a Windows platform, copies `s` to a new variable `scopy`, and clears `s` from the MATLAB workspace. `s` is then restored to the workspace with `instrfind` and is shown to be identical to `scopy`.

```
s = serial('COM1');
scopy = s;
clear s
s = instrfind;
isequal(scopy,s)
ans =
     1
```

More About

Tips

If `obj` is connected to the device and it is cleared from the workspace, then `obj` remains connected to the device. You can restore `obj` to the workspace with the `instrfind` function. A serial port object connected to the device has a `Status` property value of `open`.

To disconnect `obj` from the device, use the `fclose` function. To remove `obj` from memory, use the `delete` function. You should remove invalid serial port objects from the workspace with `clear`.

See Also

`delete` | `isvalid` | `fclose` | `instrfind` | `Status`

Introduced before R2006a

clf

Clear current figure window

Syntax

```
clf
clf('reset')
clf(fig)
clf(fig,'reset')
figure_handle = clf(...)
```

Description

`clf` deletes from the current figure all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to `on`).

`clf('reset')` deletes from the current figure all graphics objects regardless of the setting of their `HandleVisibility` property and resets all figure properties except `Position`, `Units`, `PaperPosition`, and `PaperUnits` to their default values.

`clf(fig)` or `clf(fig,'reset')` clears the single figure with handle `fig`.

`figure_handle = clf(...)` returns the handle of the figure. This is useful when the figure `IntegerHandle` property is `off` because the noninteger handle becomes invalid when the reset option is used (i.e., `IntegerHandle` is reset to `on`, which is the default).

Alternatives

Use **Clear Figure** from the figure window's **Edit** menu to clear the contents of a figure. You can also create a *desktop shortcut* to clear the current figure with one mouse click. See “Create Shortcuts to Rerun Commands”.

More About

Tips

The `clf` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `clf` deletes only those objects whose `HandleVisibility` property is set to `on`.

See Also

`cla` | `clc` | `hold` | `reset`

Introduced before R2006a

clipboard

Copy and paste strings to and from system clipboard

Syntax

```
clipboard('copy', data)
str = clipboard('paste')
data = clipboard('pastespecial')
```

Description

`clipboard('copy', data)` sets the clipboard contents to *`data`*. If *`data`* is not a character array, the clipboard uses `mat2str` to convert *`data`* to a string.

`str` `= clipboard('paste')` returns the current contents of the clipboard as a string or, if MATLAB cannot convert the current clipboard contents to a string, as an empty string (`''`).

`data` `= clipboard('pastespecial')` returns the current contents of the clipboard as an array by using `uiimport`.

Definitions

The `clipboard` function requires Oracle® Java software.

See Also

`load` | `mat2str` | `uiimport`

Introduced before R2006a

clock

Current date and time as date vector

Syntax

```
c = clock
```

Description

`c = clock` returns a six-element date vector containing the current date and time in decimal form:

```
[year month day hour minute seconds]
```

Examples

Round clock Output to Integer Display

Use `clock` to return the current date and time.

```
format shortg
c = clock
```

```
c =

        2015           2          23           9          55      59.151
```

The sixth element of the date vector output (seconds) is accurate to several digits beyond the decimal point.

Use the `fix` function to round to integer display format.

```
fix(c)
```

```
ans =

        2015           2          23           9          55          59
```

More About

Tips

- To time the duration of an event, use the `timeit` or `tic` and `toc` functions instead of `clock` and `etime`. The `clock` function is based on the system time, which can be adjusted periodically by the operating system, and thus might not be reliable in time comparison operations.
- To return a datetime scalar representing the current date and time, type:

  ```
  t = datetime('now')
  ```

See Also

`cputime` | `date` | `datetime` | `etime` | `fix` | `now` | `tic` | `timeit` | `toc`

Introduced before R2006a

close

Remove specified figure

Syntax

```
close
close(h)
close name
close all
close all hidden
close all force
status = close(...)
```

Description

`close` deletes the current figure or the specified figure(s). It optionally returns the status of the close operation.

`close` deletes the current figure (equivalent to `close(gcf)`).

`close(h)` deletes the figure identified by `h`. If `h` is an array, `close` deletes all figures identified by `h`. `h` can also be the figure `Number`.

`close name` deletes the figure with the specified name.

`close all` deletes all figures whose handles are not hidden.

`close all hidden` deletes all figures including those with hidden handles.

`close all force` deletes all figures, including GUIs for which `CloseRequestFcn` has been altered to not close the window.

`status = close(...)` returns `1` if the specified windows have been deleted and `0` otherwise.

More About

Algorithms

The `close` function works by evaluating the specified figure's `CloseRequestFcn` property with the statement

```
eval(get(h,'CloseRequestFcn'))
```

The default `CloseRequestFcn`, `closereq`, deletes the current figure using `delete(get(groot,'CurrentFigure'))`. If you specify an array of figure handles, `close` executes each figure's `CloseRequestFcn` in turn. If an error that terminates the execution of a `CloseRequestFcn` occurs, the figure is not deleted. Note that using your computer's window manager (i.e., the **Close** menu item) also calls the figure's `CloseRequestFcn`.

If a figure's handle is hidden (i.e., the figure's `HandleVisibility` property is set to `callback` or `off` and the root `ShowHiddenHandles` property is set to `on`), you must specify the `hidden` option when trying to access a figure using the `all` option.

To delete all figures unconditionally, use the statements

```
set(groot,'ShowHiddenHandles','on')
c = get(groot,'Children');
delete(c)
```

The figure `CloseRequestFcn` allows you to either delay or abort the closing of a figure once the `close` function has been issued. For example, you can display a dialog box to see if the user really wants to delete the figure or save and clean up before closing.

When coding a `CloseRequestFcn` callback, make sure that it does not call `close`, because this sets up a recursion that results in a MATLAB warning. Instead, the callback should destroy the figure with `delete`. The `delete` function does not execute the figure's `CloseRequestFcn`; it deletes the specified figure.

See Also

`delete` | `figure` | `gcf`

Introduced before R2006a

close

Class: Tiff

Close Tiff object

Syntax

```
close(tiffobj)
```

Description

`close(tiffobj)` closes a Tiff object.

Examples

Close Tiff Object

Open a Tiff object and then close it.

```
t = Tiff('example.tif','r');
close(t)
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFClose` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTiff - TIFF Library and Utilities.

close (avifile)

Close Audio/Video Interleaved (AVI) file

Note: `avifile` has been removed. Use `VideoWriter` instead.

Syntax

```
aviobj = close(aviobj)
```

Description

`aviobj = close(aviobj)` finishes writing and closes the AVI file associated with *`aviobj`*, which is an AVI file object created using the `avifile` function.

To close all open AVI files, use the `clear mex` command.

See Also

`avifile` | `addframe (avifile)`

Introduced before R2006a

close

Class: FTP

Close connection to FTP server

Syntax

```
close(ftpobj)
```

Description

`close(ftpobj)` closes the connection to the FTP server.

Tips

- If you do not run `close` at the end of your session, the connection either times out automatically or terminates when you exit MATLAB.
- After calling `close`, calling any other FTP method on the same object automatically reopens the connection.
- `close` does not return any output to indicate success or failure.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

Examples

Connect to the MathWorks FTP server, and then disconnect:

```
mw=ftp('ftp.mathworks.com');
```

```
close(mw)
```

See Also

`ftp`

Introduced before R2006a

close

Class: VideoWriter

Close file after writing video data

Syntax

```
close(writerObj)
```

Description

`close(writerObj)` closes the file associated with object writerObj. The object remains in the workspace. If you call the `open` method after closing, `open` discards any existing contents of the file.

Input Arguments

`writerObj`

`VideoWriter` object created by the `VideoWriter` function.

Examples

AVI File from Animation

Write a sequence of frames to a compressed AVI file, `peaks.avi`.

Prepare the new file.

```
writerObj = VideoWriter('peaks.avi');
open(writerObj);
```

Generate initial data and set axes and figure properties.

```
Z = peaks; surf(Z);
```

```
axis tight
set(gca,'nextplot','replacechildren');
set(gcf,'Renderer','zbuffer');
```

Setting the `Renderer` property to `zbuffer` or `Painters` works around limitations of `getframe` with the OpenGL renderer on some Windows systems.

Create a set of frames and write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe;
   writeVideo(writerObj,frame);
end

close(writerObj);
```

See Also

`open` | `writeVideo` | `VideoWriter`

closereq

Default figure close request function

Syntax

```
closereq
```

Description

`closereq` deletes the current figure. For more information, see the `CloseRequestFcn` figure property.

Introduced before R2006a

cmopts

(To be removed) Name of source control system

Note: `cmopts` will be removed in a future release. View the currently selected source control system through **Preferences** instead.

Syntax

```
cmopts
```

Description

`cmopts` returns the name of your version control system.

Output Arguments

Value Returned by cmopts	Description	Platform Supported On
`clearcase`	ClearCase® software from IBM® Rational®	UNIX platforms
`customverctrl`	Custom interface created using `customverctrl` function	UNIX platforms
`cvs`	Concurrent Version System (CVS)	UNIX platforms
`none`	No source control system selected	
`pvcs`	PVCS® and ChangeMan® software	UNIX platforms
`rcs`	Revision Control System (RCS)	UNIX platforms

Value Returned by cmopts	Description	Platform Supported On
Any SCC-compliant source control system, for example, `Microsoft Visual SourceSafe`	Varies	Windows platforms

Alternatives

To view the currently selected source control system, click the **Preferences** button on the **Home** tab, and select **General > Source Control**.

Introduced before R2006a

cmpermute

Rearrange colors in colormap

Syntax

```
[Y,newmap] = cmpermute(X,map)
[Y,newmap] = cmpermute(X,map,index)
```

Description

`[Y,newmap] = cmpermute(X,map)` randomly reorders the colors in `map` to produce a new colormap, `newmap`. The `cmpermute` function also modifies the values in `X` to maintain correspondence between the indices and the colormap, and returns the result in `Y`. The image `Y` and associated colormap, `newmap`, produce the same image as `X` and `map`.

`[Y,newmap] = cmpermute(X,map,index)` uses an ordering matrix (such as the second output of `sort`) to define the order of colors in the new colormap.

Class Support

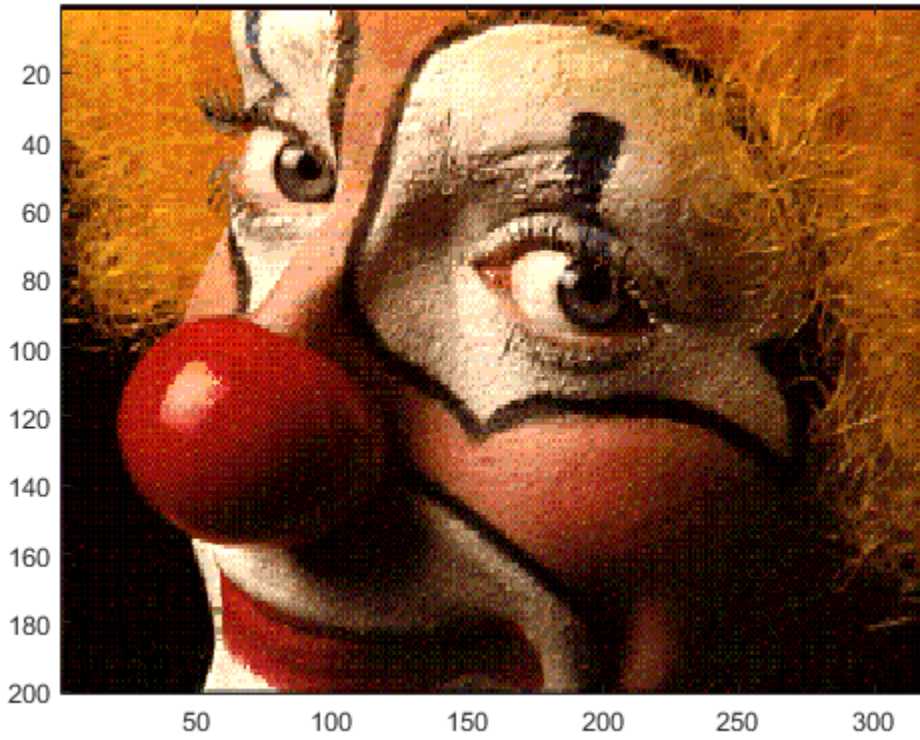
The input image `X` can be of class `uint8` or `double`. `Y` is returned as an array of the same class as `X`.

Examples

Randomly Reorder Colormap and Display Image

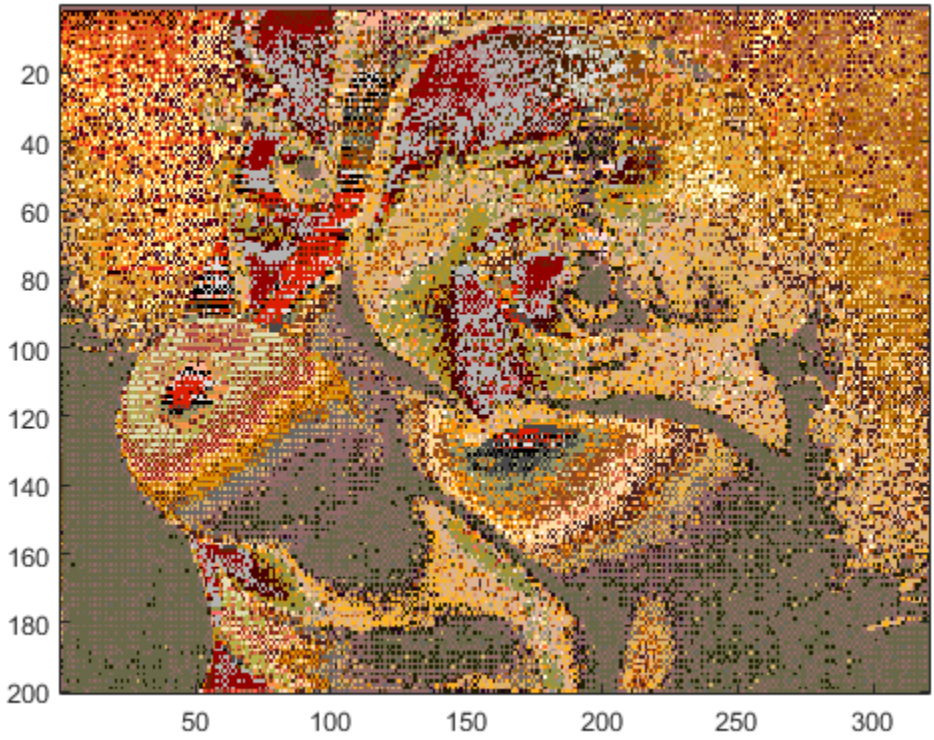
Load the `clown` data set to get image `X` and its associated colormap, `map`. Display the image.

```
load clown
figure
image(X)
colormap(map)
```



Randomly reorder the colormap to get the new colormap, `newmap`. Display image `X` with the new colormap.

```
[Y, newmap] = cmpermute(X,map);
colormap(newmap)
```



See Also

`randperm` | `sort`

cmunique

Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed image

Syntax

```
[Y,newmap] = cmunique(X,map)
[Y,newmap] = cmunique(RGB)
[Y,newmap] = cmunique(I)
```

Description

`[Y,newmap] = cmunique(X,map)` returns the indexed image `Y` and associated colormap, `newmap`, that produce the same image as `(X,map)` but with the smallest possible colormap. The `cmunique` function removes duplicate rows from the colormap and adjusts the indices in the image matrix accordingly.

`[Y,newmap] = cmunique(RGB)` converts the truecolor image `RGB` to the indexed image `Y` and its associated colormap, `newmap`. The return value `newmap` is the smallest possible colormap for the image, containing one entry for each unique color in `RGB`.

Note: `newmap` might be very large, because the number of entries can be as many as the number of pixels in `RGB`.

`[Y,newmap] = cmunique(I)` converts the grayscale image `I` to an indexed image `Y` and its associated colormap, `newmap`. The return value, `newmap`, is the smallest possible colormap for the image, containing one entry for each unique intensity level in `I`.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`. The class of the output image `Y` is `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `Y` is of class `double`.

Examples

Eliminate Duplicate Entries in Colormap

Use the `magic` function to define `X` as a 4-by-4 array that uses every value in the range between 1 and 16.

```
X = magic(4);
```

Use the `gray` function to create an eight-entry colormap. Then, concatenate the two eight-entry colormaps to create a colormap with 16 entries, `map`. In `map`, entries 9 through 16 are duplicates of entries 1 through 8.

```
map = [gray(8); gray(8)];
size(map)
```

```
ans =

    16     3
```

Use `cmunique` to eliminate duplicate entries in the colormap.

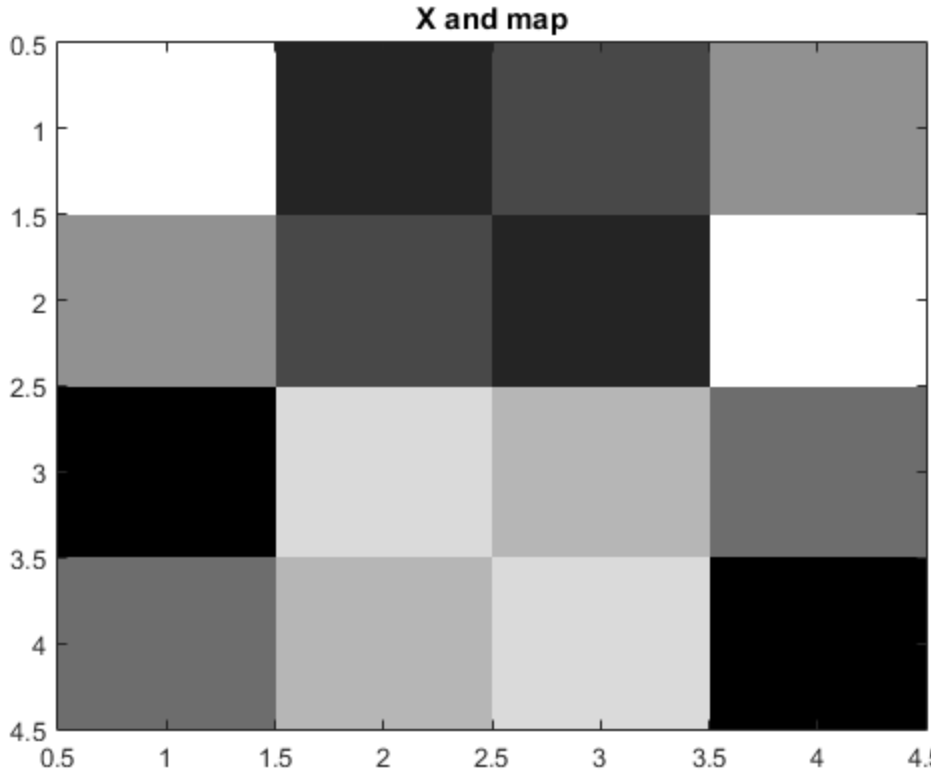
```
[Y, newmap] = cmunique(X, map);
size(newmap)
```

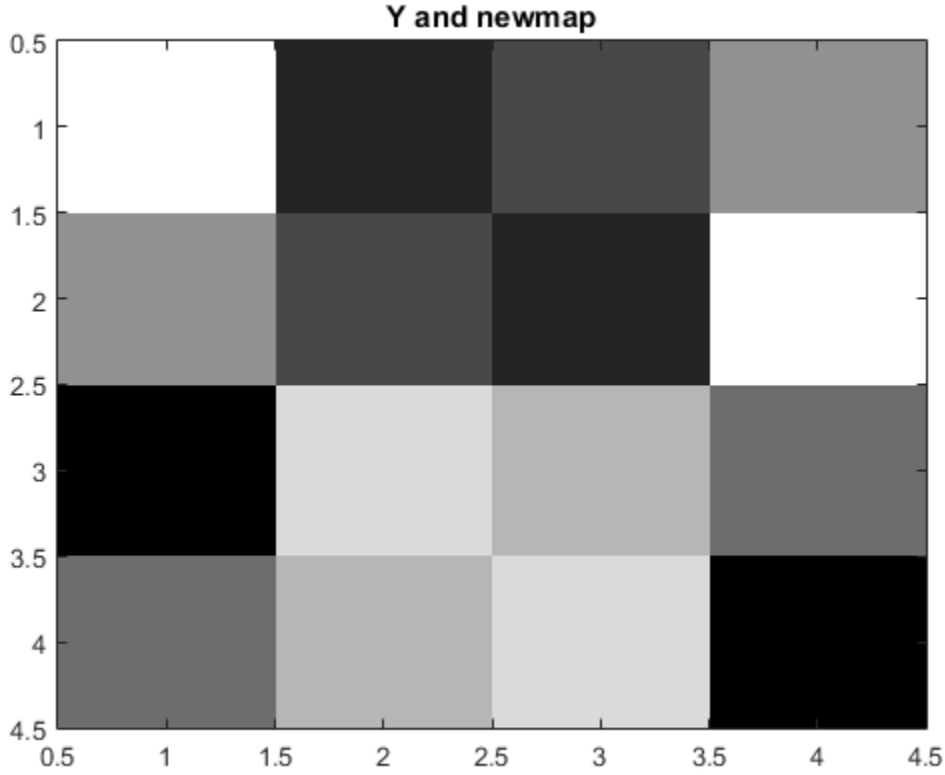
```
ans =

     8     3
```

`cmunique` adjusts the values in the original image `X` so that `Y` and `newmap` produce the same image as `X` and `map`.

```
figure
image(X)
colormap(map)
title('X and map')

figure
image(Y)
```

```
colormap(newmap)
title('Y and newmap')
```





See Also

`rgb2ind`

colamd

Column approximate minimum degree permutation

Syntax

```
p = colamd(S)
```

Description

`p = colamd(S)` returns the column approximate minimum degree permutation vector for the sparse matrix `S`. For a non-symmetric matrix `S`, `S(:,p)` tends to have sparser LU factors than `S`. The Cholesky factorization of `S(:,p)' * S(:,p)` also tends to be sparser than that of `S'*S`.

`knobs` is a two-element vector. If S is `m`-by-`n`, then rows with more than `(knobs(1))*n` entries are ignored. Columns with more than `(knobs(2))*m` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs(1)` = `knobs(2) = spparms('wh_frac')`.

`stats` is an optional vector that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `colamd`
`stats(2)`	Number of dense or empty columns ignored by `colamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `colamd` (roughly of size `2.2*nnz(S) + 4*m + 7*n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid
`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists
`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `colamd`. For this reason, `colamd` verifies that `S` is valid:

- If a row index appears two or more times in the same column, `colamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `colamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `colamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`) .

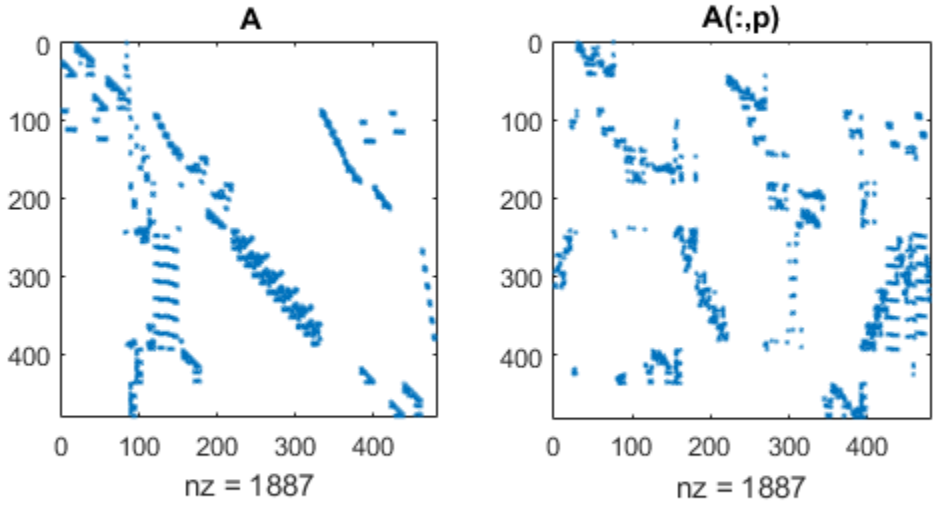
The ordering is followed by a column elimination tree post-ordering.

Examples

Compare Sparse Matrix and LU Factorization

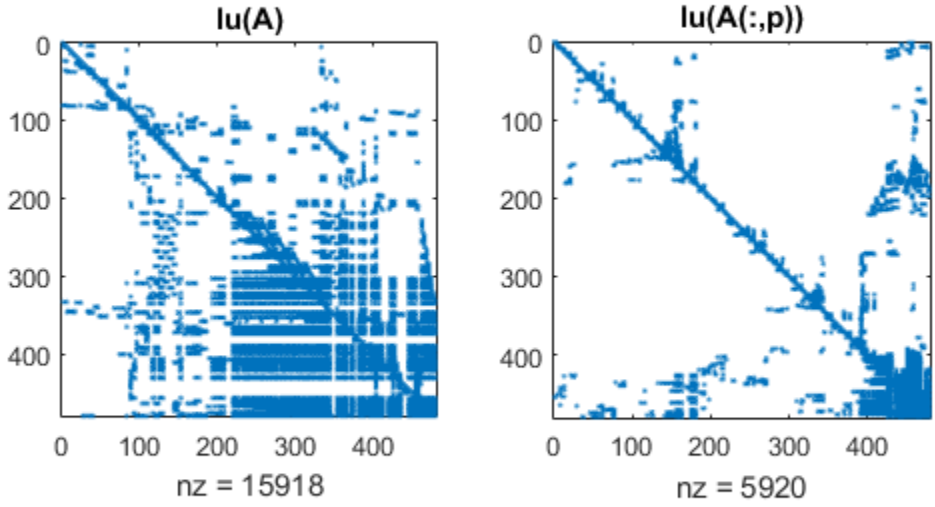
The Harwell-Boeing collection of sparse matrices and the MATLAB® demos directory include a test matrix `west0479`. It is a matrix of order 479 resulting from a model due to Westerberg of an eight-stage chemical distillation column. The spy plot shows evidence of the eight stages. The `colamd` ordering scrambles this structure.

```
load west0479
A = west0479;
p = colamd(A);

figure()
subplot(1,2,1), spy(A,4), title('A')
subplot(1,2,2), spy(A(:,p),4), title('A(:,p)')
```



Comparing the spy plot of the LU factorization of the original matrix with that of the reordered matrix shows that minimum degree reduces the time and storage requirements by better than a factor of 2.8. The nonzero counts are 15918 and 5920, respectively.

```
figure()
subplot(1,2,1), spy(lu(A),4), title('lu(A)')
subplot(1,2,2), spy(lu(A(:,p)),4), title('lu(A(:,p))')
```



References

[1] The authors of the code for “colamd” are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. Sparse Matrix Algorithms Research at the University of Florida: http://www.cise.ufl.edu/research/sparse/

See Also

`colperm` | `symamd` | `symrcm` | `spparms`

Introduced before R2006a

colorbar

Colorbar showing color scale

Syntax

```
colorbar
colorbar(placement)
colorbar(Name,Value)
colorbar(placement,Name,Value)

colorbar(ax, ___ )
colorbar('peer',ax ___ )
h = colorbar( ___ )

colorbar('off')
colorbar(h,'off')
colorbar(ax,'off')
```

Description

`colorbar` displays a vertical colorbar to the right of the current axes. Colorbars display the current colormap and indicate the mapping of data values into the colormap.

`colorbar(placement)` displays a colorbar in the location specified by `placement`, which is a location string such as `'northoutside'`.

`colorbar(Name,Value)` specifies colorbar properties using one or more `Name,Value` pair arguments. For example, `'Direction','reverse'` reveres the color scale.

`colorbar(placement,Name,Value)` specifies both the colorbar location and other colorbar properties.

`colorbar(ax, ___ )` adds a colorbar to the axes specified by `ax` instead of the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in the previous syntaxes.

`colorbar('peer',ax___)` adds a colorbar to the axes specified by `ax` instead of the current axes. This syntax is not recommended and might be removed in a future release. Use `colorbar(ax,__)` instead.

`h = colorbar(___)` returns the colorbar object. Use `h` to set properties of the colorbar after it is created.

Note: Starting in R2014b, the `colorbar` function returns a colorbar object. In previous releases it returns an axes object.

`colorbar('off')` deletes all colorbars associated with the current axes.

`colorbar(h,'off')` deletes the colorbar specified by `h`.

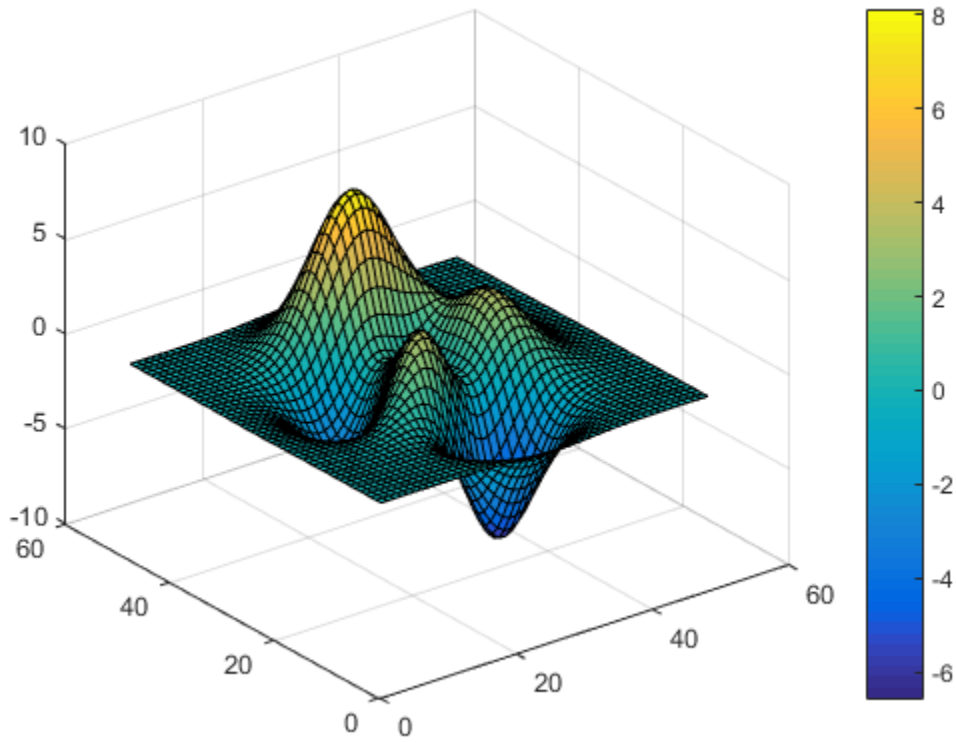
`colorbar(ax,'off')` deletes all colorbars associated with the axes specified by `ax` instead of the current axes (`gca`).

Examples

Add Colorbar to Graph

Add a colorbar to a surface plot indicating the color scale.

```
figure
surf(peaks)
colorbar
```

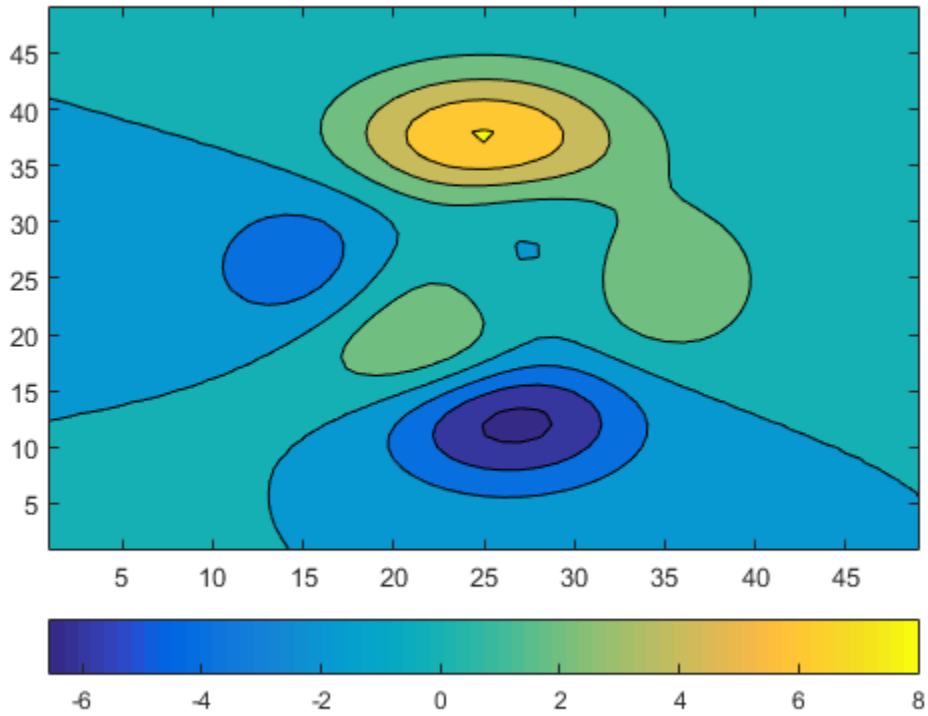


By default, the `colorbar` function adds a vertical colorbar to the right side of the graph.

Add Horizontal Colorbar to Graph

Add a horizontal colorbar below a plot by specifying the colorbar location as `'southoutside'`.

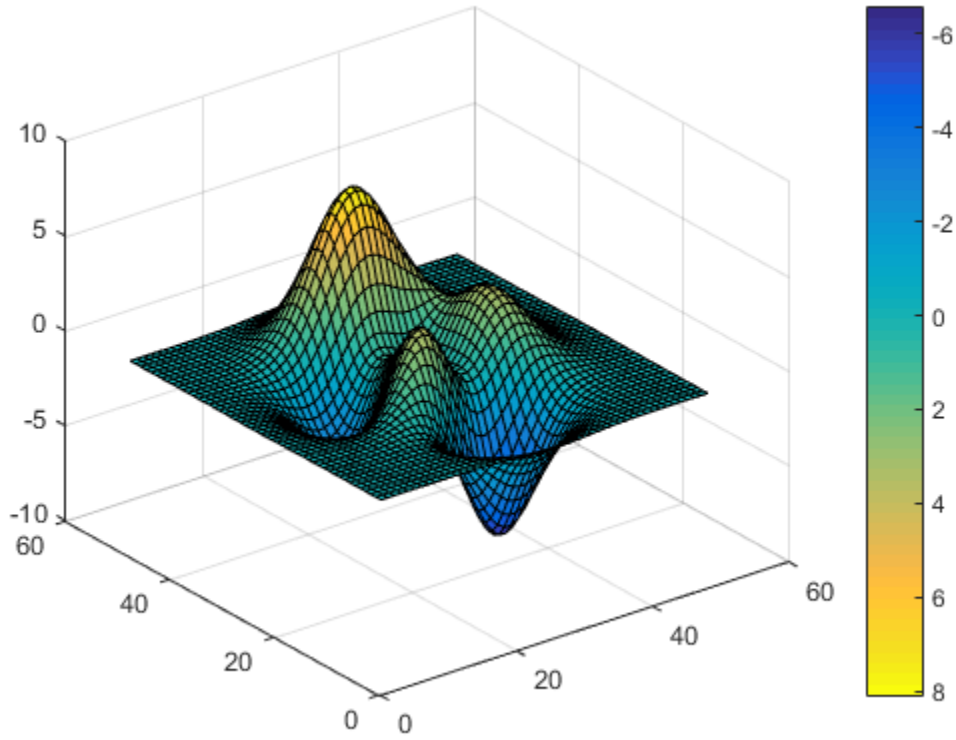
```
figure
contourf(peaks)
colorbar('southoutside')
```



Reverse Colorbar Direction

Reverse the direction of values in a colorbar on a graph by setting the `'Direction'` property of the colorbar to `'reverse'`.

```
figure
surf(peaks)
colorbar('Direction','reverse')
```

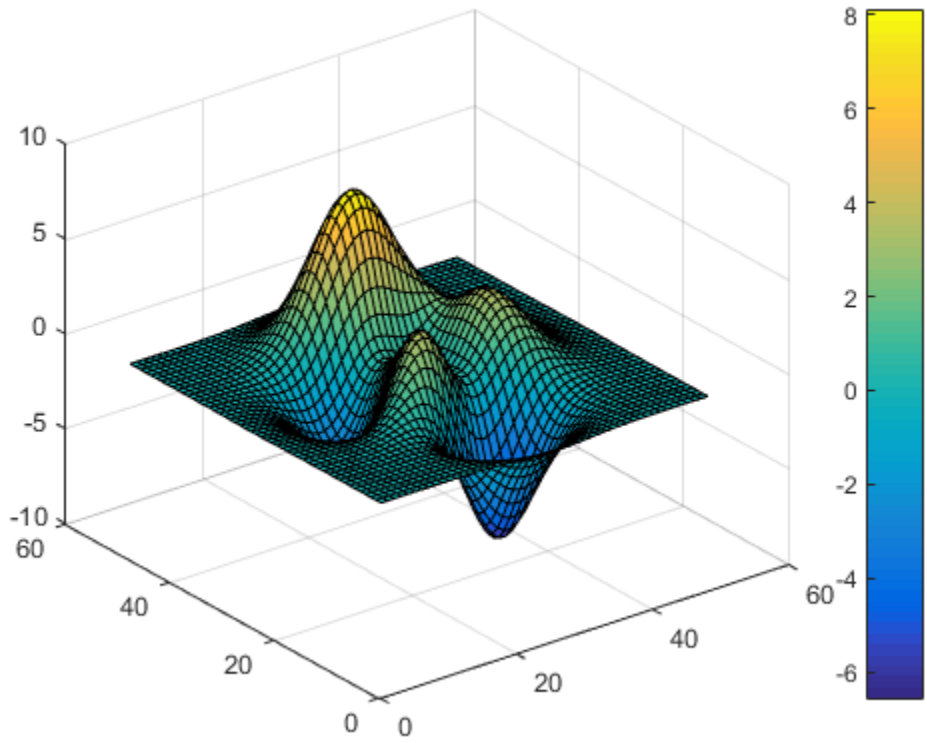


The colorbar values ascend from top to bottom instead of ascending from bottom to top.

Display Colorbar Ticks on Opposite Side

Display the colorbar tick marks and tick labels on the side of a colorbar facing the surface plot.

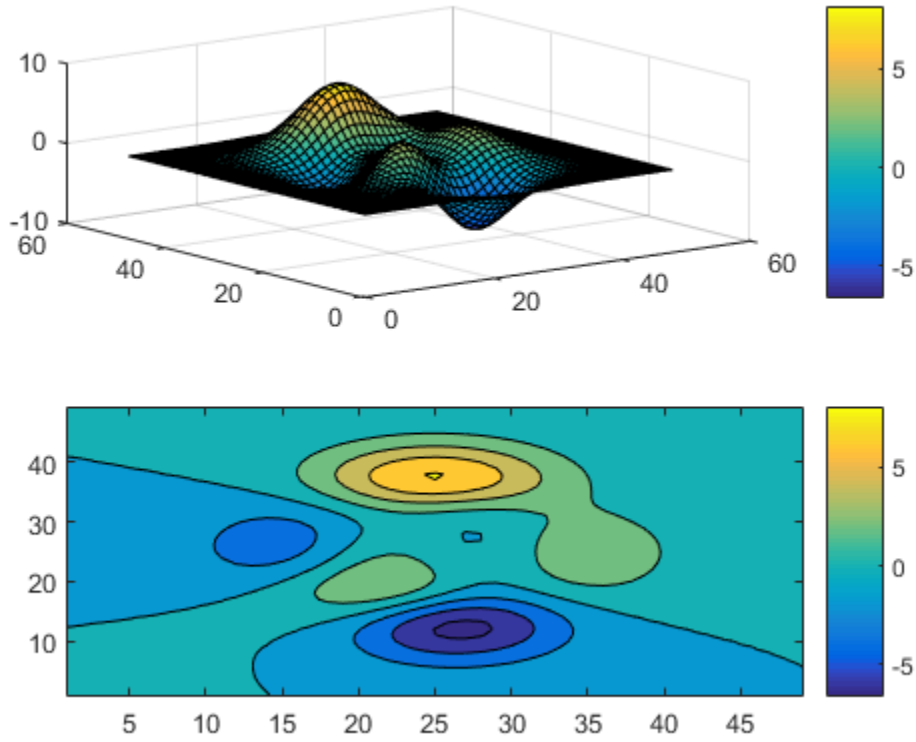
```
surf(peaks)
colorbar('AxisLocation','in')
```



Add Colorbars to Subplots

Create a figure with two subplots. Add colorbars to both subplots.

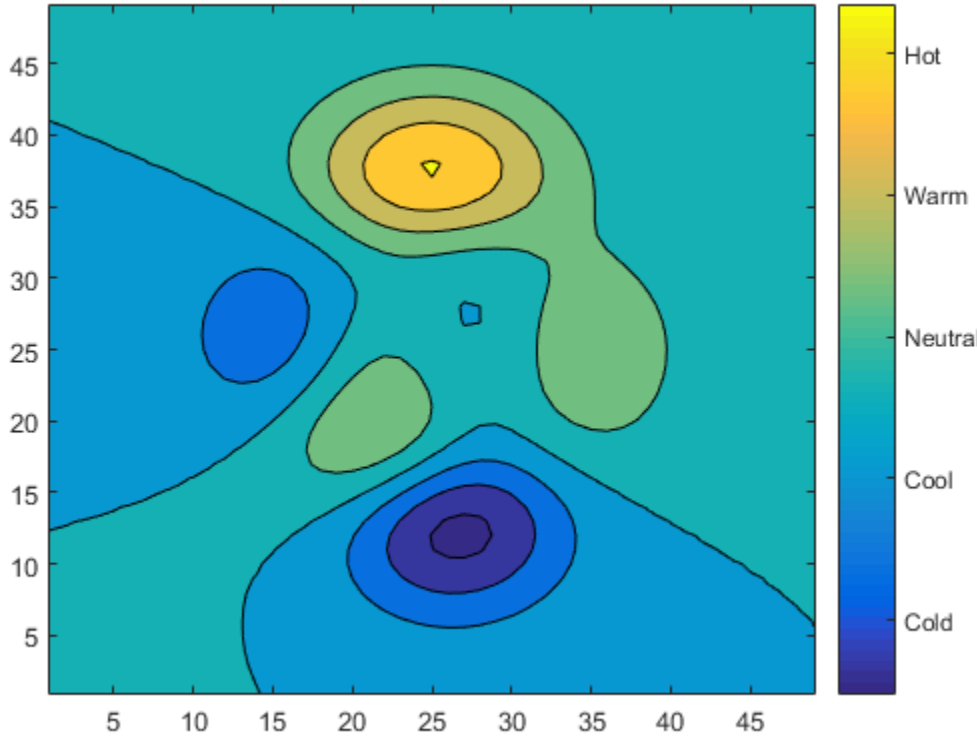
```
figure
subplot(2,1,1) % upper subplot
surf(peaks)
colorbar

subplot(2,1,2) % lower subplot
contourf(peaks)
colorbar
```



Specify Colorbar Ticks and Tick Labels

Add a colorbar to a plot and specify the colorbar tick marks and tick labels. Specify the same number of tick labels as tick marks. If you do not specify enough tick labels, then the `colorbar` function repeats the labels.

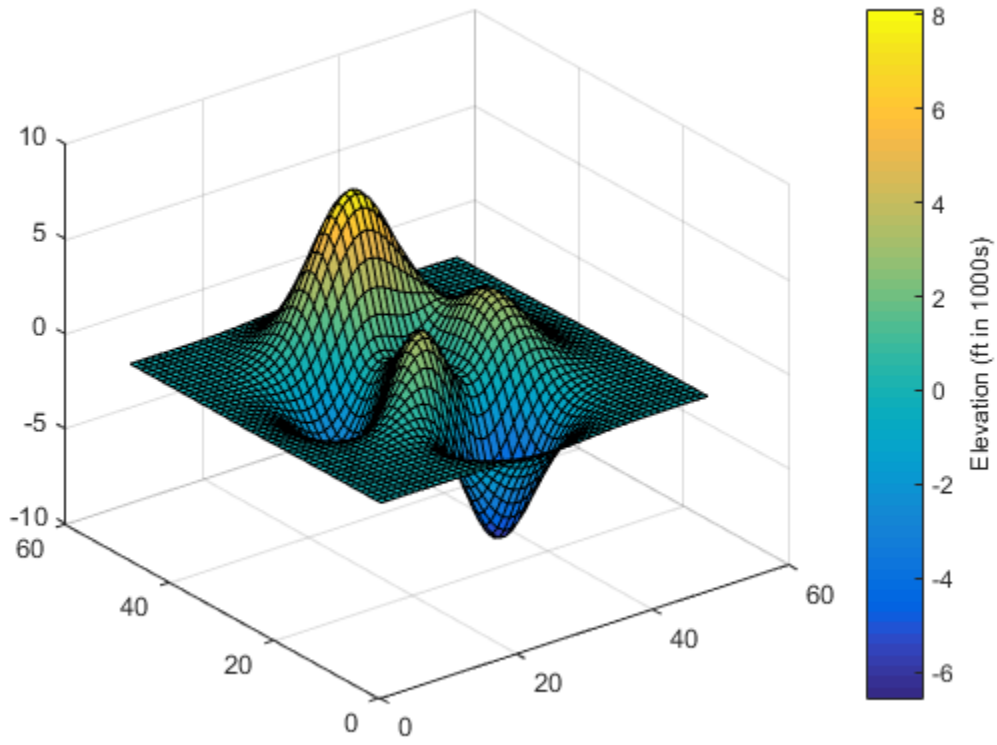
```
contourf(peaks)
colorbar('Ticks',[-5,-2,1,4,7],...
         'TickLabels',{'Cold','Cool','Neutral','Warm','Hot'})
```



Label Colorbar

Add a text label along a colorbar. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

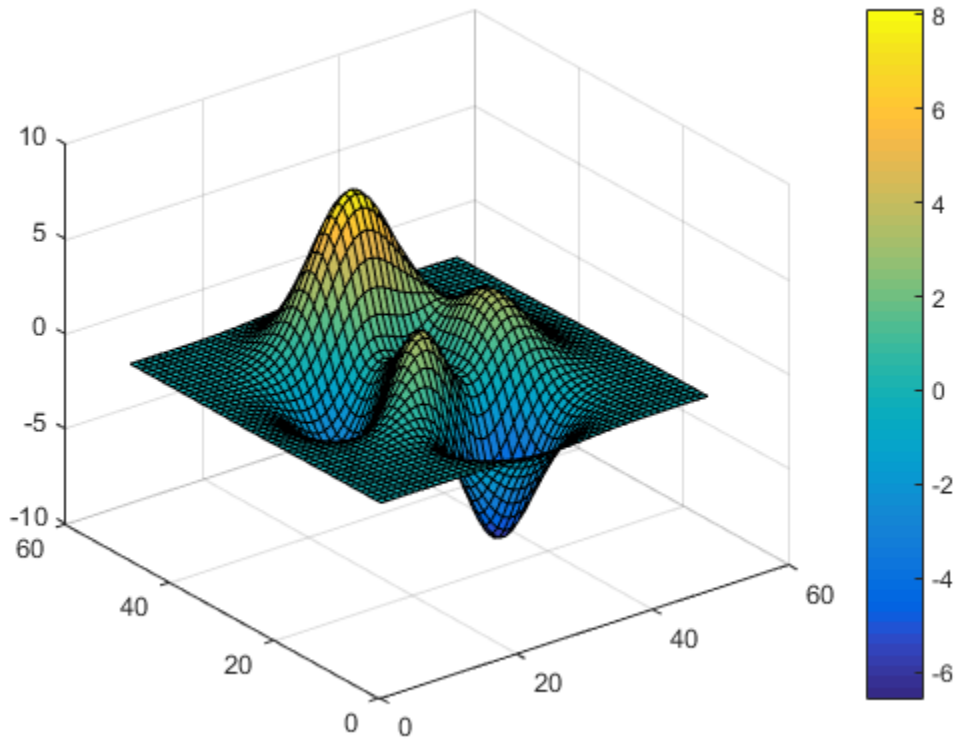
```
surf(peaks)
c = colorbar;
c.Label.String = 'Elevation (ft in 1000s)';
```



Delete Colorbar

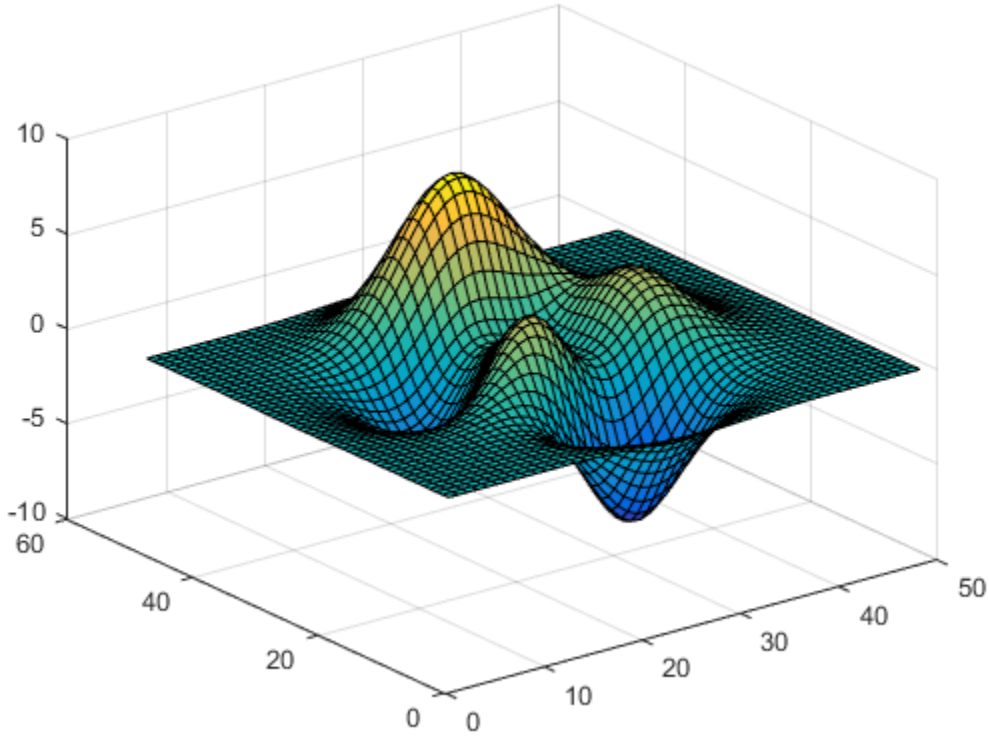
Add a colorbar to a surface plot.

```
figure
surf(peaks)
colorbar
```



Delete the colorbar from the surface plot.

```
colorbar('off')
```



- “Change Colorbar Width”
- “Change Mapping of Data Values into the Colormap”

Input Arguments

`placement` — Colorbar location and orientation

`'eastoutside'` (default) | location string

Colorbar location and orientation with respect to the axes, specified as a location string. This table lists the location strings.

String	Resulting Location	Resulting Orientation
`'north'`	Top of axes	Horizontal
`'south'`	Bottom of axes	Horizontal
`'east'`	Right side of axes	Vertical
`'west'`	Left side of axes	Vertical
`'northoutside'`	Top outside of axes	Horizontal
`'southoutside'`	Bottom outside of axes	Horizontal
`'eastoutside'`	Right outside of axes (default)	Vertical
`'westoutside'`	Left outside of axes	Vertical

If a colorbar already exists in the specified location, then an updated colorbar replaces the existing one. To ensure that the colorbar does not overlap the graph, specify a location with the suffix, `outside`.

You also can set the colorbar location using its `Location` property. For example, `colorbar('Location','northoutside')` is the same as `colorbar('northoutside')`. If you use the shorthand syntax, `colorbar('northoutside')`, then MATLAB sets the colorbar `Location` property value to `'northoutside'`.

Example: `colorbar('westoutside')`

ax — Axes object

axes object

Axes object. If you do not specify an axes, then the `colorbar` function adds a colorbar for the current axes.

Name-Value Pair Arguments

The colorbar properties listed here are only a subset. For a complete list see Colorbar Properties.

Example: `colorbar('FontSize',12,'Direction','reverse')` sets the font size of the colorbar to 12 points and reverses the orientation of the colorbar.

'Location' — Location with respect to the axes

`'eastoutside'` (default) | tring

Location with respect to the axes, specified as one of the strings listed in this table.

String	Resulting Location	Resulting Orientation
`'north'`	Top of axes	Horizontal
`'south'`	Bottom of axes	Horizontal
`'east'`	Right side of axes	Vertical
`'west'`	Left side of axes	Vertical
`'northoutside'`	Top outside of axes	Horizontal
`'southoutside'`	Bottom outside of axes	Horizontal
`'eastoutside'`	Right outside of axes (default)	Vertical
`'westoutside'`	Left outside of axes	Vertical
`'manual'`	Determined by `Position` property	Vertical

To display the colorbar in a location that does not appear in the table, use the Position property to specify a custom location. If you set the Position property, then MATLAB sets the Location property to `'manual'`. The associated axes does not resize to accommodate the colorbar when the Location property is set to `'manual'`.

`'Label'` — Label that displays along colorbar

text object (default)

Label that displays along the colorbar, returned as a text object. This text object contains properties that control the label appearance and the text that displays. Use the `Label` property to access the text object, for example:

```
c = colorbar;
c.Label

ans =

  Text with properties:

               String: ''
             FontSize: 10
           FontWeight: 'normal'
             FontName: 'Helvetica'
```

```
                Color: [0.1500 0.1500 0.1500]
   HorizontalAlignment: 'left'
              Position: [0 0 0]
                 Units: 'data'

  Show all properties
```

By default, there is no text displayed. To display text or change existing text, set the String property for the text object, for example:

```
c.Label.String = 'Label Text Goes Here';
```

To change the label appearance, such as the font style or color, set other text properties. For example, this code changes the font size.

```
c.Label.FontSize = 12;
```

For a full list of options, see Text Properties.

Note: The `Label` property is different from other colorbar properties. You cannot set the `Label` property as a name-value pair when creating the colorbar. Instead, use dot notation (demonstrated above) or the `set` function.

`'TickLabels'` — Tick mark labels

cell array of strings | numeric array | string

Tick mark labels, specified as a cell array of strings, a numeric array, or a string. By default, the colorbar labels the tick marks with numeric values. If you specify labels and do not specify enough labels for all the tick marks, then MATLAB cycles through the labels.

Example: `{'cold','warm','hot'}`

`'TickLabelInterpreter'` — Interpretation of characters in tick labels

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret strings using a subset of the TeX markup.
- `'latex'` — Interpret strings using a subset of LaTeX markup.
- `'none'` — Display literal characters

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Modifiers remain in effect until the end of the string, except for superscripts and subscripts which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	Φ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	ß	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	Θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	v	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The FontName, FontWeight, and FontAngle properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

`'Ticks'` — Tick mark locations

vector of monotonically increasing numeric values

Tick mark locations, specified as a vector of monotonically increasing numeric values. The values do not need to be equally spaced. If you do not want tick marks displayed, then set the property to the empty vector, `[]`.

Example: `[-1,0,1,2,3,4,5]`

Data Types: `single` | `double`

`'Direction'` — Direction of color scale

`'normal'` (default) | `'reverse'`

Direction of color scale, specified as one of these values:

- `'normal'` — Display the colormap and labels ascending from bottom to top for a vertical colorbar, and ascending from left to right for a horizontal colorbar. This is the default value.
- `'reverse'` — Display the colormap and labels descending from bottom to top for a vertical colorbar, and descending from left to right for a horizontal colorbar.

`'FontSize'` — Font size
9 (default) | scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default value is 9 points. If you change the axes font size, then MATLAB automatically sets the colorbar font size to 90% of the axes font size. If you manually set the colorbar font size, then changing the axes font size does not affect the colorbar.

Example: `12`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

`h` — Colorbar object
colorbar object

Colorbar object. Use `h` to set properties of the colorbar after it is created.

More About

Tips

- Adding a colorbar might resize the axes to accommodate the colorbar.
- If an axes does not exist, then the `colorbar` function creates a blank axes and displays a colorbar with the default colormap.
- You can use `colorbar('delete')` or `colorbar('hide')` instead of `colorbar('off')` to delete all colorbars in the current axes.

See Also

Functions
`caxis` | `colormap`

Properties
Colorbar Properties

Introduced before R2006a

Colorbar Properties

Control colorbar appearance and behavior

Colorbar properties control the appearance and behavior of a colorbar object. By changing property values, you can modify certain aspects of the colorbar. Use dot notation to refer to a particular object and property:

```
c = colorbar;
w = c.LineWidth;
c.LineWidth = 1.5;
```

Appearance

`Color` — Color of tick marks, text, and box outline

`[0 0 0]` (default) | RGB triplet | color string

Color of the tick marks, text, and box outline, specified as an RGB triplet or a color string.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0 1 0]`

Example: `'green'`

`Box` — Box outline

`'on'` (default) | `'off'`

Box outline, specified as one of these values:

- `'on'` — Display the box outline around the colorbar.
- `'off'` — Do not display the box outline around the colorbar.

`LineWidth` — Width of box outline

`0.5` (default) | positive value

Width of box outline, specified as a positive value in point units. One point equals 1/72 inches.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Label` — Label that displays along colorbar

text object (default)

Label that displays along the colorbar, returned as a text object. This text object contains properties that control the label appearance and the text that displays. Use the `Label` property to access the text object, for example:

```
c = colorbar;
c.Label
```

```
ans =

  Text with properties:

               String: ''
             FontSize: 10
           FontWeight: 'normal'
             FontName: 'Helvetica'
                Color: [0.1500 0.1500 0.1500]
    HorizontalAlignment: 'left'
             Position: [0 0 0]
                Units: 'data'

  Show all properties
```

By default, there is no text displayed. To display text or change existing text, set the String property for the text object, for example:

```
c.Label.String = 'Label Text Goes Here';
```

To change the label appearance, such as the font style or color, set other text properties. For example, this code changes the font size.

```
c.Label.FontSize = 12;
```

For a full list of options, see Text Properties.

Note: The `Label` property is different from other colorbar properties. You cannot set the `Label` property as a name-value pair when creating the colorbar. Instead, use dot notation (demonstrated above) or the `set` function.

Location and Size

`Location` — Location with respect to the axes

`'eastoutside'` (default) | tring

Location with respect to the axes, specified as one of the strings listed in this table.

String	Resulting Location	Resulting Orientation
`'north'`	Top of axes	Horizontal
`'south'`	Bottom of axes	Horizontal
`'east'`	Right side of axes	Vertical
`'west'`	Left side of axes	Vertical
`'northoutside'`	Top outside of axes	Horizontal
`'southoutside'`	Bottom outside of axes	Horizontal
`'eastoutside'`	Right outside of axes (default)	Vertical
`'westoutside'`	Left outside of axes	Vertical
`'manual'`	Determined by `Position` property	Vertical

To display the colorbar in a location that does not appear in the table, use the Position property to specify a custom location. If you set the Position property, then MATLAB sets the Location property to `'manual'`. The associated axes does not resize to accommodate the colorbar when the Location property is set to `'manual'`.

`Position` — Custom location and size

four-element vector

Custom location and size, specified as a four-element vector of the form `[left, bottom, width, height]`. The `left` and `bottom` elements specify the distance from the lower-left corner of the figure or to the lower-left corner of the colorbar. The `width` and `height` elements specify the dimensions of the colorbar. The Units property determines the position units.

If you specify the Position property, then MATLAB changes the Location property to `'manual'`. The associated axes does not resize to accommodate the colorbar when the `Location` property is `'manual'`.

Example: `[0.1 0.1 0.3 0.7]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

`Units`	Description
`'normalized'` (default)	Normalized with respect to the container, which is usually the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.

Units	Description
'pixels'	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the container window.

This property affects the Position property. If you change the units, then it is good practice to return it to its default value after completing your computation to prevent affecting other functions that assume Units is the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the colorbar, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

Tick Marks and Tick Labels

`Ticks` — Tick mark locations

vector of monotonically increasing numeric values

Tick mark locations, specified as a vector of monotonically increasing numeric values. The values do not need to be equally spaced. If you do not want tick marks displayed, then set the property to the empty vector, `[]`.

Example: `[-1,0,1,2,3,4,5]`

Data Types: `single` | `double`

`TicksMode` — Selection mode for `Ticks`

`'auto'` (default) | `'manual'`

Selection mode for `Ticks`, specified as one of these values:

- `'auto'` — Automatically choose the values.
- `'manual'` — Use manually specified values. To specify the values, set the Ticks property.

`TickLabels` — Tick mark labels

cell array of strings | numeric array | string

Tick mark labels, specified as a cell array of strings, a numeric array, or a string. By default, the colorbar labels the tick marks with numeric values. If you specify labels and

do not specify enough labels for all the tick marks, then MATLAB cycles through the labels.

Example: `{'cold','warm','hot'}`

`TickLabelsMode` — Selection mode for `TickLabels`

`'auto'` (default) | `'manual'`

Selection mode for `TickLabels`, specified as one of these values:

- `'auto'` — Automatically choose the tick mark labels.
- `'manual'` — Use manually specified values. To specify the values, set the TickLabels property.

`TickLabelInterpreter` — Interpretation of characters in tick labels

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret strings using a subset of the TeX markup.
- `'latex'` — Interpret strings using a subset of LaTeX markup.
- `'none'` — Display literal characters

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Modifiers remain in effect until the end of the string, except for superscripts and subscripts which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`

Modifier	Description	Example of String
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsui`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ε	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrighta`	↔

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	Θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	°
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The FontName, FontWeight, and FontAngle properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

`Direction` — Direction of color scale

`'normal'` (default) | `'reverse'`

Direction of color scale, specified as one of these values:

- `'normal'` — Display the colormap and labels ascending from bottom to top for a vertical colorbar, and ascending from left to right for a horizontal colorbar. This is the default value.
- `'reverse'` — Display the colormap and labels descending from bottom to top for a vertical colorbar, and descending from left to right for a horizontal colorbar.

`AxisLocation` — Location of tick marks, tick labels, and colorbar label

`'out'` (default) | `'in'`

Location of tick marks, tick labels, and colorbar label, specified as one of these values:

- `'out'` — Display the tick marks and labels on the side of the colorbar towards the outside of the figure. This is the default value.
- `'in'` — Display the tick marks and labels on the side of the colorbar towards the inside of the figure.

`AxisLocationMode` — Selection mode for `AxisLocation`

`'auto'` (default) | `'manual'`

Selection mode for `AxisLocation`, specified as one of these values:

- `'auto'` — Automatically choose the location.
- `'manual'` — Use a manually specified location. To specify the location, set the AxisLocation property.

`TickDirection` — Tick mark direction

`'in'` (default) | `'out'`

Tick mark direction, specified as one of these values:

- `'in'` — Display the tick marks facing inward from the colorbar box.
- `'out'` — Display the tick marks facing outward from the colorbar box.

`TickLength` — Tick mark length

`0.01` (default) | scalar

Tick mark length, specified as a scalar. Specify the tick length in units normalized to the length of the colorbar axis.

Example: `0.05`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Limits` — Minimum and maximum tick mark values

two-element vector

The minimum and maximum tick mark values, specified as a two-element vector. The second vector element must be greater than the first element.

Example: `[0 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LimitsMode` — Selection mode for limits

`'auto'` (default) | `'manual'`

Selection mode for limits, specified as one of these values:

- `'auto'` — Automatically chooses the limits.
- `'manual'` — Use manually specified limits. To specify the limits, set the Limits property.

Font Style

FontAngle — Character slant

'normal' (default) | 'italic'

Character slant, specified as 'normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Note: The 'oblique' value has been removed. Use 'italic' instead.

FontName — Font name

'Helvetica' (default) | 'FixedWidth' | system supported font name

Font name, specified as the name of the font to use or the string 'FixedWidth'. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string 'FixedWidth'. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The 'FixedWidth' value relies on the root FixedWidthFontName property. Setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

Example: 'Cambria'

FontSize — Font size

9 (default) | scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default value is 9 points. If you change the axes font size, then MATLAB automatically sets the colorbar font size to 90% of the axes font size. If you manually set the colorbar font size, then changing the axes font size does not affect the colorbar.

Example: 12

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

FontWeight — Thickness of text characters

'normal' (default) | 'bold'

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

Visibility

`Visible` — Visibility of colorbar

`'on'` (default) | `'off'`

Visibility of colorbar, specified as one of these values:

- `'on'` — Display the colorbar.
- `'off'` — Hide the colorbar without deleting it. You still can access the properties of an invisible colorbar object.

Identifiers

`Type` — Type of graphics object

`'colorbar'`

Type of graphics object, returned as `'colorbar'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

`Tag` — Tag to associate with colorbar

`'colorbar'` (default) | string

Tag to associate with the colorbar, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with colorbar

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the colorbar object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent of colorbar

figure object | uipanel object | uitab object

Parent of the colorbar, specified as a figure object, uipanel object, or a uitab object.

The colorbar must have the same parent as the associated axes. If you change the parent of the associated axes, then the colorbar automatically updates to use the same parent.

`Children` — Children of colorbar

empty `GraphicsPlaceholder` array

The colorbar has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of colorbar object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The colorbar object handle is always visible.
- `'off'` — The colorbar object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

- `'callback'` — The colorbar object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the colorbar at the command-line, but allows callback functions to access it.

If the colorbar object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the colorbar. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The colorbar object — You can access properties of the colorbar object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the colorbar. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the colorbar when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the colorbar.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The Visible property must be set to `'on'`. The HitTest property determines if the colorbar responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the colorbar passes the click to the object below it in the current view of the figure window. The HitTest property of the colorbar has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the colorbar. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the colorbar that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the colorbar object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'off'` (default) | `'on'`

Callback interruption, specified as `'off'` or `'on'`. The `Interruptible` property determines if a running callback can be interrupted.

If the `ButtonDownFcn` callback of the colorbar is the running callback, then the `Interruptible` property determines if it can be interrupted by another callback. The `Interruptible` property has two possible values:

- `'off'` — The running callback cannot be interrupted. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.
- `'on'` — The running callback can be interrupted. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.

- If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

Example: `'off'`

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the colorbar tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the colorbar. Setting the `CreateFcn` property on an existing colorbar has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during colorbar creation. MATLAB executes the callback after creating the colorbar and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The colorbar object — You can access properties of the colorbar object from within the callback function. You also can access the colorbar object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the colorbar. MATLAB executes the callback before destroying the colorbar so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The colorbar object — You can access properties of the colorbar object from within the callback function. You also can access the colorbar object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of colorbar

`'off'` (default) | `'on'`

Deletion status of colorbar, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the colorbar begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the colorbar no longer exists.

Check the value of the BeingDeleted property to verify that the colorbar is not about to be deleted before querying or modifying it.

See Also

`colorbar`

More About

- "Access Property Values"
- "Graphics Object Properties"

colordef

Set default property values to display different color schemes

Syntax

```
colordef white
colordef black
colordef none
colordef(fig,color_option)
h = colordef('new',color_option)
```

Description

`colordef` enables you to select either a white or black background for graphics display. It sets axis lines and labels so that they contrast with the background color.

`colordef white` sets the axis background, axis lines and labels, and the figure background to the default system colors.

`colordef black` sets the axis background color to black, the axis lines and labels to white, and the figure background color to dark gray.

`colordef none` sets the figure coloring to that used by MATLAB Version 4. The most noticeable difference is that the axis background is set to `'none'`, making the axis background and figure background colors the same. The figure background color is set to black.

`colordef(fig,color_option)` sets the color scheme of the figure identified by the handle `fig` to one of the color options `'white'`, `'black'`, or `'none'`. When you use this syntax to apply `colordef` to an existing figure, the figure must have no graphic content. If it does, you should first clear it (via `clf`) before using this form of the command.

`h = colordef('new',color_option)` returns the handle to a new figure created with the specified color options (i.e., `'white'`, `'black'`, or `'none'`). This form of the command is useful for creating GUIs when you may want to control the default environment. The figure is created with `'visible','off'` to prevent flashing.

More About

Tips

`colordef` affects only subsequently drawn figures, not those currently on the display. This is because `colordef` works by setting default property values (on the root or figure level). You can list the currently set default values on the root level with the statement

```
get(groot,'Default')
```

You can remove all default values using the `reset` command:

```
reset(groot)
```

See the `get` and `reset` references pages for more information.

See Also

`whitebg` | `clf`

Introduced before R2006a

colormap

View and set current colormap

Syntax

```
colormap name
colormap default

colormap(map)
colormap(fig,map)
colormap(ax,map)

colormap(fig,'default')
colormap(ax,'default')

cmap = colormap
cmap = colormap(fig)
cmap = colormap(ax)
```

Description

`colormap name` sets the colormap for the current figure to the built-in colormap specified by name. The new colormap uses the same number of colors as the current colormap. The figure colormap affects all axes in the figure, unless you set an axes colormap separately. For more information, see “What Is a Colormap?” on page 1-1354.

`colormap default` uses the default colormap, which is the `parula` colormap with 64 colors. Versions of MATLAB prior to R2014b use `jet` as the default.

`colormap(map)` sets the colormap for the current figure to the colormap specified by `map`. The figure colormap affects all axes in the figure, unless you set an axes colormap separately. Use this syntax if you want to use a built-in colormap with a specific number of colors or if you want to use a custom colormap.

- For a built-in colormap with a specific number of colors, specify `map` as one of the built-in colormap functions and pass it an integer value as an input argument. For

example, `colormap(summer(10))` uses 10 colors from the `summer` colormap. If you do not specify the number of colors, such as `colormap(summer)`, then the colormap contains the same number of colors as the current colormap.

- For a custom colormap, specify `map` as a three-column matrix of values in the range `[0,1]` where each row is an RGB triplet that defines one color of the colormap. An RGB triplet is a three-element row vector specifying the red, green, and blue intensities for a color.

`colormap(fig,map)` sets the colormap for the figure specified by `fig`.

`colormap(ax,map)` sets the colormap for the axes specified by `ax`. Each axes within a figure can have a unique colormap. After you set an axes colormap, changing the figure colormap does not affect the axes.

`colormap(fig,'default')` sets the colormap for the figure specified by `fig` to the default colormap.

`colormap(ax,'default')` sets the colormap for the axes specified by `ax` to the default colormap.

`cmap = colormap` returns the three-column matrix of RGB triplets defining the colormap for the current figure.

`cmap = colormap(fig)` returns the colormap for the figure specified by `fig`.

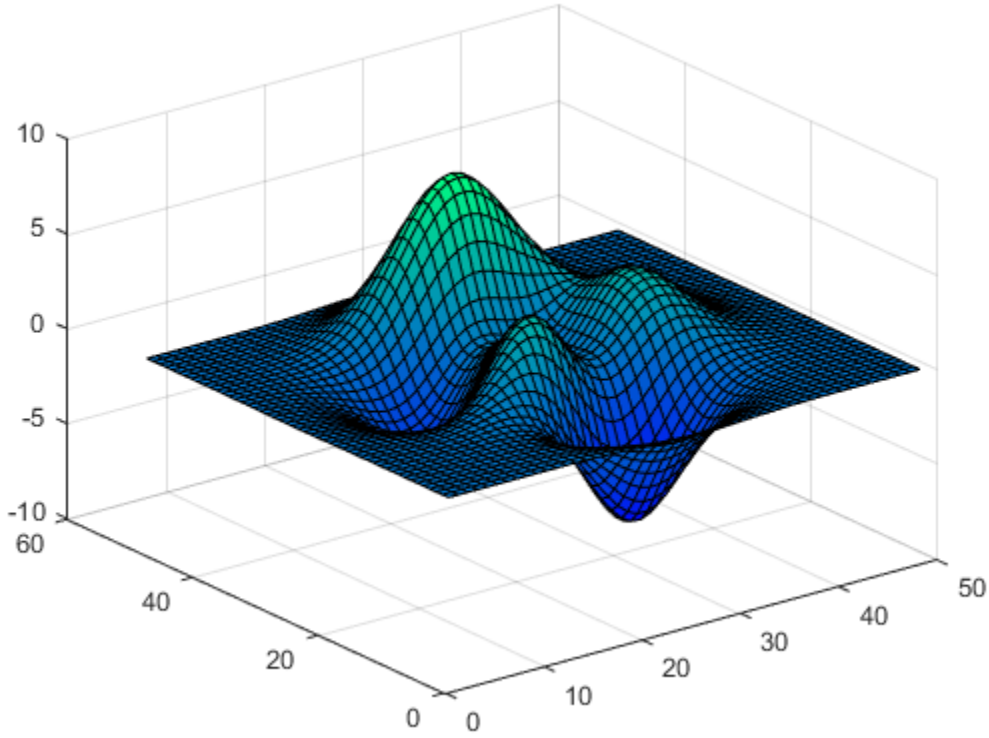
`cmap = colormap(ax)` returns the colormap for the axes specified by `ax`.

Examples

Change Colormap for Figure

Create a surface plot and set the colormap to `winter`.

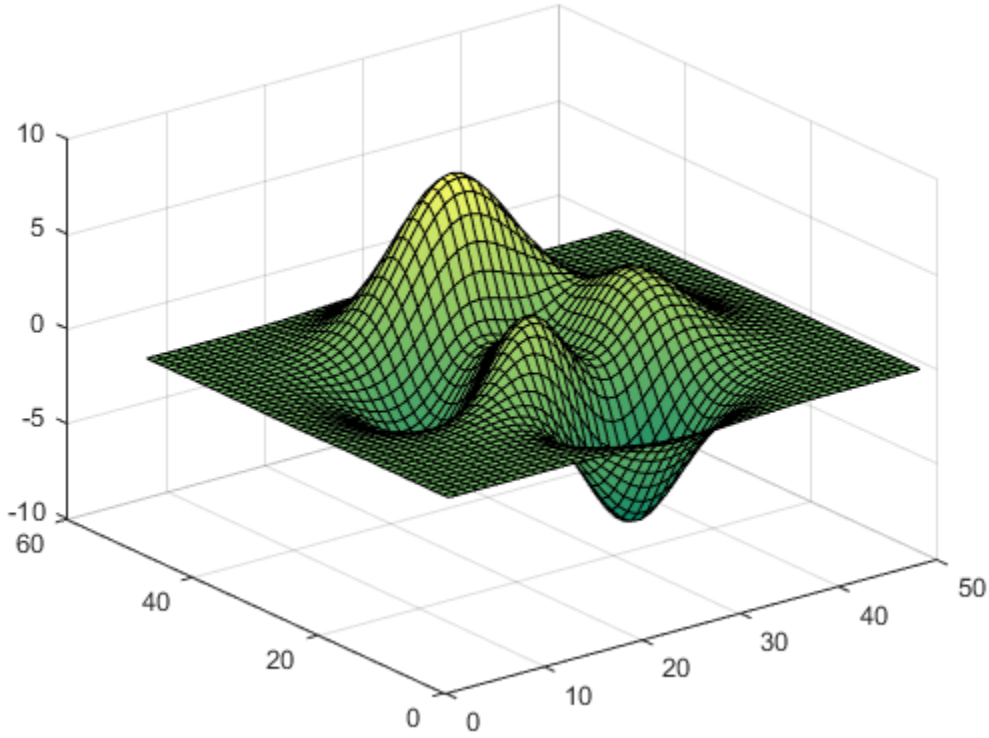
```
figure
surf(peaks)
colormap winter
```



Set Colormap Back to Default

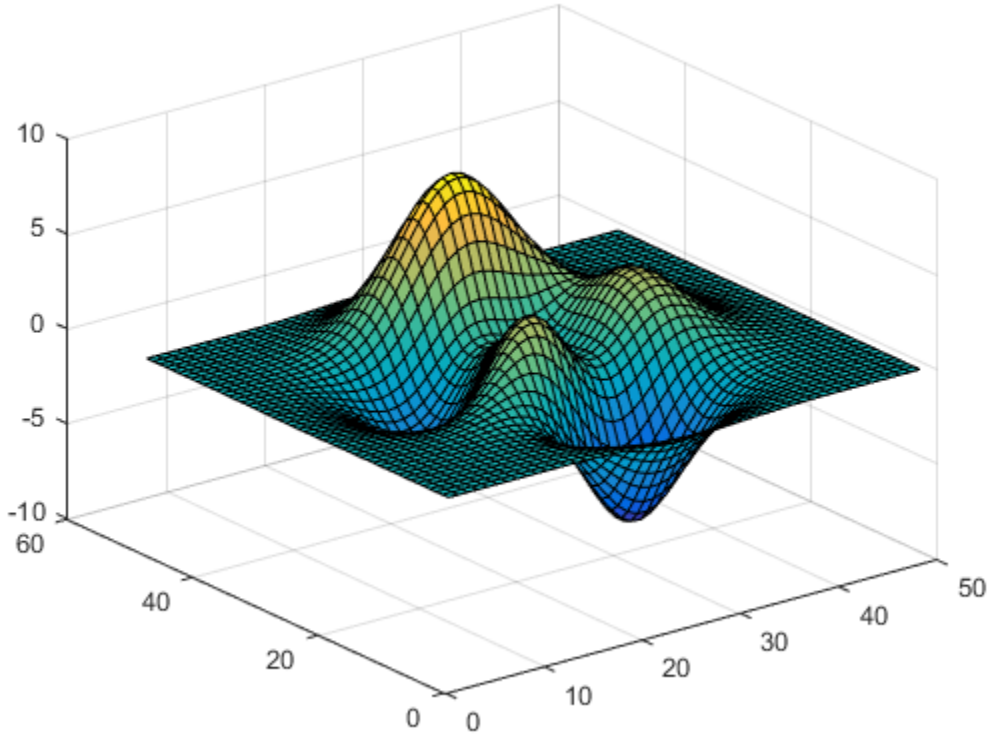
First, change the colormap for the current figure to `summer`.

```
figure
surf(peaks)
colormap summer
```



Now set the colormap back to your system's default value. If you have not specified a different default value, then the default colormap is `parula`.

```
colormap default
```

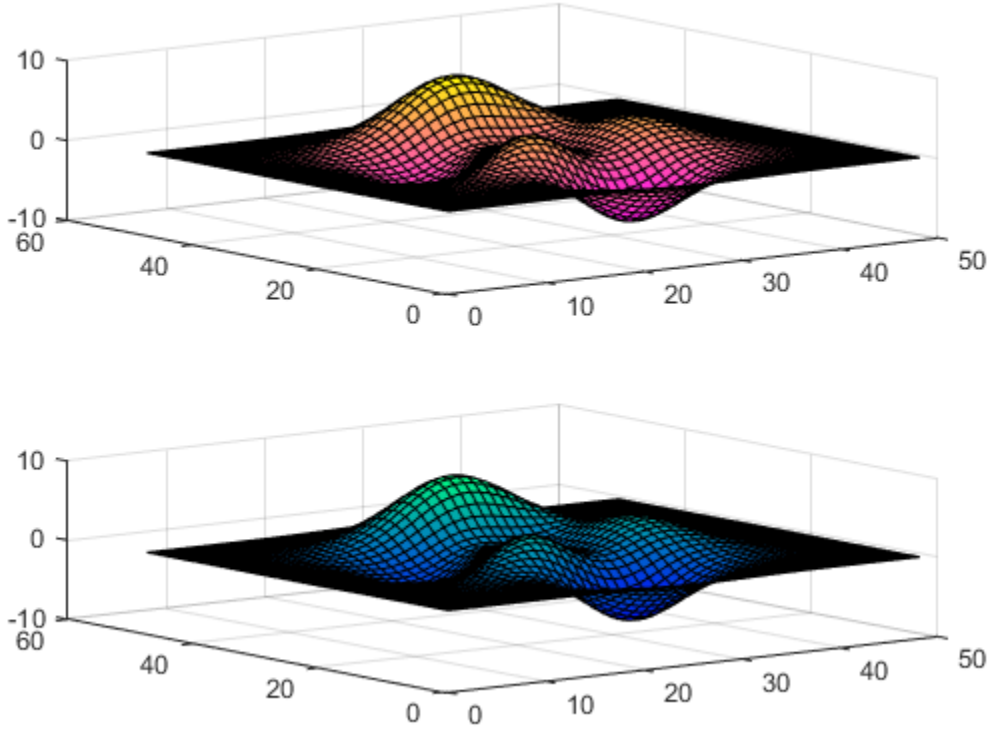


Use Different Colormaps for Each Axes in Figure

Create a figure with two subplots and store the axes handles, `ax1` and `ax2`. Use a different colormap for each axes by passing the axes handles to the `colormap` function. In the upper subplot, create a surface plot using the `spring` colormap. In the lower subplot, create a surface plot using the `winter` colormap.

```
figure
ax1 = subplot(2,1,1);
surf(peaks)
colormap(ax1,spring)

ax2 = subplot(2,1,2);
surf(peaks)
```

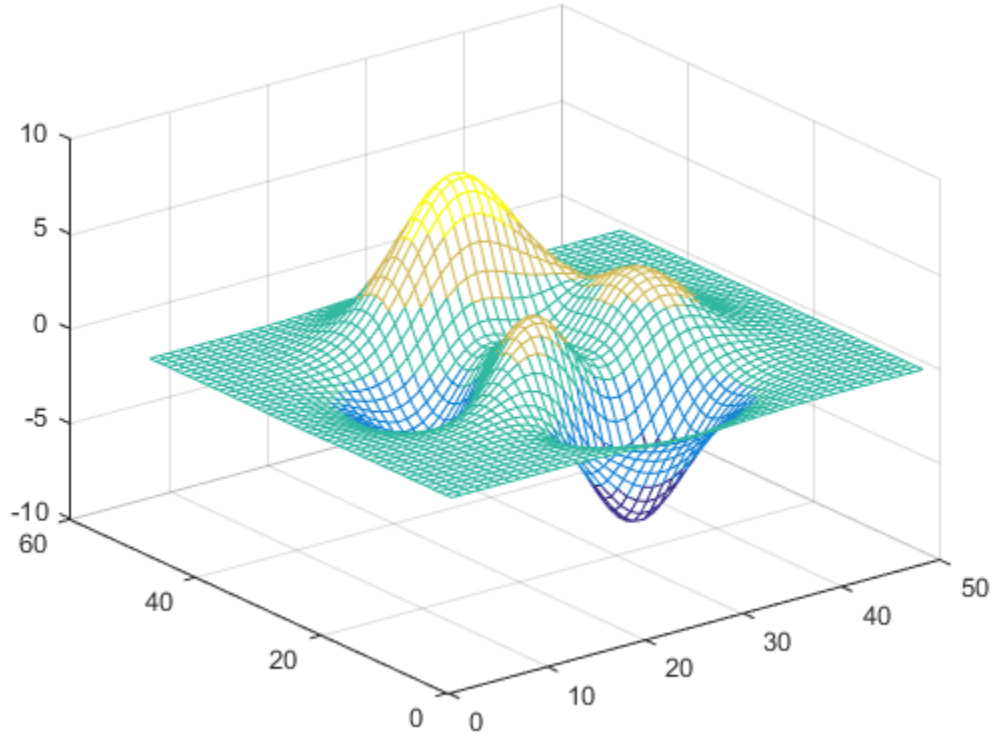
```
colormap(ax2,winter)
```



Specify Number of Colors for Colormap

Specify the number of colors used in a colormap by passing an integer as an input argument to the built-in colormap. Use five colors from the parula colormap.

```
figure
mesh(peaks)
colormap(parula(5))
```



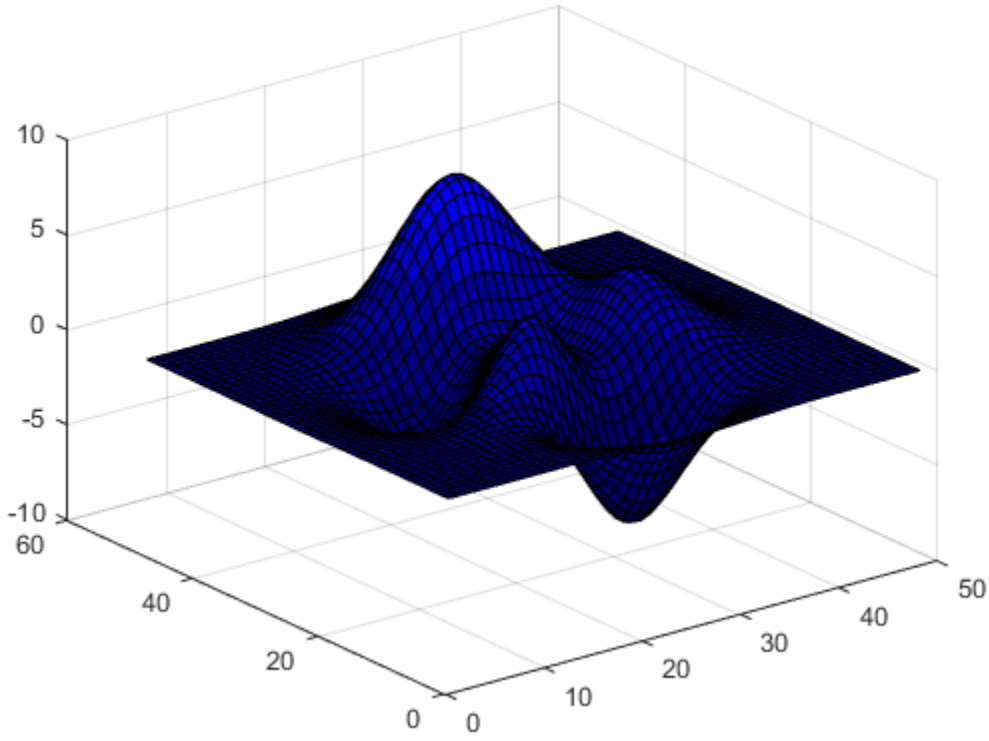
Create Custom Colormap

Create a custom colormap by defining a three-column matrix of values between 0.0 and 1.0. Each row defines a three-element RGB triplet. The first column specifies the red intensities. The second column specifies the green intensities. The third column specifies the blue intensities.

Use a colormap of blue values by setting the first two columns to zeros.

```
map = [0, 0, 0.3
    0, 0, 0.4
    0, 0, 0.5
    0, 0, 0.6
    0, 0, 0.8
```

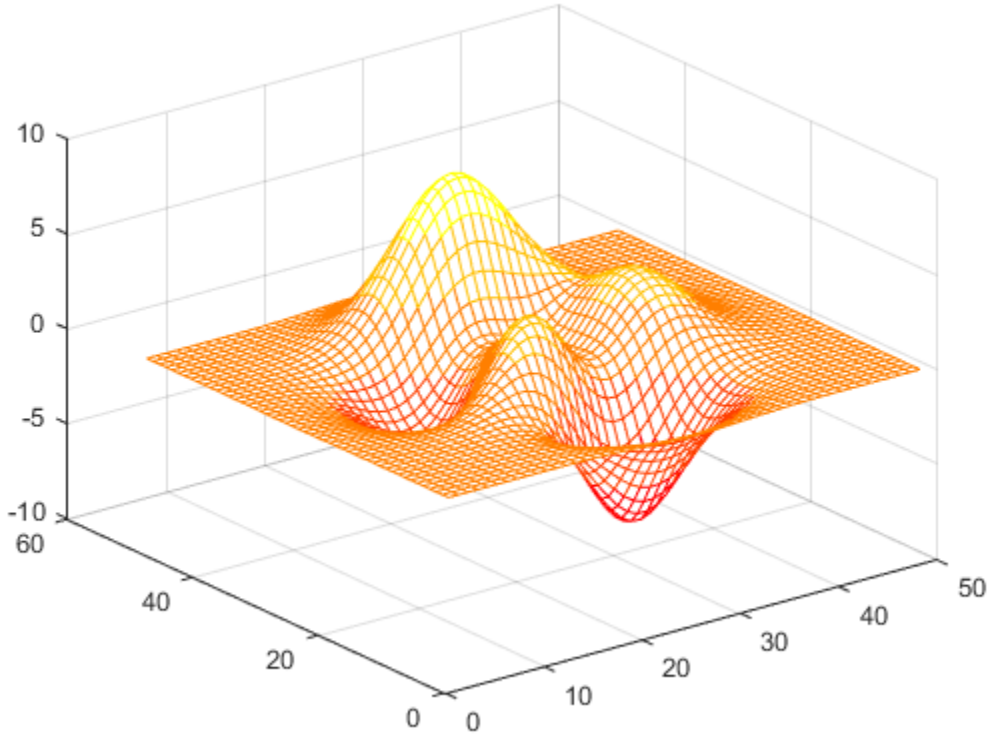
```
    0, 0, 1.0];

figure
surf(peaks)
colormap(map)
```



Return Colormap Values Used in Plot

Create a surface plot of the `peaks` function and specify a colormap.

```
figure
mesh(peaks)
colormap(autumn(5))
```



Return the three-column matrix of values that define the colors used in the plot. Each row is an RGB triplet color value that specifies one color of the colormap.

```
cmap = colormap
```

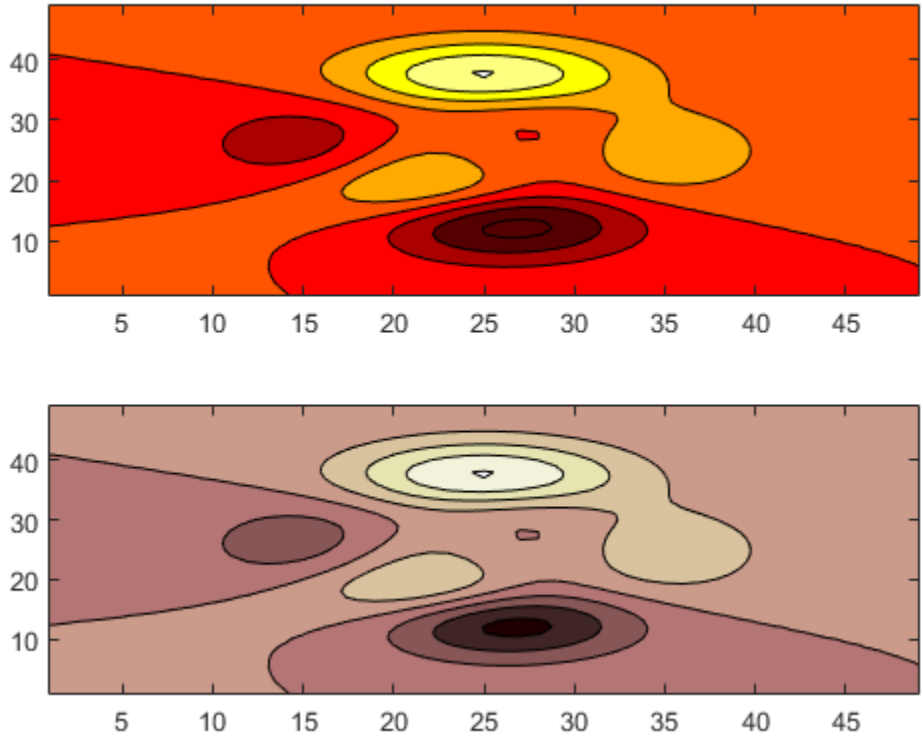
```
cmap =

    1.0000         0         0
    1.0000    0.2500         0
    1.0000    0.5000         0
    1.0000    0.7500         0
    1.0000    1.0000         0
```

Return Colormap Values for Specific Axes

Return the colormap values for a specific axes by passing its axes handle to the `colormap` function.

Create a figure with two subplots and return the axes handles, `ax1` and `ax2`. Add a filled contour plot to each axes and use a different colormap for each axes.

```
figure
ax1 = subplot(2,1,1);
contourf(peaks)
colormap(ax1,hot(8));

ax2 = subplot(2,1,2);
contourf(peaks)
colormap(ax2,pink);
```



Return the colormap values used in the upper subplot by passing its axes handle, `ax1`, to the `colormap` function. Each row is an RGB triplet color value that specifies one color of the colormap.

```
cmap = colormap(ax1)
```

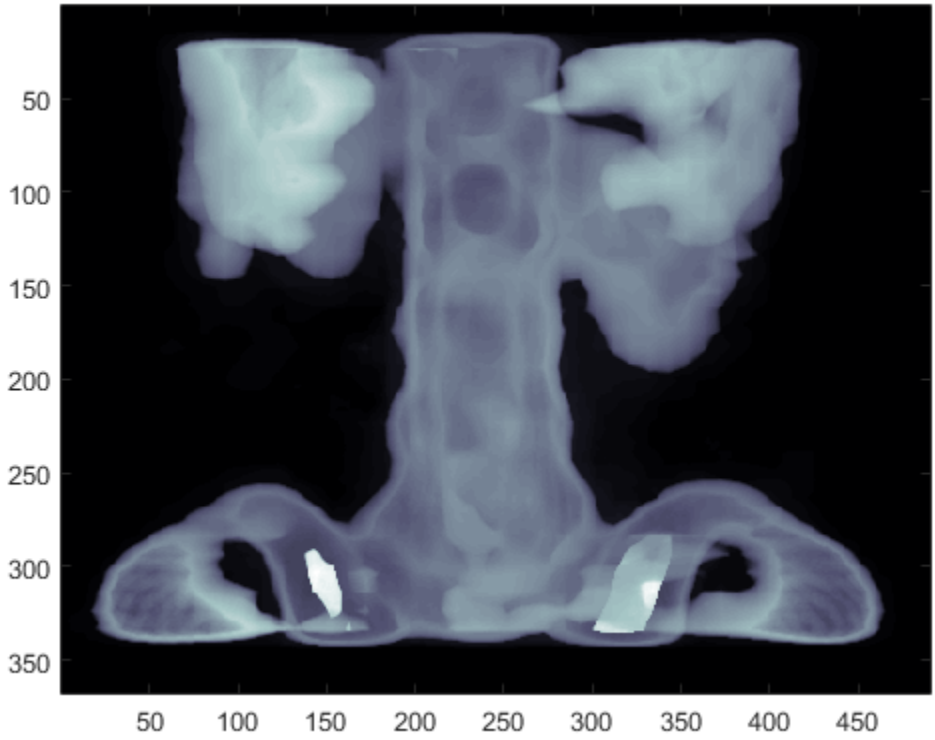
```
cmap =

    0.3333         0         0
    0.6667         0         0
    1.0000         0         0
    1.0000    0.3333         0
    1.0000    0.6667         0
```

```
    1.0000    1.0000         0
    1.0000    1.0000    0.5000
    1.0000    1.0000    1.0000
```

Change Colormap for Figure with Image

Load the `spine` data set that returns the image `X` and its associated colormap `map`. Display `X` using the `image` function and set the colormap to `map`.

```
load spine
figure
image(X)
colormap(map)
```



- “Change Mapping of Data Values into the Colormap”

Input Arguments

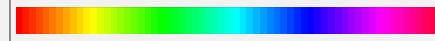
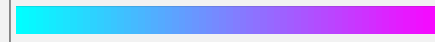
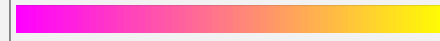
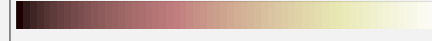
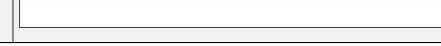
`name` — Built-in colormap

`parula` (default) | built-in colormap

Built-in colormap, specified as a one of the colormaps listed in the table. The default colormap is `parula`. Versions of MATLAB prior to R2014b use `jet` as the default.

The `colormap` function applies the new colormap with the same number of colors as the current colormap. To change the number of colors, use the map input argument.

This table lists the built-in colormaps.

Colormap Name	Color Scale
`parula`	
`jet`	
`hsv`	
`hot`	
`cool`	
`spring`	
`summer`	
`autumn`	
`winter`	
`gray`	
`bone`	
`copper`	
`pink`	
`lines`	
`colorcube`	
`prism`	
`flag`	
`white`	

`map` — Built-in or custom colormap

`parula` (default) | built-in colormap function | three-column matrix of RGB triplets

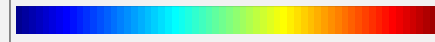
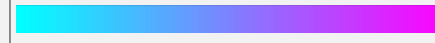
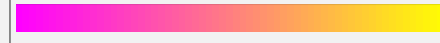
Built-in or custom colormap, specified as one of the built-in colormap functions or a three-column matrix of RGB triplets. The default colormap is `parula`. Versions of MATLAB prior to R2014b use `jet` as the default.

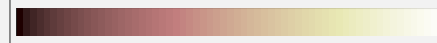
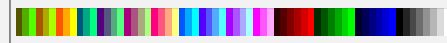
Built-In Colormap Functions

To use a built-in colormap with a specific number of colors, specify `map` as one of the built-in colormaps functions and pass it an integer value as an input argument. For example, `colormap(parula(12))` uses a map with 12 colors from the `parula` colormap. If you do not specify the number of colors, such as `colormap(parula)`, then MATLAB uses the same number of colors as the current colormap.

Note: Both the `name` and the `map` input arguments provide the same built-in colormap options. However, if you use the input argument `name`, then the `colormap` function processes `name` as a string and you cannot control the number of colors. If you use the input argument `map`, then the `colormap` function processes `map` as a function, which means that you can pass the built-in colormap an integer value to control the number of colors.

This table lists the built-in colormaps functions.

Colormap Name	Color Scale
`parula`	
`jet`	
`hsv`	
`hot`	
`cool`	
`spring`	
`summer`	
`autumn`	
`winter`	
`gray`	
`bone`	
`copper`	

Colormap Name	Color Scale
`pink`	
`lines`	
`colorcube`	
`prism`	
`flag`	
`white`	

Custom Colormap

To create a custom colormap, specify `map` as a three-column matrix of RGB triplets where each row defines one color. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`. For example, this matrix defines a colormap containing five colors.

```
map = [0.2, 0.1, 0.5
    0.1, 0.5, 0.8
    0.2, 0.7, 0.6
    0.8, 0.7, 0.3
    0.9, 1, 0];
```

This table lists the RGB triplet values for common colors.

Color	RGB Triplet
yellow	`[1,1,0]`
magenta	`[1,0,1]`
cyan	`[0,1,1]`
red	`[1,0,0]`
green	`[0,1,0]`
blue	`[0,0,1]`
white	`[1,1,1]`
black	`[0,0,0]`

`fig` — Figure object

figure object

Figure object. The figure colormap affects plots for all axes within the figure, unless you specify an axes colormap separately.

Each figure has a `Colormap` property. When you change the figure colormap using the `colormap` function, MATLAB updates the `Colormap` property of the figure.

`ax` — Axes object

axes object

Axes object. You can define a unique colormap for each axes. After you set an axes colormap, changing the figure colormap does not affect the axes.

Output Arguments

`cmap` — Colormap values

three-column matrix of RGB triplets

Colormap values, returned as a three-column matrix of RGB triplets. Each row of the matrix defines one RGB triplet that specifies one color of the colormap. The values are in the range `[0,1]`.

More About

What Is a Colormap?

A colormap is matrix of values between `0` and `1` that define the colors for graphics objects such as surface, image, and patch objects. MATLAB draws the objects by mapping data values to colors in the colormap.

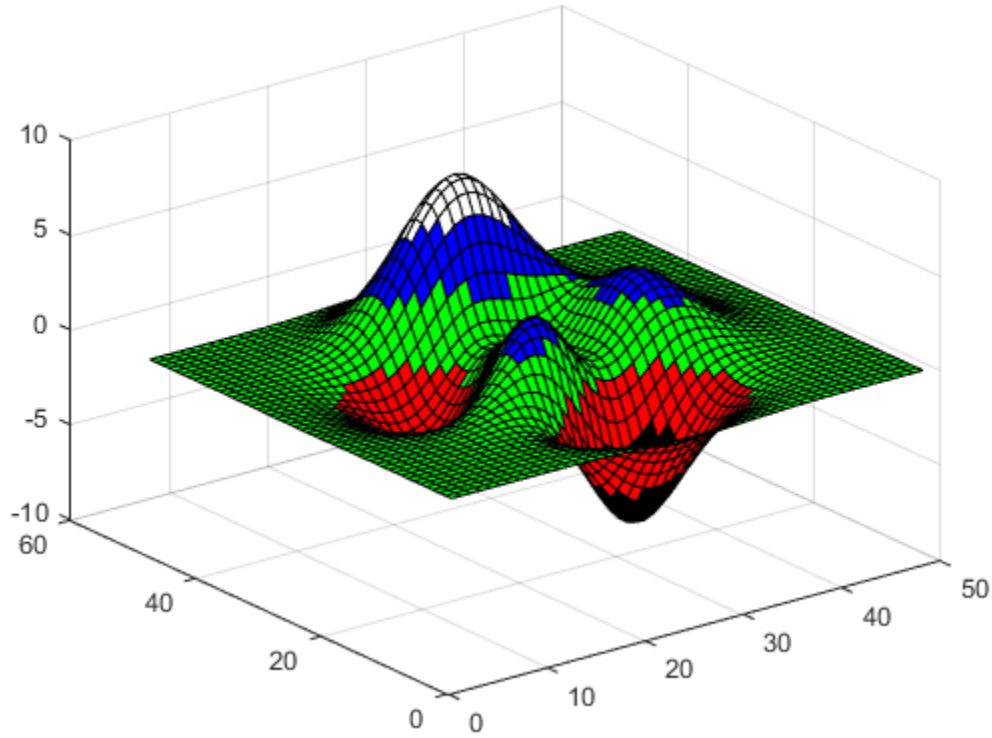
Colormaps can be any length, but must be three columns wide. Each row in the matrix defines one color using an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`. A value of `0` indicates no color and a value of `1` indicates full intensity. For example, this is a colormap with five colors: black, red, green, blue, and white.

```
mymap = [0 0 0
    1 0 0
```

```
    0 1 0
    0 0 1
    1 1 1];
```

Colormaps are associated with axes and figures. Use the `colormap` function to change the colormap for a particular axes or figure. For example, create a surface plot and use `mymap` as the colormap for the figure.

```
surf(peaks)
colormap(mymap)
```



Tips

- To control the mapping of data values into the colormap, use the `caxis` function.

See Also

`brighten` | `caxis` | `colorbar` | `contrast` | `hsv2rgb` | `imwrite` | `ind2rgb` | `rgbplot`

Introduced before R2006a

colormapeditor

Open colormap editor

Syntax

```
colormapeditor
```

Description

`colormapeditor` displays the colormap for the current axes as a strip of rectangular cells in the colormap editor. Node pointers are colored cells below the colormap strip that indicate points in the colormap where the rate of the variation of R, G, and B values changes. You can also work in the HSV colorspace by setting the **Interpolating Colorspace** selector to HSV.

You can also start the colormap editor by selecting **Colormap** from the **Edit** menu.

Node Pointer Operations

You can select and move node pointers to change a range of colors in the colormap. The color of a node pointer remains constant as you move it, but the colormap changes by linearly interpolating the RGB values between nodes.

Change the color at a node by double-clicking the node pointer. A color picker box appears, from which you can select a new color. After you select a new color at a node, the colors between nodes are reinterpolated.

You can select a different color map using the **Standard Colormaps** submenu of the **Tools** menu. The Plotting Tools Property Editor has a dropdown menu that also lets you select from standard colormaps, but does not help you to modify a colormap.

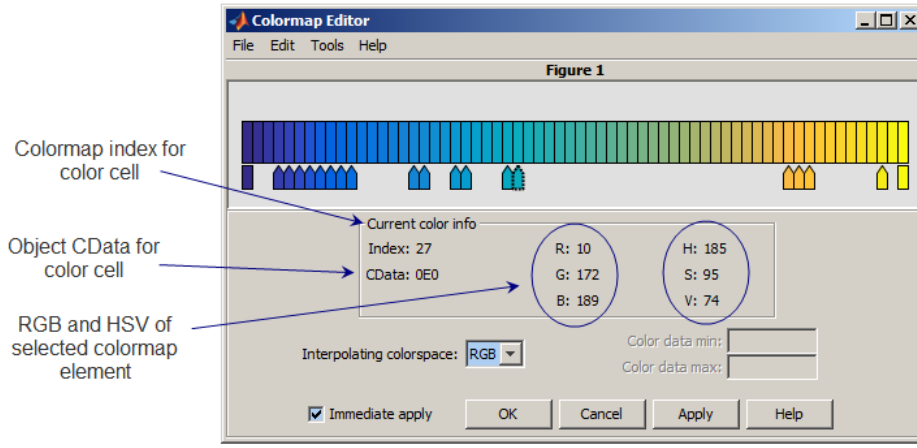
Operation	How to Perform
Select a built-in colormap	**Tools > Standard Colormaps**

Operation	How to Perform
Add a node	Click below the corresponding cell in the colormap strip.
Select a node	Left-click the node.
Select multiple nodes	Adjacent: left-click first node, **Shift+click** the last node. Nonadjacent: left-click first node, **Ctrl+click** subsequent nodes.
Move a node	Select and drag with the mouse or select and use the left and right arrow keys.
Move multiple nodes	Select multiple nodes and use the left and right arrow keys to move nodes as a group. Movement stops when one of the selected nodes hits an unselected node or an end node.
Delete a node	Select the node and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Delete multiple nodes	Select the nodes and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Display color picker for a node	Double-click the node pointer.

Current Color Info

When you put the mouse over a color cell or node pointer, the colormap editor displays the following information about that colormap element:

- The element's index in the colormap
- The value from the graphics object color data that is mapped to the node's color (i.e., data from the `CData` property of any image, patch, or surface objects in the figure)
- The color's RGB and HSV color value



Interpolating Colorspace

The colorspace determines what values are used to calculate the colors of cells between nodes. For example, in the RGB colorspace, internode colors are calculated by linearly interpolating the red, green, and blue intensity values from one node to the next. Switching to the HSV colorspace causes the colormap editor to recalculate the colors between nodes using the hue, saturation, and value components of the color definition.

Note that when you switch from one colorspace to another, the color editor preserves the number, color, and location of the node pointers, which can cause the colormap to change.

Interpolating in HSV. Since hue is conceptually mapped about a color circle, the interpolation between hue values can be ambiguous. To minimize this ambiguity, the interpolation uses the shortest distance around the circle. For example, interpolating between two nodes, one with hue of 2 (slightly orange red) and another with a hue of 356 (slightly magenta red), does not result in hues 3,4,5...353,354,355 (orange/red-yellow-green-cyan-blue-magenta/red). Taking the shortest distance around the circle gives 357,358,1,2 (orange/red-red-magenta/red).

Color Data Min and Max

The **Color Data Min** and **Color Data Max** text fields enable you to specify values for the axes `CLim` property. These values change the mapping of object color data (the `CData` property of images, patches, and surfaces) to the colormap.

Examples

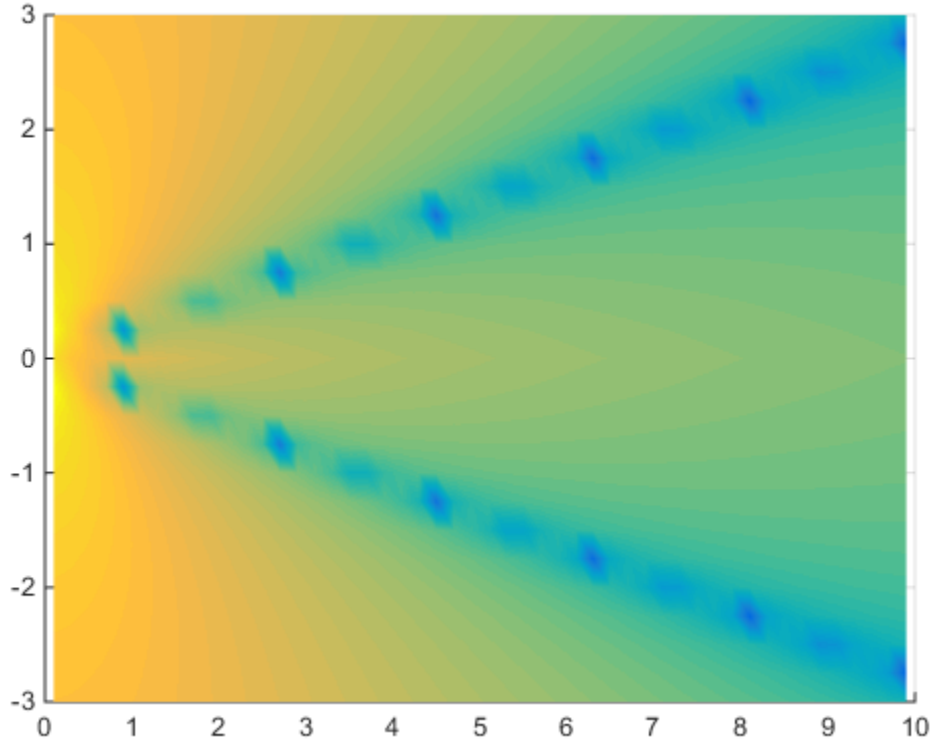
This example modifies a default MATLAB colormap so that ranges of data values are displayed in specific ranges of color. The graph is a slice plane illustrating a cross section of fluid flow through a jet nozzle. See the `slice` reference page for more information on this type of graph.

Enhance Colormap

This example modifies a default MATLAB colormap so that ranges of data values are displayed more prominently.

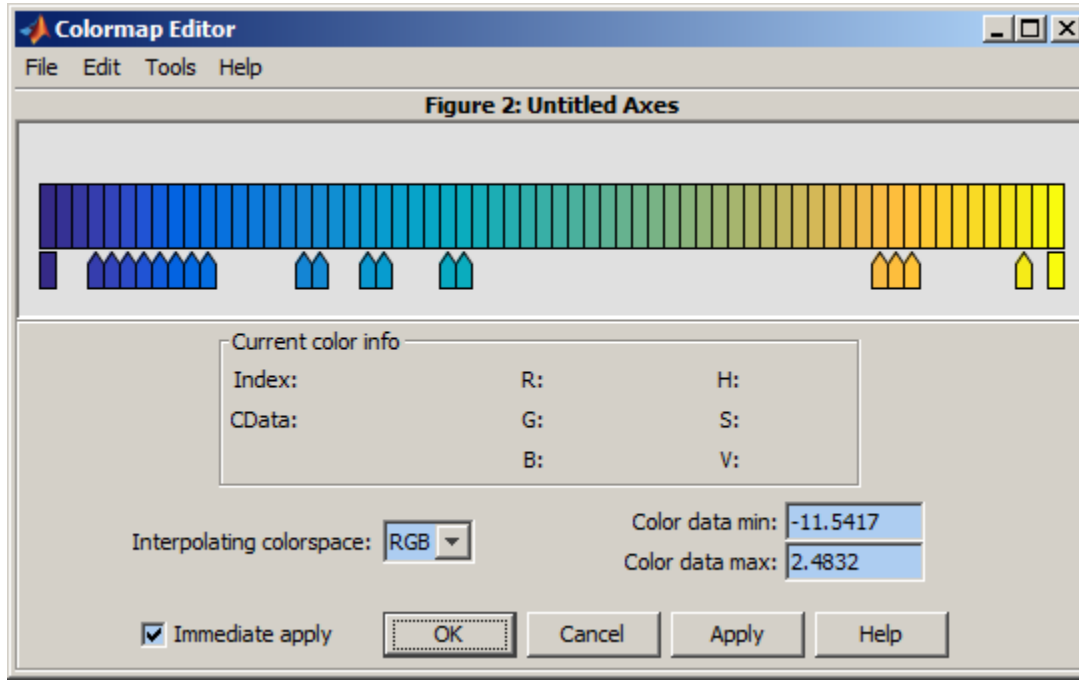
Create a slice plane for the flow dataset:

```
[x,y,z,v] = flow;
hz = slice(x,y,z,v,[],[],0);
hz.EdgeColor = 'none';
hz.FaceColor = 'interp';
view(2)
```

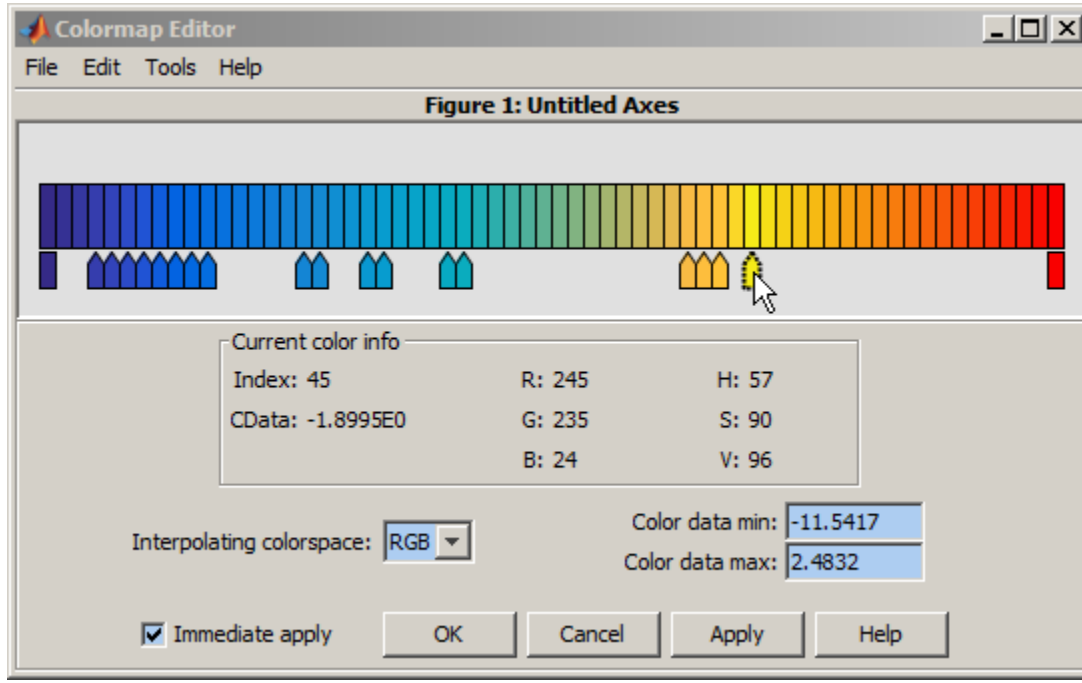


Start the colormap editor:

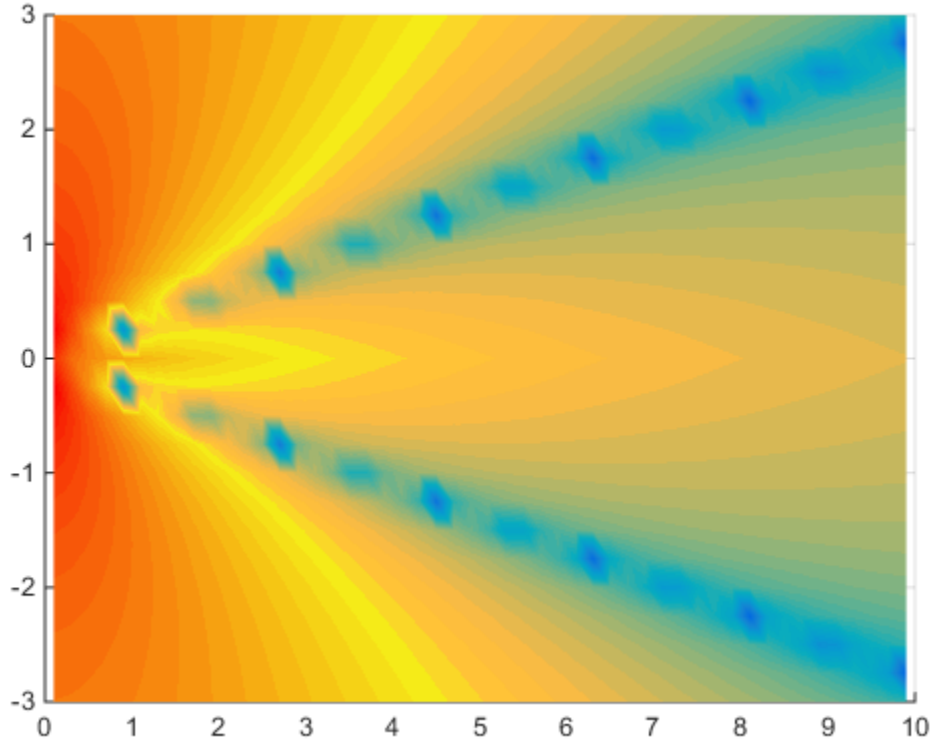
```
colormapeditor
```



To reveal more information about the flow data, create more color variation in the region of interest. Add the color red for the largest color data value and blend it with the yellow and orange colors:



The new colormap reveals more of the flow data variation in the higher data values:



Saving the Modified Colormap

You can save the modified colormap using the `colormap` function.

After you have applied your changes, save the current axes colormap in a variable:

```
mycmap = colormap(ax);
```

To use this colormap in another axes, set the colormap for the axes to `mycmap`:

```
colormap(ax_new,mycmap)
```

To save your modified colormap in a MAT-file, use the `save` command to save the `mycmap` workspace variable:

```
save('MyColormaps','mycmap')
```

To use your saved colormap in another MATLAB session, `load` the variable into the workspace and assign the colormap to the axes:

```
load('MyColormaps','mycmap')
colormap(ax,mycmap)
```

See Also

`colormap` | `save` | `set` | `get` | `load` | `propertyeditor`

Introduced before R2006a

ColorSpec (Color Specification)

Color specification

Description

`ColorSpec` is not a function; it refers to the three ways in which you specify color for MATLAB graphics:

- RGB triple
- Short name
- Long name

The short names and long names are MATLAB strings that specify one of eight predefined colors. The RGB triple is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1]. The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Examples

To change the background color of a figure to green, specify the color with a short name, a long name, or an RGB triple. These statements generate equivalent results:

```
whitebg('g')
whitebg('green')
whitebg([0 1 0]);
```

You can use `ColorSpec` anywhere you need to define a color. For example, this statement changes the figure background color to pink:

```
set(gcf,'Color',[1,0.4,0.6])
```

More About

Tips

The eight predefined colors and any colors you specify as RGB values are not part of a figure's colormap, nor are they affected by changes to the figure's colormap. They are referred to as *fixed* colors, as opposed to *colormap* colors.

Some high-level functions (for example, `scatter`) accept a colorspec as an input argument and use it to set the `CData` of graphic objects they create. When using such functions, take care not to specify a colorspec in a property/value pair that sets `CData`; values for `CData` are always n-length vectors or n-by-3 matrices, where n is the length of `XData` and `YData`, never strings.

See Also

`bar` | `bar3` | `colordef` | `fill` | `fill3` | `colormap` | `whitebg` | `uisetcolor`

colperm

Sparse column permutation based on nonzero count

Syntax

```
j = colperm(S)
```

Description

`j = colperm(S)` generates a permutation vector `j` such that the columns of `S(:,j)` are ordered according to increasing count of nonzero entries. This is sometimes useful as a preordering for LU factorization; in this case use `lu(S(:,j))`.

If `S` is symmetric, then `j = colperm(S)` generates a permutation `j` so that both the rows and columns of `S(j,j)` are ordered according to increasing count of nonzero entries. If `S` is positive definite, this is sometimes useful as a preordering for Cholesky factorization; in this case use `chol(S(j,j))`.

Examples

The `n`-by-`n` *arrowhead* matrix

```
A = [ones(1,n); ones(n-1,1) speye(n-1,n-1)]
```

has a full first row and column. Its LU factorization, `lu(A)`, is almost completely full. The statement

```
j = colperm(A)
```

returns `j = [2:n 1]`. So `A(j,j)` sends the full row and column to the bottom and the rear, and `lu(A(j,j))` has the same nonzero structure as `A` itself.

On the other hand, the Bucky ball example,

```
B = bucky
```

has exactly three nonzero elements in each row and column, so `j = colperm(B)` is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.

More About

Algorithms

The algorithm involves a sort on the counts of nonzeros in each column.

See Also

`chol` | `colamd` | `lu` | `spparms` | `symamd` | `symrcm`

Introduced before R2006a

Combine

Convenience function for static .NET System.Delegate Combine method

Syntax

```
result = Combine(delegateA,delegateB)
```

Description

`result = Combine(delegateA,delegateB)` combines two delegates into a new delegate.

Input Arguments

`delegateA`

.NET `System.Delegate` object. The first delegate in the new delegate.

Default:

`delegateB`

.NET `System.Delegate` object. The last delegate in the new delegate.

Default:

Output Arguments

`result`

.NET `System.Delegate` object. A new delegate that delegates to the input delegate `delegateA`, then `delegateB`

Alternatives

Use the static `Combine` method of the `System.Delegate` class.

More About

- "Combine and Remove .NET Delegates"
- MSDN System.Delegate.Combine Method reference page

See Also

`Remove` | `RemoveAll`

Introduced in R2011a

comet

2-D comet plot

Syntax

```
comet(y)
comet(x,y)
comet(x,y,p)
comet(axes_handle,...)
```

Description

`comet(y)` displays a comet graph of the vector `y`. A comet graph is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

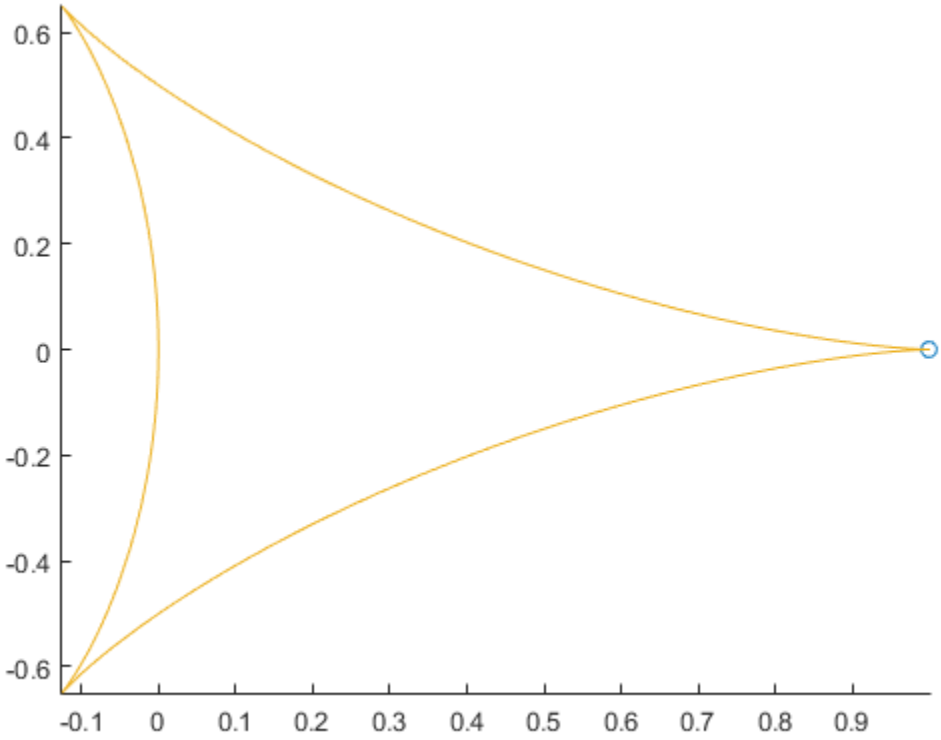
`comet(x,y)` displays a comet graph of vector `y` versus vector `x`.

`comet(x,y,p)` specifies a comet body of length `p*length(y)`. `p` defaults to `0.1`.

`comet(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

Create Comet Graph

```
t = 0:.01:2*pi;
x = cos(2*t).*(cos(t).^2);
y = sin(2*t).*(sin(t).^2);
comet(x,y);
```



More About

Tips

- Comet graphs do not support data tips.

See Also

`comet3` | `animatedline`

Introduced before R2006a

comet3

3-D comet plot

Syntax

```
comet3(z)
comet3(x,y,z)
comet3(x,y,z,p)
comet3(axes_handle,...)
```

Description

A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet3(z)` displays a 3-D comet graph of the vector `z`.

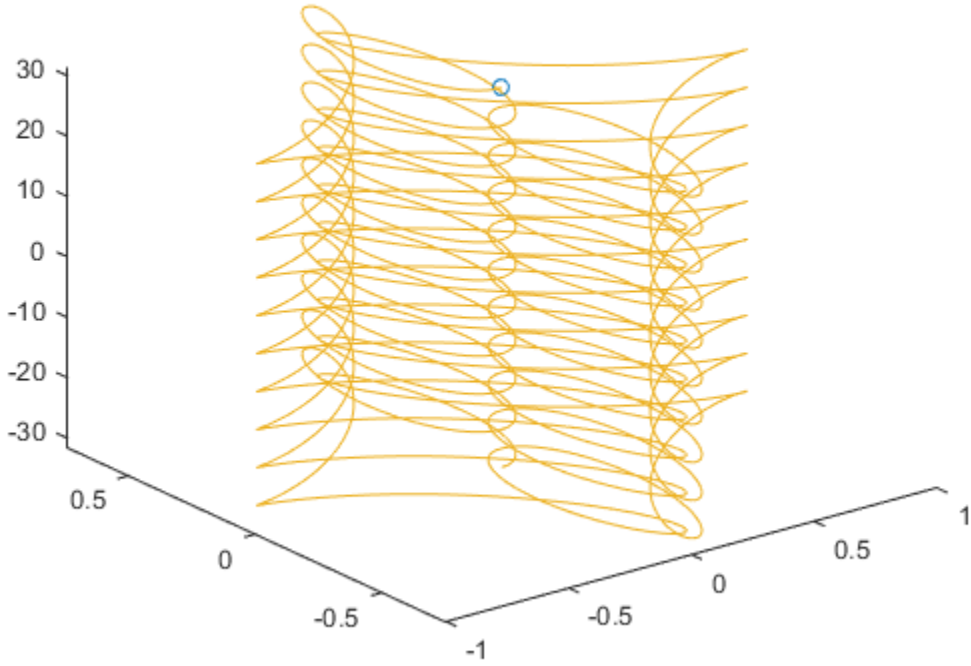
`comet3(x,y,z)` displays a comet graph of the curve through the points `[x(i),y(i),z(i)]`.

`comet3(x,y,z,p)` specifies a comet body of length `p*length(y)`. `p` must be between `0` and `1`.

`comet3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

Create 3-D Comet Graph

```
t = -10*pi:pi/250:10*pi;
x = (cos(2*t).^2).*sin(t);
y = (sin(2*t).^2).*cos(t);
comet3(x,y,t);
```



See Also

`comet` | `animatedline`

Introduced before R2006a

commandhistory

Open Command History window, or select it if already open

Syntax

```
commandhistory
```

Description

`commandhistory` opens the MATLAB Command History window when it is closed, and selects the Command History window when it is open. The Command History window presents a log of the statements most recently run in the Command Window.

More About

- "Command History"

See Also

`diary` | `prefdir` | `startup`

Introduced before R2006a

commandwindow

Open Command Window, or select it if already open

Syntax

```
commandwindow
```

Description

`commandwindow` opens the MATLAB Command Window when it is closed, and selects the Command Window when it is open.

More About

Tips

To determine the number of columns and rows that display in the Command Window, given its current size, use

```
matlab.desktop.commandwindow.size
```

The number of columns is based on the width of the Command Window. With the matrix display width preference set to 80 columns, the number of columns is always 80.

- "Optimize Desktop Layout for Limited Screen Space"
- "Set Command Window Preferences"

See Also

`commandhistory` | `input` | `inputdlg`

Introduced before R2006a

compan

Companion matrix

Syntax

```
A = compan(u)
```

Description

`A = compan(u)` returns the corresponding companion matrix whose first row is `-u(2:n)/u(1)`, where `u` is a vector of polynomial coefficients. The eigenvalues of `compan(u)` are the roots of the polynomial.

Examples

The polynomial $(x-1)(x-2)(x+3) = x^3 - 7x + 6$ has a companion matrix given by

```
u = [1  0  -7  6]
A = compan(u)
A =
    0    7   -6
    1    0    0
    0    1    0
```

The eigenvalues are the polynomial roots:

```
eig(compan(u))

ans =
   -3.0000
    2.0000
    1.0000
```

This is also `roots(u)`.

See Also

`eig` | `poly` | `polyval` | `roots`

Introduced before R2006a

compass

Plot arrows emanating from origin

Syntax

```
compass(U,V)
compass(Z)
compass(...,LineSpec)
compass(axes_handle,...)
h = compass(...)
```

Description

A compass graph displays the vectors with components `(U,V)` as arrows emanating from the origin. `U`, `V`, and `Z` are in Cartesian coordinates and plotted on a circular grid.

`compass(U,V)` displays a compass graph having *n* arrows, where *n* is the number of elements in `U` or `V`. The location of the base of each arrow is the origin. The location of the tip of each arrow is a point relative to the base and determined by `[U(i),V(i)]`.

`compass(Z)` displays a compass graph having *n* arrows, where *n* is the number of elements in `Z`. The location of the base of each arrow is the origin. The location of the tip of each arrow is relative to the base as determined by the real and imaginary components of `Z`. This syntax is equivalent to `compass(real(Z),imag(Z))`.

`compass(...,LineSpec)` draws a compass graph using the line type, marker symbol, and color specified by `LineSpec`.

`compass(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

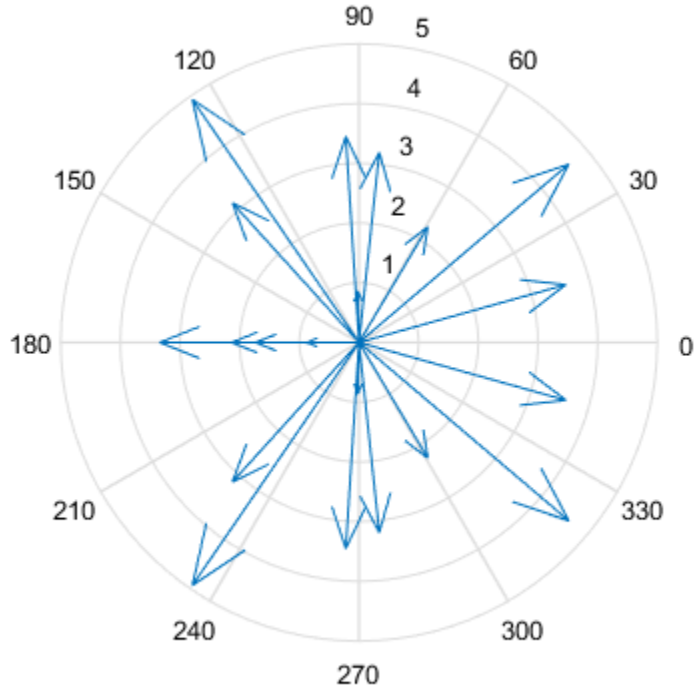
`h = compass(...)` returns handles to line objects.

Examples

Create Compass Graph

Create a compass graph of the eigenvalues of a random matrix.

```
rng(0,'twister') % initialize random number generator
M = randn(20,20);
Z = eig(M);

figure
compass(Z)
```



See Also

`feather` | `quiver` | `rose` | `polar` | `LineSpec`

Introduced before R2006a

complex

Create complex array

Syntax

```
z = complex(a,b)
z = complex(x)
```

Description

`z = complex(a,b)` creates a complex output, `z`, from two real inputs, such that `z = a + bi`.

The `complex` function provides a useful substitute for expressions, such as `a + 1i*b` or `a + 1j*b`, when

- `a` and `b` are not `double` or `single`
- `b` is all zeros

`z = complex(x)` returns the complex equivalent of `x`, such that `isreal(z)` returns logical `0` (`false`).

- If `x` is real, then `z` is `x + 0i`.
- If `x` is complex, then `z` is identical to `x`.

Examples

Complex Scalar from Two Real Scalars

Use the `complex` function to create the complex scalar, `3 + 4i`.

```
z = complex(3,4)
```

```
z =
```

```
   3.0000 + 4.0000i
```

Complex Vector from Two Complex Vectors

Create a complex `uint8` vector from two real `uint8` vectors.

```
a = uint8([1;2;3;4]);
b = uint8([2;2;7;7]);

z = complex(a,b)

z =

    1 +    2i
    2 +    2i
    3 +    7i
    4 +    7i
```

The size of `z`, 4-by-1, is the same as the size of the input arguments.

Complex Scalar from One Real Scalar

Create a complex scalar with zero imaginary part.

```
z = complex(12)

z =

  12.0000 + 0.0000i
```

Verify that `z` is complex.

```
isreal(z)

ans =

     0
```

Input Arguments

a — Real component

scalar | vector | matrix | multidimensional array

Real component, specified as a scalar, vector, matrix, or multidimensional array.

The size of `a` must match the size of `b`, unless one is a scalar. If either `a` or `b` is a scalar, MATLAB expands the scalar to match the size of the other input.

`a` and `b` must be the same data type with the following exceptions:

- `single` can combine with `double`.
- scalar `double` can combine with an integer data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

b — Imaginary component

scalar | vector | matrix | multidimensional array

Imaginary component, specified as a scalar, vector, matrix, or multidimensional array.

The size of `b` must match the size of `a`, unless one is a scalar. If either `a` or `b` is a scalar, MATLAB expands the scalar to match the size of the other input.

`a` and `b` must be the same data type with the following exceptions:

- `single` can combine with `double`.
- scalar `double` can combine with an integer data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

x — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Output Arguments

z — Complex array

scalar | vector | matrix | multidimensional array

Complex array, returned as a scalar, vector, matrix, or multidimensional array.

The size of `z` is the same as the input arguments.

The following describes the data type of `z`, when `a` and `b` have different data types.

- If either `a` or `b` is `single`, then `z` is `single`.
- If either `a` or `b` is an integer data type, then `z` is the same integer data type.

More About

Tips

- If `b` contains only zeros, then `z` is complex and the value of all its imaginary components is `0`. In contrast, the addition `a + 0i` returns a strictly real result.
- "Complex Numbers"

See Also

`abs` | `angle` | `conj` | `i` | `imag` | `isreal` | `j` | `real`

Introduced before R2006a

computeStrip

Class: Tiff

Index number of strip containing specified coordinate

Syntax

```
stripNumber = computeStrip(tiffobj,row)
stripNumber = computeStrip(tiffobj,row,plane)
```

Description

`stripNumber = computeStrip(tiffobj,row)` returns the index number of the strip containing the given row. The value of `row` must be one-based.

`stripNumber = computeStrip(tiffobj,row,plane)` returns the index number of the strip containing the given row in the specified plane, if the value of the `PlanarConfiguration` tag is `Tiff.PlanarConfiguration.Separate`. The values of row and plane must be one-based.

`computeStrip` clamps out-of-range coordinate values to the bounds of the image.

Examples

Determine Index Number of Strip

Determine the index number of the strip containing a row in the second image of a file.

Create a `Tiff` object associated with the example file, `example.tif`, and make the second image the current directory.

```
t = Tiff('example.tif','r');
setDirectory(t,2)
```

Get the number of rows in the image. Then, get the index of the strip containing the middle row.

```
numRows = getTag(t,'ImageLength');
stripNumber = computeStrip(t,numRows/2)
```

```
stripNumber =

     4
```

Close the `Tiff` object.

```
close(t)
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFComputeStrip` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.computeTile`

computeTile

Class: Tiff

Index number of tile containing specified coordinates

Syntax

```
tileNumber = computeTile(tiffobj,[row col])
tileNumber = computeTile(tiffobj,[row col],plane)
```

Description

`tileNumber = computeTile(tiffobj,[row col])` returns the index number of the tile containing the pixel specified by the one-based indices, `row` and `col`.

`tileNumber = computeTile(tiffobj,[row col],plane)` returns the index number of the tile containing the pixel specified by the indices in the specified plane, if the value of the `PlanarConfiguration` tag is `Tiff.PlanarConfiguration.Separate`. The row, column, and plane coordinate values are one-based.

`computeTile` clamps out-of-range coordinate values to the bounds of the image.

Examples

Get Index Number of Tile Containing Last Pixel

Open a `Tiff` object and get the dimensions of the image to calculate coordinates.

```
t = Tiff('example.tif','r');
numRows = getTag(t,'ImageLength');
numCols = getTag(t,'ImageWidth');
```

Get the ID number of the tile containing the coordinates.

```
tileNum = computeTile(t,[numRows numCols])
```

```
tileNum =

   110
```

Close the `Tiff` object.

```
close(t)
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

References

This method corresponds to the `TIFFComputeTile` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTiff - TIFF Library and Utilities.

See Also

`Tiff.computeStrip`

computer

Information about computer on which MATLAB software is running

Syntax

```
str = computer
archstr = computer('arch')
[str,maxsize] = computer
[str,maxsize,endian] = computer
```

Description

`str = computer` returns the string `str` with the computer type on which MATLAB is running.

`archstr = computer('arch')` returns the string `archstr` which is used by the `mex` command and standalone applications to locate MATLAB library files.

`[str,maxsize] = computer` returns the integer `maxsize`, the maximum number of elements allowed in an array with this version of MATLAB.

`[str,maxsize,endian] = computer` returns either `'L'` for little-endian byte ordering or `'B'` for big-endian byte ordering.

Platform	Word Size	str	archstr	maxsize	endian	ispc	isunix	ismac
Microsoft Windows	32-bit	PCWIN	win32	2^31 - 1	L	1	0	0
	64-bit	PCWIN64	win64	2^48 - 1	L	1	0	0
Linux	64-bit	GLNXA64	glnxa64	2^48 - 1	L	0	1	0
Apple Mac	64-bit	MACI64	maci64	2^48 - 1	L	0	1	1

More About

Tips

In some cases, both 32-bit and 64-bit versions of MATLAB can run on the same platform. In this case, the value returned by `computer` reflects which of these are running. For example, if you run a 32-bit version of MATLAB on a Windows x64 platform, `computer` returns `PCWIN`, indicating that the 32-bit version is running. You can get this information and the value of `archstr` from the **Help** menu, as described in "Information About Your Installation".

See Also

`getenv` | `setenv` | `ispc` | `isunix` | `ismac`

Introduced before R2006a

cond

Condition number with respect to inversion

Syntax

```
c = cond(X)
c = cond(X,p)
```

Description

The *condition number* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and the linear equation solution. Values of `cond(X)` and `cond(X,p)` near 1 indicate a well-conditioned matrix.

`c = cond(X)` returns the 2-norm condition number, the ratio of the largest singular value of `X` to the smallest.

`c = cond(X,p)` returns the matrix condition number in `p`-norm:

```
norm(X,p) * norm(inv(X),p)
```

If p is...	Then cond(X,p) returns the...
1	1-norm condition number
2	2-norm condition number
'fro'	Frobenius norm condition number
inf	Infinity norm condition number

More About

Algorithms

The algorithm for `cond` (when `p = 2`) uses the singular value decomposition, `svd`. When the input matrix is sparse, `cond` ignores any specified `p` value and calls `condest`.

See Also

`condeig` | `norm` | `condest` | `normest` | `rank` | `rcond` | `svd`

Introduced before R2006a

condeig

Condition number with respect to eigenvalues

Syntax

```
c = condeig(A)
[V,D,s] = condeig(A)
```

Description

`c = condeig(A)` returns a vector of condition numbers for the eigenvalues of `A`. These condition numbers are the reciprocals of the cosines of the angles between the left and right eigenvectors.

`[V,D,s] = condeig(A)` is equivalent to

```
[V,D] = eig(A);
s = condeig(A);
```

Large condition numbers imply that `A` is near a matrix with multiple eigenvalues.

See Also

`balance` | `cond` | `eig`

Introduced before R2006a

condest

1-norm condition number estimate

Syntax

```
c = condest(A)
c = condest(A,t)
[c,v] = condest(A)
```

Description

`c = condest(A)` computes a lower bound `c` for the 1-norm condition number of a square matrix `A`.

`c = condest(A,t)` changes `t`, a positive integer parameter equal to the number of columns in an underlying iteration matrix. Increasing the number of columns usually gives a better condition estimate but increases the cost. The default is `t = 2`, which almost always gives an estimate correct to within a factor 2.

`[c,v] = condest(A)` also computes a vector `v` which is an approximate null vector if `c` is large. `v` satisfies `norm(A*v,1) = norm(A,1)*norm(v,1)/c`.

Note: `condest` invokes `rand`. If repeatable results are required then use `rng` to set the random number generator to its startup settings before using `condest`.

```
rng('default')
```

More About

Tips

This function is particularly useful for sparse matrices.

Algorithms

`condest` is based on the 1-norm condition estimator of Hager [1] and a block-oriented generalization of Hager's estimator given by Higham and Tisseur [2]. The heart of the

algorithm involves an iterative search to estimate $\left\|A^{-1}\right\|_1$ without computing A^{-1}. This is posed as the convex but nondifferentiable optimization problem $\max\left\|A^{-1}\mathbf{x}\right\|_1$ subject to $\left\|\mathbf{x}\right\|_1 = 1$

References

[1] William W. Hager, "Condition Estimates," *SIAM J. Sci. Stat. Comput. 5*, 1984, 311-316, 1984.

[2] Nicholas J. Higham and Françoise Tisseur, "A Block Algorithm for Matrix 1-Norm Estimation with an Application to 1-Norm Pseudospectra, "*SIAM J. Matrix Anal. Appl.*, Vol. 21, 1185-1201, 2000.

See Also

`cond` | `norm` | `normest`

Introduced before R2006a

coneplot

Plot velocity vectors as cones in 3-D vector field

Syntax

```
coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)
coneplot(U,V,W,Cx,Cy,Cz)
coneplot(...,s)
coneplot(...,color)
coneplot(...,'quiver')
coneplot(...,'method')
coneplot(X,Y,Z,U,V,W,'nointerp')
coneplot(axes_handle,...)
h = coneplot(...)
```

Description

`coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)` plots velocity vectors as cones pointing in the direction of the velocity vector and having a length proportional to the magnitude of the velocity vector. `X`, `Y`, `Z` define the coordinates for the vector field. `U`, `V`, `W` define the vector field. These arrays must be the same size, monotonic, and represent a Cartesian, axis-aligned grid (such as the data produced by `meshgrid`). `Cx`, `Cy`, `Cz` define the location of the cones in the vector field. The section "Specifying Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.

`coneplot(U,V,W,Cx,Cy,Cz)` (omitting the `X`, `Y`, and `Z` arguments) assumes `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p]= size(U)`.

`coneplot(...,s)` automatically scales the cones to fit the graph and then stretches them by the scale factor `s`. If you do not specify a value for `s`, `coneplot` uses a value of 1. Use `s = 0` to plot the cones without automatic scaling.

`coneplot(...,color)` interpolates the array `color` onto the vector field and then colors the cones according to the interpolated values. The size of the `color` array must be the same size as the `U`, `V`, `W` arrays. This option works only with cones (that is, not with the `quiver` option).

`coneplot(...,'quiver')` draws arrows instead of cones (see `quiver3` for an illustration of a quiver plot).

`coneplot(...,'method')` specifies the interpolation method to use. *`method`* can be `linear`, `cubic`, or `nearest`. `linear` is the default. (See `interp3` for a discussion of these interpolation methods.)

`coneplot(X,Y,Z,U,V,W,'nointerp')` does not interpolate the positions of the cones into the volume. The cones are drawn at positions defined by `X`, `Y`, `Z` and are oriented according to `U`, `V`, `W`. Arrays `X`, `Y`, `Z`, `U`, `V`, `W` must all be the same size.

`coneplot(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = coneplot(...)` returns the handle to the `patch` object used to draw the cones.

`coneplot` automatically scales the cones to fit the graph, while keeping them in proportion to the respective velocity vectors.

Examples

3-D Cone Plot

Plot velocity vector cones for vector volume data representing motion of air through a rectangular region of space.

Load the data. The `wind` data set contains the arrays `u`, `v`, and `w` that specify the vector components and the arrays `x`, `y`, and `z` that specify the coordinates.

```
load wind
```

Establish the range of the data to place the slice planes and to specify where you want the cone plots.

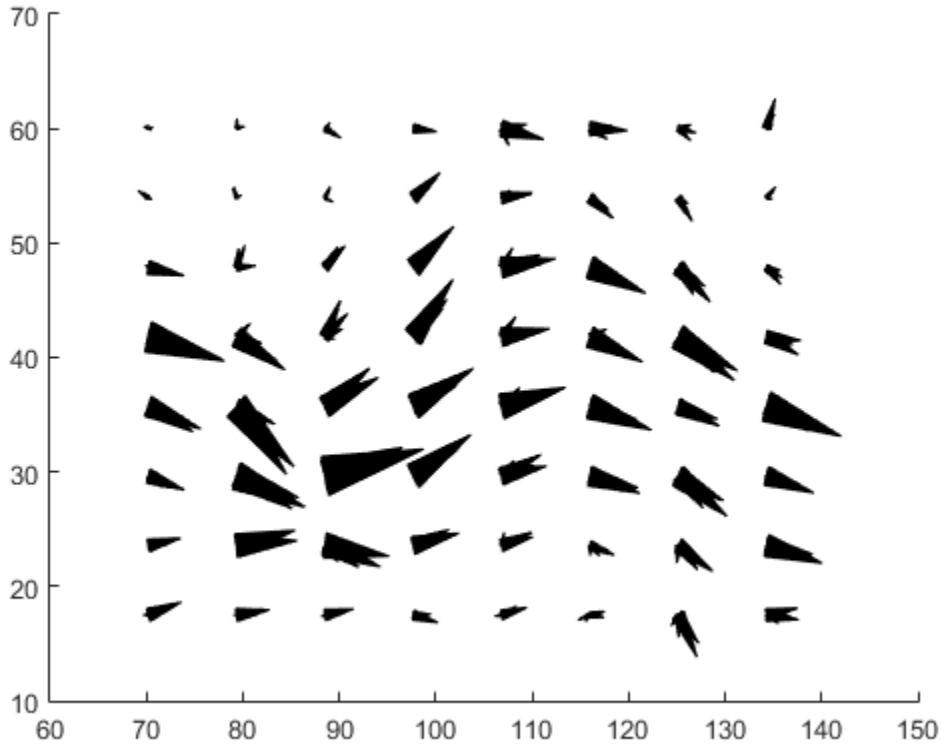
```
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
ymax = max(y(:));
zmin = min(z(:));
```

Define where to plot the cones. Select the full range in `x` and `y` and select the range 3 to 15 in `z`.

```
xrange = linspace(xmin,xmax,8);
yrange = linspace(ymin,ymax,8);
zrange = 3:4:15;
[cx,cy,cz] = meshgrid(xrange,yrange,zrange);
```

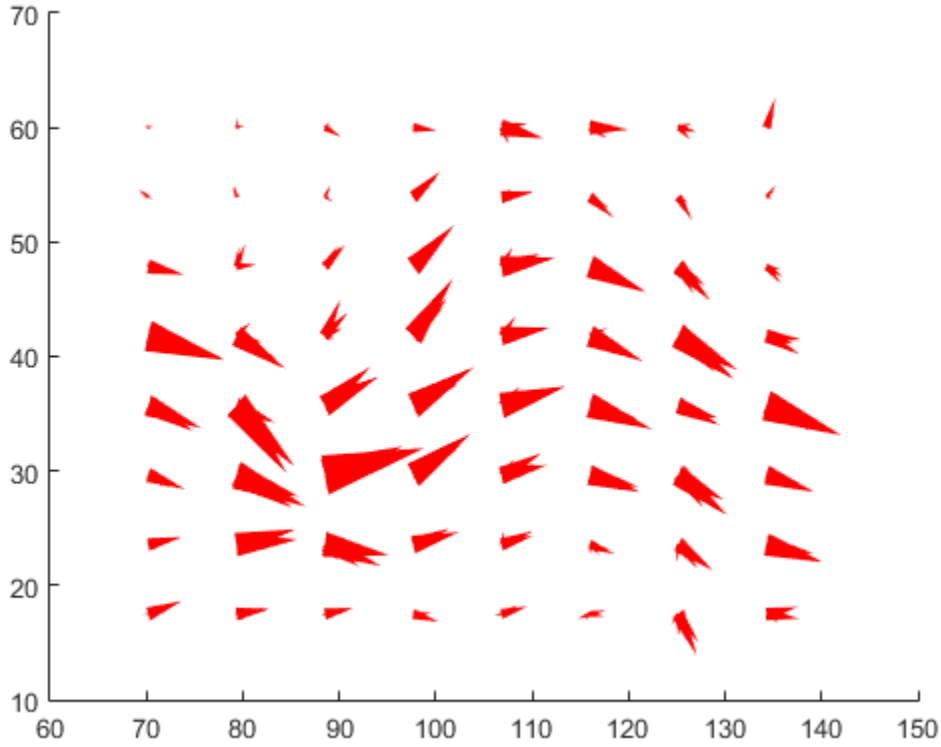
Plot the cones and set the scale factor to 5 to make the cones larger than the default size.

```
figure
hcone = coneplot(x,y,z,u,v,w,cx,cy,cz,5);
```



Set the cone colors.

```
hcone.FaceColor = 'red';
hcone.EdgeColor = 'none';
```

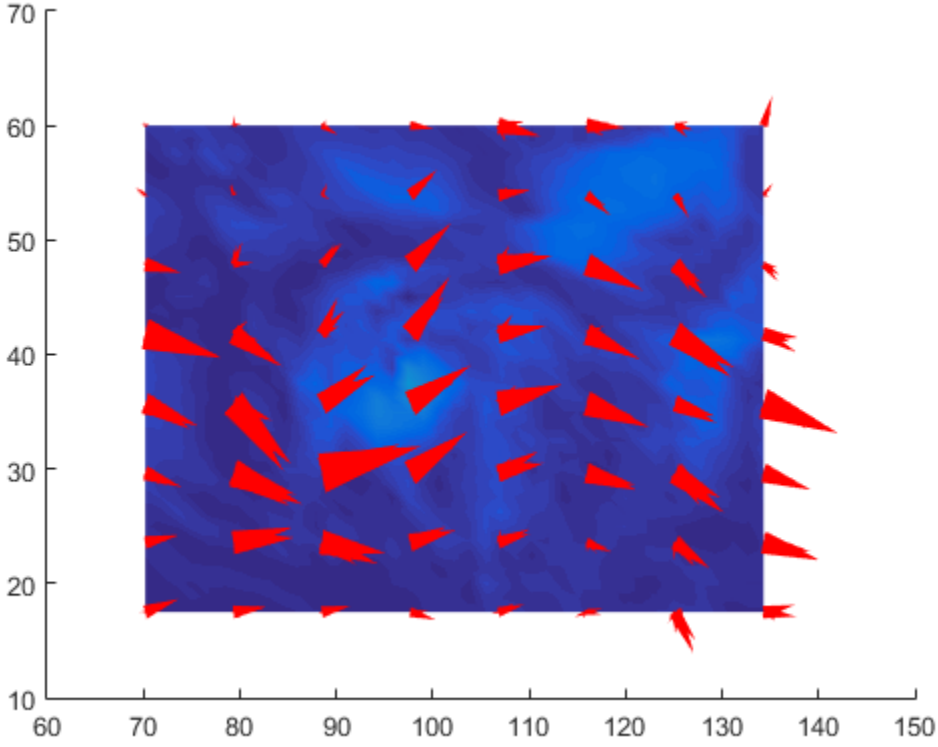


Calculate the magnitude of the vector field (which represents wind speed) to generate scalar data for the `slice` command.

```
hold on
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

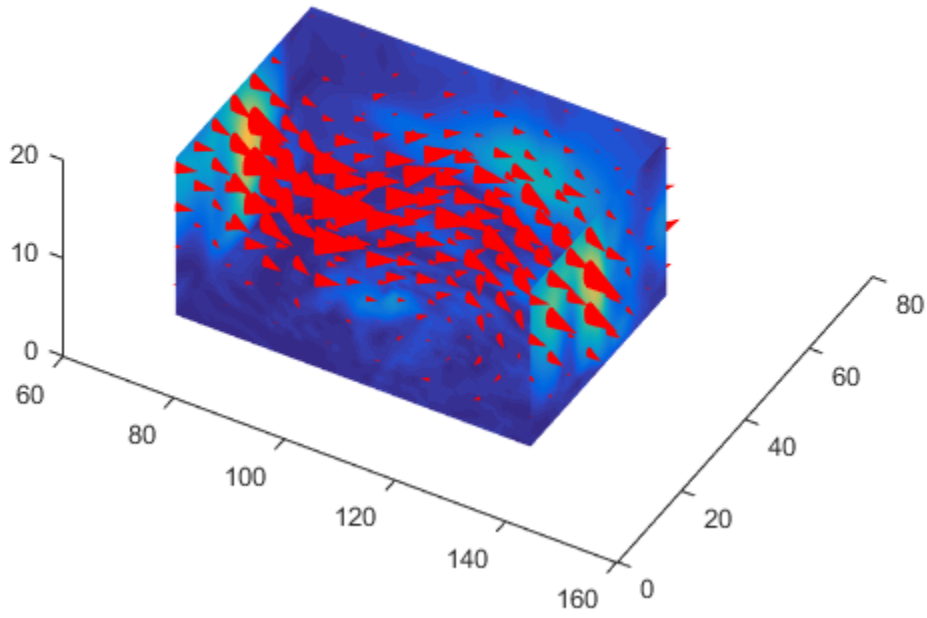
Create slice planes along the *x*-axis at `xmin` and `xmax`, along the *y*-axis at `ymax`, and along the *z*-axis at `zmin`. Specify interpolated face color so the slice coloring indicates wind speed, and do not draw edges.

```
hsurfaces = slice(x,y,z,wind_speed,[xmin,xmax],ymax,zmin);
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
hold off
```



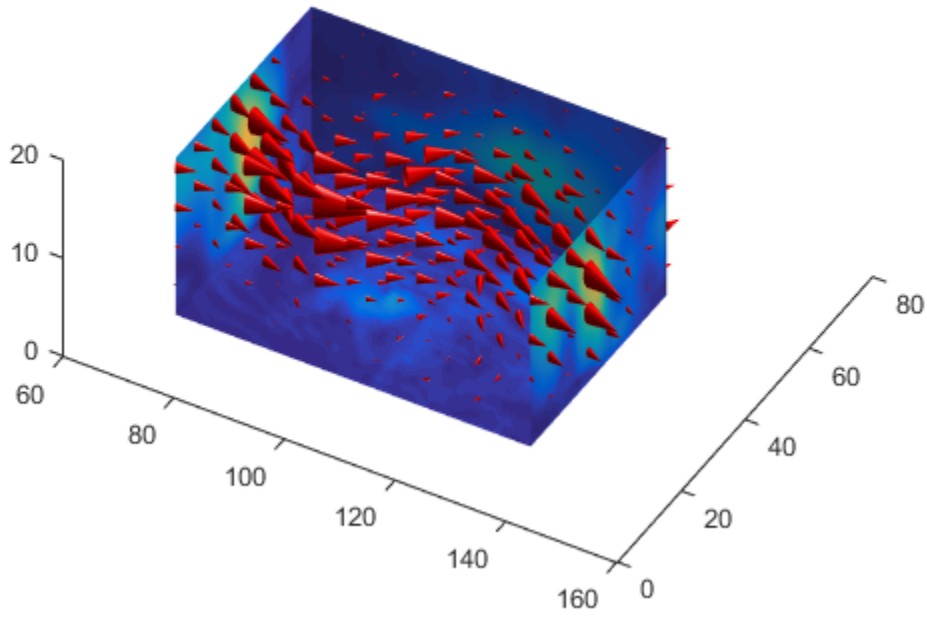
Change the axes view and set the data aspect ratio.

```
view(30,40)
daspect([2,2,1])
```



Add a light source to the right of the camera and use Gouraud lighting to give the cones and slice planes a smooth, three-dimensional appearance.

```
camlight right
lighting gouraud
set(hsurfaces,'AmbientStrength',0.6)
hcone.DiffuseStrength = 0.8;
```



- “Overview of Volume Visualization”

See Also

`isosurface` | `patch` | `reducevolume` | `smooth3` | `streamline` | `stream2` | `stream3` | `subvolume`

Introduced before R2006a

conj

Complex conjugate

Syntax

```
ZC = conj(Z)
```

Description

`ZC = conj(Z)` returns the complex conjugate of the elements of `Z`.

More About

Algorithms

If `Z` is a complex array:

```
conj(Z) = real(Z) - i*imag(Z)
```

See Also

`i` | `j` | `imag` | `real`

Introduced before R2006a

continue

Pass control to next iteration of `for` or `while` loop

Syntax

```
continue
```

Description

`continue` passes control to the next iteration of a `for` or `while` loop. It skips any remaining statements in the body of the loop for the current iteration. The program continues execution from the next iteration.

`continue` applies only to the body of the loop where it is called. In nested loops, `continue` skips remaining statements only in the body of the loop in which it occurs.

Examples

Selectively Display Values in Loop

Display the multiples of 7 from 1 through 50. If a number is not divisible by 7, use `continue` to skip the `disp` statement and pass control to the next iteration of the `for` loop.

```
for n = 1:50
    if mod(n,7)
        continue
    end
    disp(['Divisible by 7: ' num2str(n)])
end
```

```
Divisible by 7: 7
Divisible by 7: 14
Divisible by 7: 21
Divisible by 7: 28
Divisible by 7: 35
Divisible by 7: 42
```

```
Divisible by 7: 49
```

Skip to Next Loop Iteration

Count the number of lines of code in the file `magic.m`. Skip blank lines and comments using a `continue` statement. `continue` skips the remaining instructions in the `while` loop and begins the next iteration.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
count
fclose(fid);
```

```
count =

    31
```

More About

Tips

- The `continue` statement skips the rest of the instructions in a `for` or `while` loop and begins the next iteration. To exit the loop completely, use a `break` statement.
- `continue` is not defined outside a `for` or `while` loop. To exit a function, use `return`.

See Also

`break` | `for` | `while`

Introduced before R2006a

contour

Contour plot of matrix

Syntax

```
contour(Z)
contour(Z,n)
contour(Z,v)
contour(X,Y,Z)
contour(X,Y,Z,n)
contour(X,Y,Z,v)
contour(...,LineSpec)
contour(...,Name,Value)
contour(ax,...)
[C,h] = contour(...)
```

Description

A contour plot displays isolines of matrix `Z`. Label the contour lines using `clabel`.

`contour(Z)` draws a contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The *x* values correspond to the column indices of `Z` and the *y* values correspond to the row indices of `Z`. The contour levels are chosen automatically.

`contour(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels where `n` is a scalar. The contour levels are chosen automatically.

`contour(Z,v)` draws a contour plot of matrix `Z` with contour lines at the data values specified in the monotonically increasing vector `v`. To display a single contour line at a particular value, define `v` as a two-element vector with both elements equal to the desired contour level. For example, to draw contour lines at level `k`, use `contour(Z,[k k])`. Specifying the vector `v` sets the `LevelListMode` property to manual.

`contour(X,Y,Z)`, `contour(X,Y,Z,n)`, and `contour(X,Y,Z,v)` draw contour plots of `Z` using `X` and `Y` to determine the *x* and *y* values.

- If `X` and `Y` are vectors, then `length(X)` must equal `size(Z,2)` and `length(Y)` must equal `size(Z,1)`. The vectors must be strictly increasing or strictly decreasing and cannot contain any repeated values.
- If `X` and `Y` are matrices, then their sizes must equal the size of `Z`. Typically, you should set `X` and `Y` so that the columns are strictly increasing or strictly decreasing and the rows are uniform (or the rows are strictly increasing or strictly decreasing and the columns are uniform).

If `X` or `Y` is irregularly spaced, then `contour` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

`contour(...,LineSpec)` draws the contours using the line type and color specified by `LineSpec`. `contour` ignores marker symbols.

`contour(...,Name,Value)` specifies contour properties using one or more property name, property value pairs. `Name` is the property name and must appear inside single quotes (`''`). `Value` is the corresponding value. For example, `'LineWidth',2` sets the contour line width to 2. For a list of contour property names and values, see Contour Properties.

`contour(ax,...)` plots into the axes specified by `ax` instead of the current axes (`gca`).

`[C,h] = contour(...)` returns the contour matrix `C` containing the data that defines the contour lines, and the contour object `h`. The `ContourMatrix` property for the contour object also contains the contour matrix. The `clabel` function uses the contour matrix to label the contour lines.

Use contour object properties to control the contour plot appearance. For a list, see Contour Properties.

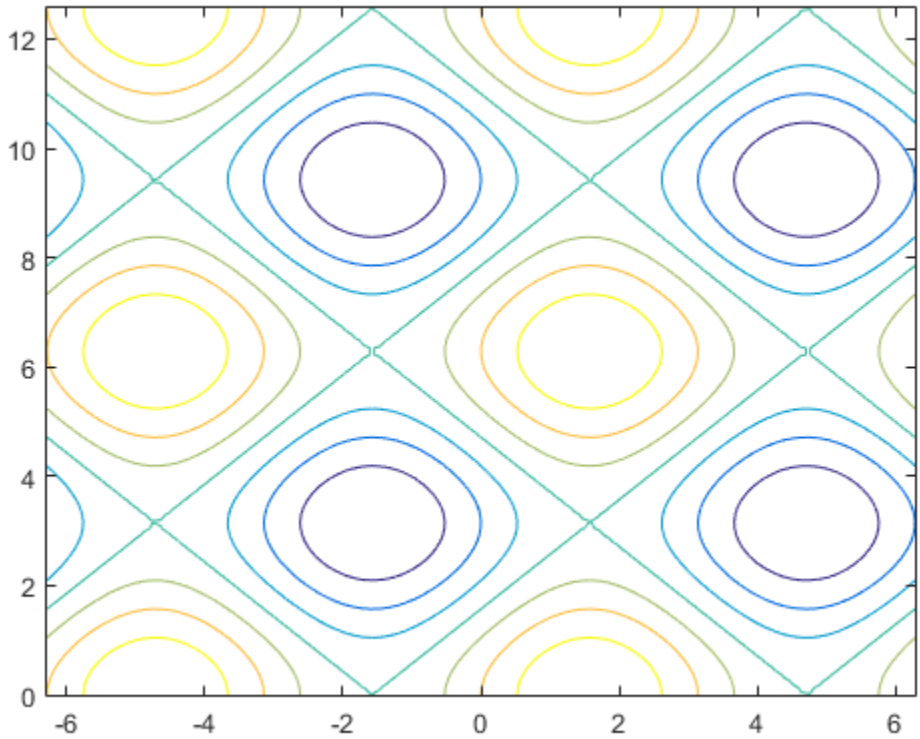
Examples

Create Contour Plot

Use the `meshgrid` function to generate matrices `X` and `Y`. Create a third matrix, `Z`, and plot its contours.

```
x = linspace(-2*pi,2*pi);
y = linspace(0,4*pi);
```

```
[X,Y] = meshgrid(x,y);
Z = sin(X)+cos(Y);

figure
contour(X,Y,Z)
```

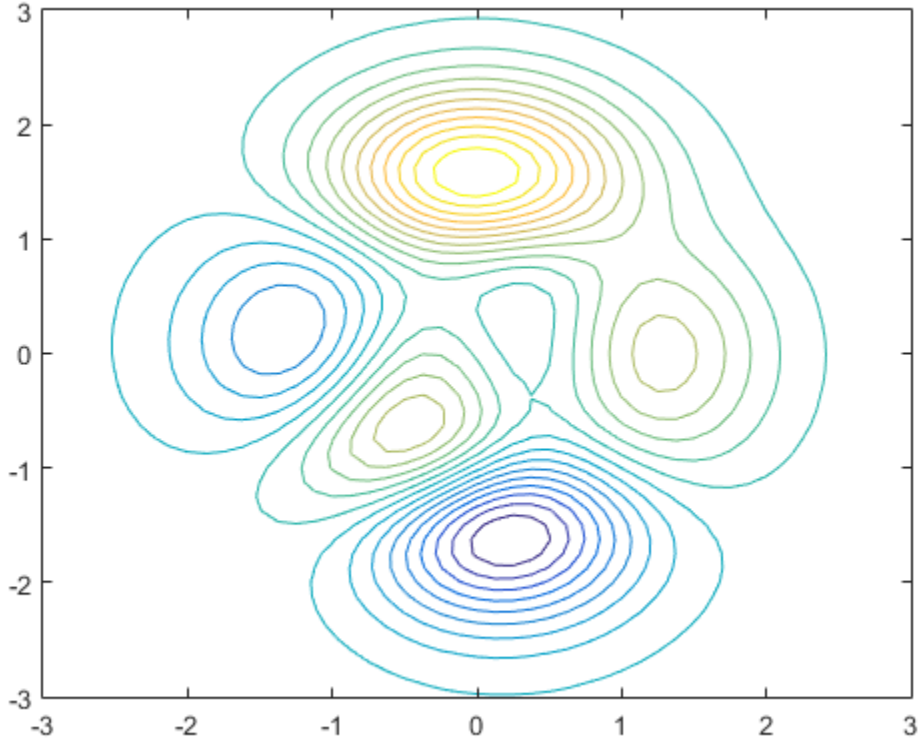


Specify Number of Contour Lines

Store the data from the `peaks` function in matrices `X`, `Y`, and `Z`. Plot 20 contours of the data in `Z`.

```
[X,Y,Z] = peaks;
figure
```

```
contour(X,Y,Z,20)
```

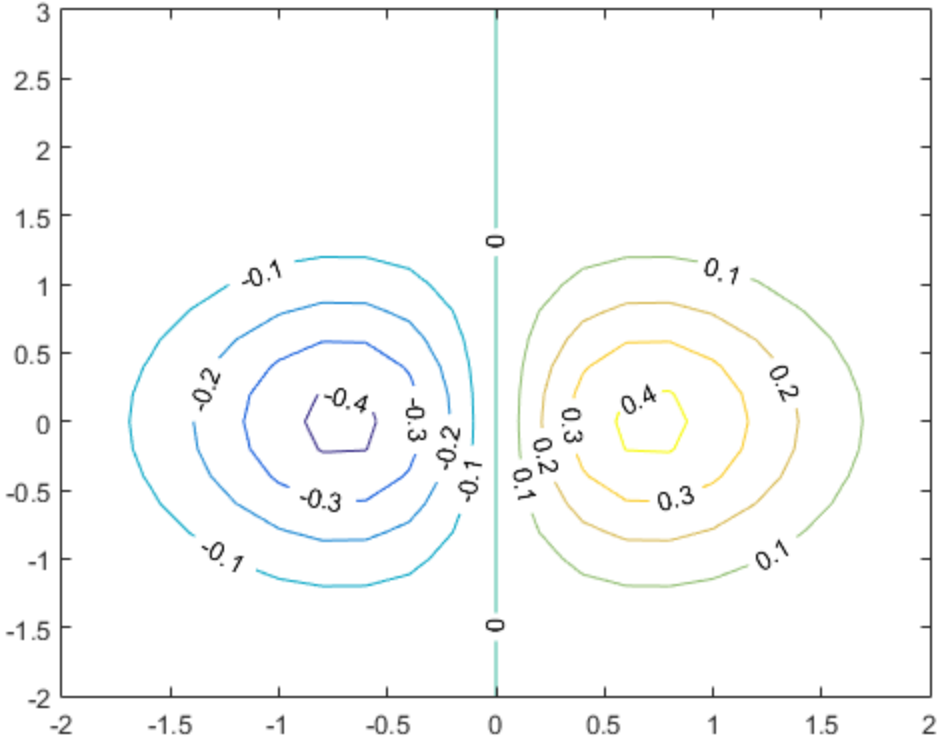


Display Contour Labels

Set up matrices `X`, `Y`, and `Z`. Create a contour plot and display the contour labels by setting the `ShowText` property to `on`.

```
x = -2:0.2:2;
y = -2:0.2:3;
[X,Y] = meshgrid(x,y);
Z = X.*exp(-X.^2-Y.^2);

figure
```

```
contour(X,Y,Z,'ShowText','on')
```

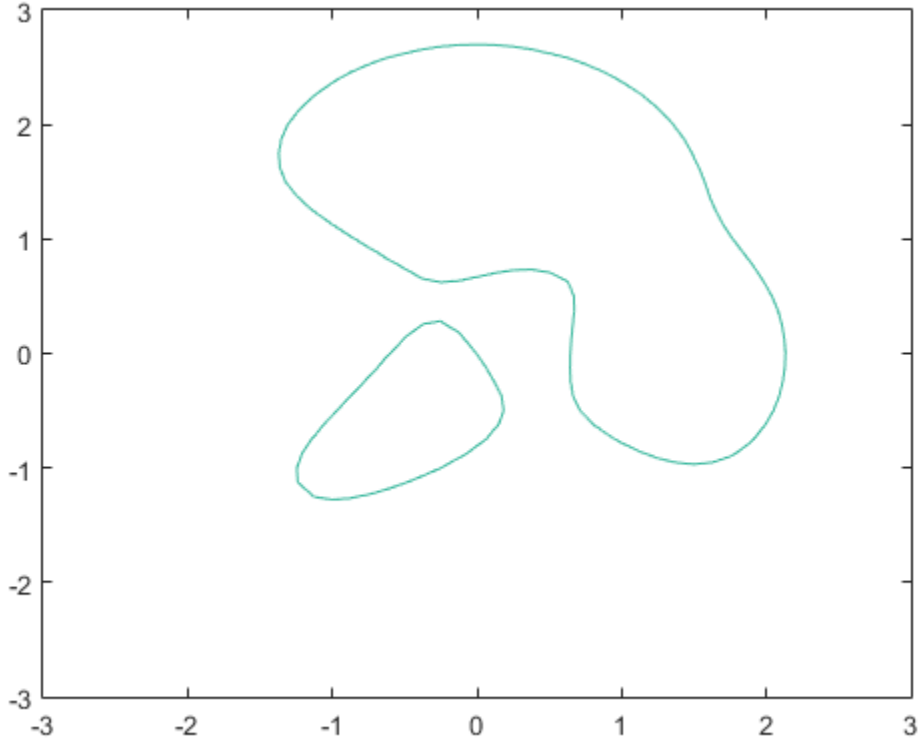


Display Single Contour Line

Create a contour plot of the `peaks` function and display only one contour level at `Z = 1`.

```
x = -3:0.125:3;
y = -3:0.125:3;
[X,Y] = meshgrid(x,y);
Z = peaks(X,Y);
v = [1,1];

figure
```

```
contour(X,Y,Z,v)
```



More About

Tips

- The `contour` function cannot determine if there are discontinuities in the input data. Add `NaN` values to the data to prevent drawing the contour lines in those regions.

- "Label Contour Plot Levels"
- "Highlight Specific Contour Levels"

See Also

Functions
`clabel` | `contour3` | `contourc` | `contourf` | `quiver`

Properties
Contour Properties | Text Properties

Introduced before R2006a

contour3

3-D contour plot

Syntax

```
contour3(Z)
contour3(Z,n)
contour3(Z,v)
contour3(X,Y,Z)
contour3(X,Y,Z,n)
contour3(X,Y,Z,v)
contour3(...,LineSpec)
contour3(...,Name,Value)
contour3(ax,...)
[C,h] = contour3(...)
```

Description

`contour3` creates a 3-D contour plot of a surface defined on a rectangular grid.

`contour3(Z)` draws a contour plot of matrix `Z` in a 3-D view. `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The *x* values correspond to the column indices of `Z` and the *y* values correspond to the row indices of `Z`. The contour levels are chosen automatically.

`contour3(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels in a 3-D view.

`contour3(Z,v)` draws a contour plot of matrix `Z` with contour lines at the values specified in vector `v`. The number of contour levels is equal to `length(v)`. Specifying the vector `v` sets the `LevelListMode` property to manual. To display a single contour line at a particular value, define `v` as a two-element vector with both elements equal to the desired contour level. For example, to draw a single contour of level `k`, use `contour3(Z,[k k])`.

`contour3(X,Y,Z)`, `contour3(X,Y,Z,n)`, and `contour3(X,Y,Z,v)` draw contour plots of `Z` using `X` and `Y` to determine the *x* and *y* values.

- If `X` and `Y` are vectors, then `length(X)` must equal `size(Z,2)` and `length(Y)` must equal `size(Z,1)`. The vectors must be strictly increasing or strictly decreasing and cannot contain any repeated values.
- If `X` and `Y` are matrices, then their sizes must equal the size of `Z`. Typically, you should set `X` and `Y` so that the columns are strictly increasing or strictly decreasing and the rows are uniform (or the rows are strictly increasing or strictly decreasing and the columns are uniform).

If `X` or `Y` is irregularly spaced, then `contour3` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

`contour3(...,LineSpec)` draws the contour lines using the line type and color specified by `LineSpec`. `contour3` ignores marker symbols.

`contour3(...,Name,Value)` specifies contour properties using one or more property name, property value pairs. `Name` is the property name and must appear inside single quotes (`''`). `Value` is the corresponding value. For example, `'LineWidth',2` sets the contour line width to 2. For a list of contour property names and values, see Contour Properties.

`contour3(ax,...)` plots into the axes specified by `ax` instead of into the current axes (`gca`).

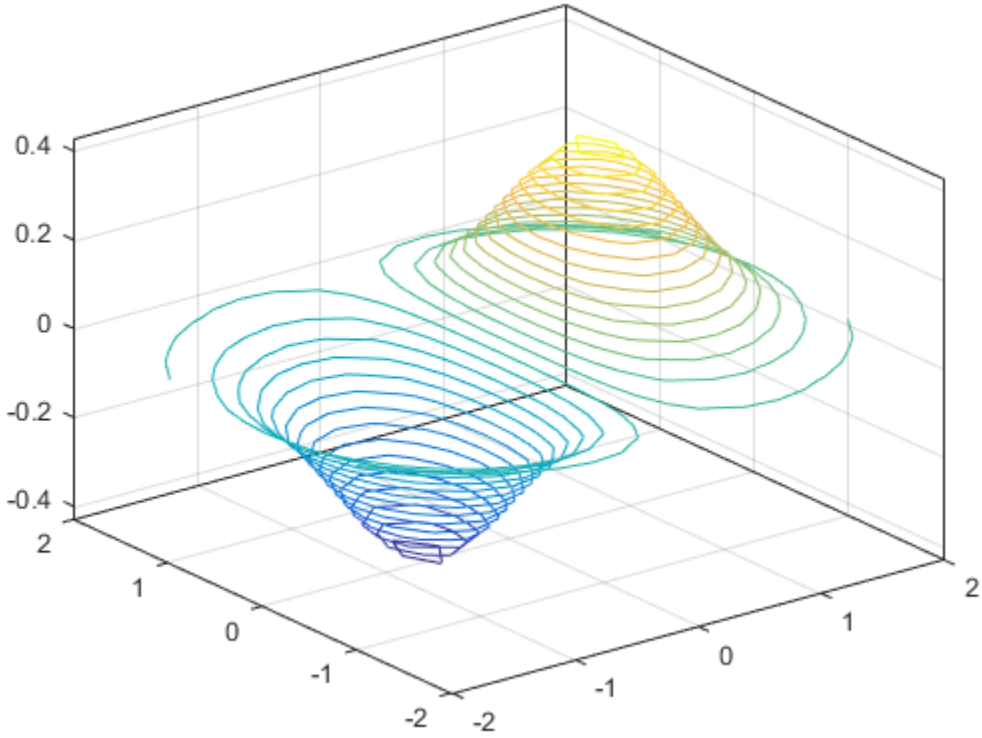
`[C,h] = contour3(...)` returns the contour matrix `C` containing the data that defines the contour lines and the contour object `h`. The `clabel` function uses the contour matrix to label the contour lines.

Examples

Create 3-D Contour Plot

Set up matrices `X` and `Y` using the `meshgrid` function. Plot 30 contours of matrix `Z`.

```
x = -2:0.25:2;
[X,Y] = meshgrid(x);
Z = X.*exp(-X.^2-Y.^2);
contour3(X,Y,Z,30)
```



More About

Tips

If you do not specify `LineSpec`, the functions `colormap` and `caxis` control the color.

Label the contour lines using `clabel`.

`contour3(...)` works the same as `contour(...)` except that the contours are drawn at their corresponding *z* values.

See Also

Functions
`contour` | `contourc` | `contourf` | `meshc` | `meshgrid` | `surfc`

Properties
Contour Properties

Introduced before R2006a

contourc

Low-level contour plot computation

Syntax

```
C = contourc(Z)
C = contourc(Z,n)
C = contourc(Z,v)
C = contourc(x,y,Z)
C = contourc(x,y,Z,n)
C = contourc(x,y,Z,v)
```

Description

`contourc` calculates the contour matrix `C` used by `contour`, `contour3`, and `contourf`. The values in `Z` determine the heights of the contour lines with respect to a plane. The contour calculations use a regularly spaced grid determined by the dimensions of `Z`.

`C = contourc(Z)` computes the contour matrix from data in matrix `Z`, where `Z` must be at least a 2-by-2 matrix. The contours are isolines in the units of `Z`. The number of contour lines and the corresponding values of the contour lines are chosen automatically.

`C = contourc(Z,n)` computes contours of matrix `Z` with `n` contour levels.

`C = contourc(Z,v)` computes contours of matrix `Z` with contour lines at the values specified in vector `v`. The length of `v` determines the number of contour levels. To compute a single contour of level `k`, use `contourc(Z,[k k])`.

`C = contourc(x,y,Z)`, `C = contourc(x,y,Z,n)`, and `C = contourc(x,y,Z,v)` compute contours of `Z` using vectors `x` and `y` to determine the *x* and *y* values. `x` and `y` must be monotonically increasing.

More About

Tips

For more information on the contour matrix, see the `ContourMatrix` property for contour objects.

Specifying irregularly spaced `x` and `y` vectors is not the same as contouring irregularly spaced data. If `x` or `y` is irregularly spaced, `contourc` calculates contours using a regularly spaced contour grid, then transforms the data to `x` or `y`.

See Also

`clabel` | `contour` | `contour3` | `contourf`

Introduced before R2006a

contourf

Filled 2-D contour plot

Syntax

```
contourf(Z)
contourf(Z,n)
contourf(Z,v)
contourf(X,Y,Z)
contourf(X,Y,Z,n)
contourf(X,Y,Z,v)
contourf(...,LineSpec)
contourf(...,Name,Value)
contourf(ax,...)
[C,h] = contourf(...)
```

Description

A filled contour plot displays isolines calculated from matrix `Z` and fills the areas between the isolines using constant colors corresponding to the current figure's colormap.

`contourf(Z)` draws a filled contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The *x* values correspond to the column indices of `Z` and the *y* values correspond to the row indices of `Z`. The contour levels are chosen automatically.

`contourf(Z,n)` draws a filled contour plot of matrix `Z` with `n` contour levels.

`contourf(Z,v)` draws a filled contour plot of matrix `Z` with contour lines at the data values specified in the monotonically increasing vector `v`. To display a single contour line at a particular value, define `v` as a two-element vector with both elements equal to the desired contour level. For example, to draw a single contour of level `k`, use `contourf(Z, [k k])`. Specifying the vector `v` sets the `LevelListMode` property to manual.

`contourf(X,Y,Z)`, `contourf(X,Y,Z,n)`, and `contourf(X,Y,Z,v)` draw filled contour plots of `Z` using `X` and `Y` to determine the *x* and *y* values.

- If `X` and `Y` are vectors, then `length(X)` must equal `size(Z,2)` and `length(Y)` must equal the number of rows in `size(Z,1)`. The vectors must be strictly increasing or strictly decreasing and cannot contain any repeated values.
- If `X` and `Y` are matrices, then their sizes must equal the size of `Z`. Typically, you should set `X` and `Y` so that the columns are strictly increasing or strictly decreasing and the rows are uniform (or the rows are strictly increasing or strictly decreasing and the columns are uniform).

If `X` or `Y` is irregularly spaced, then `contourf` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

`contourf(...,LineSpec)` draws the contour lines using the line type and color specified by `LineSpec`. `contourf` ignores marker symbols.

`contourf(...,Name,Value)` specifies contour properties using one or more property name, property value pairs. `Name` is the property name and must appear inside single quotes (`''`). `Value` is the corresponding value. For example, `'LineWidth',2` sets the contour line width to 2. For a list of contour property names and values, see Contour Properties.

`contourf(ax,...)` plots into the axes specified by `ax` instead of into the current axes (`gca`).

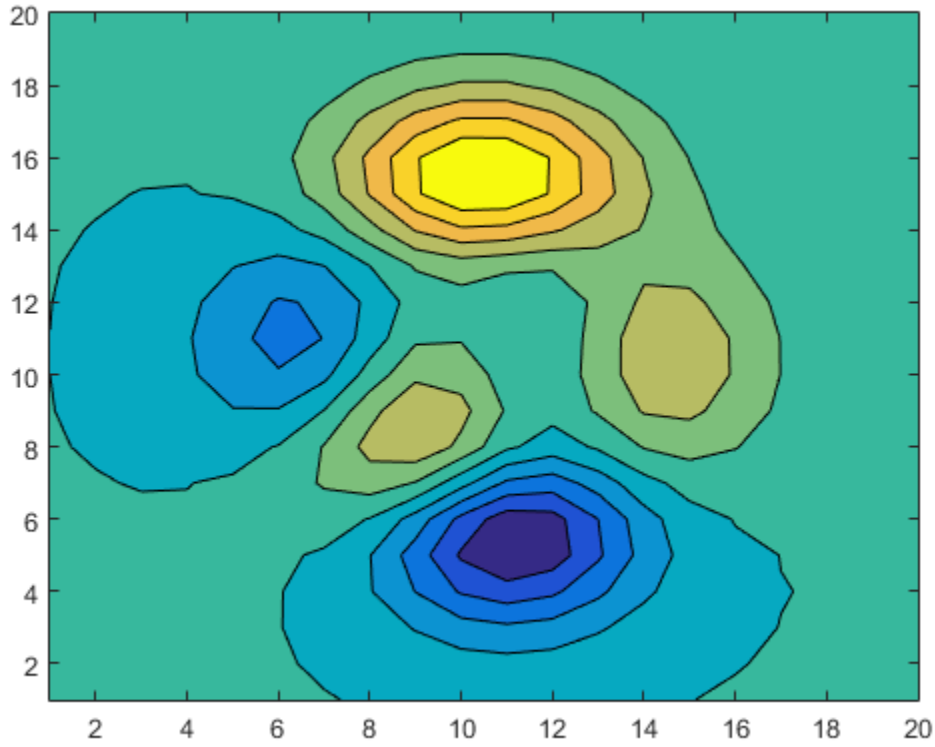
`[C,h] = contourf(...)` returns the contour matrix `C` containing the data that defines the contour lines and a contour object `h`. The `ContourMatrix` property for the contour object also contains the contour matrix. The `clabel` function uses the contour matrix to label the contour lines.

Examples

Create Filled Contour Plot

Use the `peaks` function to define `z` as a 20-by-20 matrix. Create a filled contour plot of `z` with 10 contour lines.

```
Z = peaks(20);
contourf(Z,10)
```



More About

Tips

`NaN` values in `Z` leave white holes with black borders in the contour plot.

- "Change Fill Colors for Contour Plot"

See Also

Functions

`clabel` | `contour` | `contour3` | `contourc` | `quiver`

Properties
Contour Properties

Introduced before R2006a

Contour Properties

Control contour appearance and behavior

Contour properties control the appearance and behavior of contour objects. By changing property values, you can modify certain aspects of the contour.

Starting in R2014b, you can use dot notation to query and set properties.

```
[C,h] = contour(...);
w = h.LineWidth;
h.LineWidth = 2;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Lines

`LineColor` — Color of contour lines

`'flat'` (default) | `'none'` | RGB triplet | color string

Color of contour lines, specified as one of these values:

- `'flat'` — Use a different color for each contour line, determined by its contour value, the colormap, and the scaling of data values into the colormap. For more information on color scaling, see `caxis`.
- `'none'` — Do not draw the contour lines.
- RGB triplet or color string — Use the same color for all contour lines. Specify the color using an RGB triplet or a color string.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

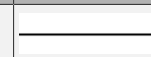
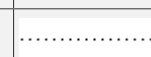
Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

`LineStyle` — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

`LineWidth` — Width of contour lines

0.5 (default) | positive value

Width of contour lines, specified as a positive value in points. One point equals 1/72 inch.

Example: 2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Contour Levels

`LevelList` — Contour levels

empty matrix (default) | vector of z values

Contour levels, specified as a vector of z values. By default, the `contour` function chooses values that span the range of values in the `ZData` property.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelListMode` — Selection mode for `LevelList`

`'auto'` (default) | `'manual'`

Selection mode for the LevelList, specified as one of these values:

- `'auto'` — Determine the values based on the `ZData` values.
- `'manual'` — Use manually specified values. To specify the values, set the LevelList property. When the mode is manual, the `contour` function does not change the values as you change ZData.

`LevelStep` — Spacing between contour lines

`0` (default) | scalar numeric value

Spacing between contour lines, specified as a scalar numeric value. For example, specify a value of 2 to draw contour lines at increments of 2. The `contour` function determines the contour interval based on the `ZData` values.

Setting this property sets the associated mode property to manual.

Example: `3.4`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelStepMode` — Selection mode for `LevelStep`

`'auto'` (default) | `'manual'`

Selection mode for the LevelStep, specified as one of these values:

- `'auto'` — Determine the value based on the `ZData` values.
- `'manual'` — Use a manually specified value. To specify the value, set the LevelStep property. When the mode is manual, the `contour` function does not change the value as you change `ZData`.

Contour Labels

`ShowText` — Contour line labels

`'off'` (default) | `'on'`

Contour line labels, specified as one of these values:

- `'off'` — Do not label the contour lines.
- `'on'` — Display text labels on each contour line indicating the contour value.

`LabelSpacing` — Spacing between contour line labels

144 (default) | numeric scalar

Spacing between contour line labels, specified as a numeric scalar in point units. One point equals 1/72 inch. You must set the `ShowText` property to `'on'` for the `LabelSpacing` property to have an effect.

If you use the `clabel` function to display the labels, then the `LabelSpacing` property has no effect. The contour plot contains a single label per line instead.

Example: `36`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TextList` — Contour lines to label

empty matrix (default) | vector of real values

Contour lines to label, specified as a vector of real values.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TextListMode` — Selection mode for `TextList`

`'auto'` (default) | `'manual'`

Selection mode for the TextList, specified as one of these values:

- `'auto'` — Use values equal to the values of the LevelList property. The contour plot includes a text label for each line.
- `'manual'` — Use manually specified values. Specify the values by setting the TextList property.

`TextStep` — Interval between labeled contour lines

0 (default) | scalar numeric value

Interval between labeled contour lines, specified as a scalar numeric value. By default, the contour plot includes a label for every contour line when the `ShowText` property is set to `'on'`.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TextStepMode` — Selection mode for `TextStep`

`'auto'` (default) | `'manual'`

Selection mode for the TextStep, specified as one of these values:

- `'auto'` — Determine value based on the `ZData` values. If the `ShowText` property is set to `'on'`, then the `contour` function labels every contour line.
- `'manual'` — Use a manually specified value. To specify the value, set the TextStep property.

Filled Contours

`Fill` — Fill between contour lines

`'off'` (default) | `'on'`

Fill between contour lines, specified as one of these values:

- `'off'` — Do not fill the spaces between contour lines with a color. This is the default value when you create the contour chart using the `contour` or `contour3` functions.
- `'on'` — Fill the spaces between contour lines with color. This is the default value when you create the contour chart using the `contourf` function.

Contour Matrix

`ContourMatrix` — Contour line definitions

[ ] (default) | two-row matrix

Contour line definitions, returned as a two-row matrix. Each contour line in the plot has an associated definition. If there are a total of `N` contour lines in the plot, then the contour matrix consists of `N` definitions:

```
C = [C(1) C(2)...C(k)...C(N)]
```

Each contour line definition follows this pattern:

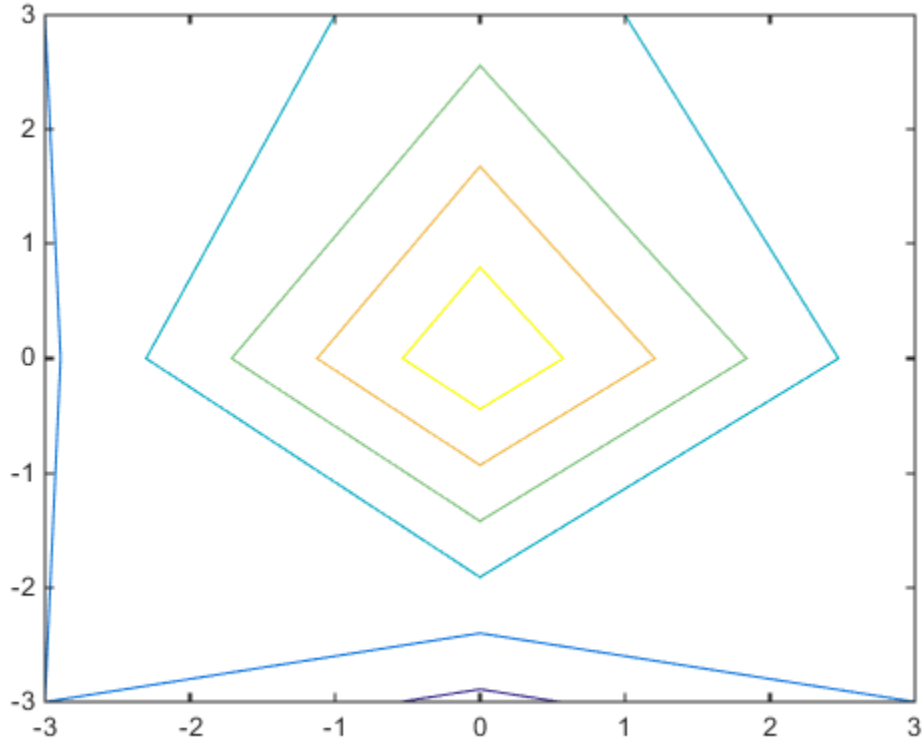
```
C(k) = [level   x(1) x(2)...
        numxy   y(1) y(2)... ]
```

The first entry, `level`, indicates the contour level where the contour line is drawn. Beneath the contour level is the number of (*x*,*y*) vertices that define the contour line. The remaining columns contain the data for each of the vertices. If the first and last vertices are the same, then the contour line is a closed loop. If a particular contour level has multiple contour lines in the graph, then the matrix contains a separate definition for each line.

Example

Create a contour plot of values from the `peaks` function.

```
[x,y,z] = peaks(3);
[C,h] = contour(x,y,z);
```



Access the contour matrix using either the output argument `C` or the `ContourMatrix` property of the contour object (`h.ContourMatrix`). The contour matrix contains definitions for each of the seven contour lines. The circles in this matrix show the beginnings of the contour line definitions.

```
C =

  Columns 1 through 8

   -0.2000    1.8165    2.0000    2.1835         0    1.0003    2.0000    3.0000
    3.0000    1.0000    1.0367    1.0000    3.0000    1.0000    1.1998    1.0002

  Columns 9 through 16

         0    1.0000    1.0359    1.0000    0.2000    1.6669    1.2324    2.0000
    3.0000    2.9991    2.0000    1.0018    5.0000    3.0000    2.0000    1.3629

  Columns 17 through 24

    2.8240    2.3331    0.4000    1.4290    2.0000    2.6130    2.0000    1.4290
    2.0000    3.0000    5.0000    2.0000    1.5261    2.0000    2.8530    2.0000

  Columns 25 through 32

    0.6000    1.6255    2.0000    2.4020    2.0000    1.6255    0.8000    1.8221
    5.0000    2.0000    1.6892    2.0000    2.5594    2.0000    5.0000    2.0000

  Columns 33 through 36

    2.0000    2.1910    2.0000    1.8221
    1.8524    2.0000    2.2657    2.0000
```

The first definition in the matrix indicates that there is a contour line drawn at the -0.2 level , consisting of the three vertices `(1.8165,1)`, `(2,1.0367)`, and `(2.1835,1)`. Since the first and last vertices are not the same, the contour line is not a closed loop. The second and third definitions indicate that there are two closed loops at the 0 level.

Data

`XData` — x values

[ ] (default) | vector or matrix

x values, specified as a vector or matrix.

- If `XData` is a vector, then `length(XData)` must equal `size(ZData,2)` and YData must also be a vector. The `XData` values must be strictly increasing or strictly decreasing and cannot contain any duplicates.
- If `XData` is a matrix, then `size(XData)` and `size(YData)` must equal `size(ZData)`. Typically, you should set the `XData` values so that the columns are strictly increasing or strictly decreasing and the rows are uniform (or the rows are strictly increasing or strictly decreasing and the columns are uniform).

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`YData` — y values

[ ] (default) | vector or matrix

y values, specified as a vector or matrix.

- If `YData` is a vector, then `length(YData)` must equal `size(ZData,1)` and XData must also be a vector. The `XData` values must be strictly increasing or strictly decreasing and cannot contain any duplicates.
- If `YData` is a matrix, then `size(XData)` and `size(YData)` must equal `size(ZData)`. Typically, you should set the `YData` values so that the columns are strictly increasing or strictly decreasing and the rows are uniform (or the rows are strictly increasing or strictly decreasing and the columns are uniform).

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`ZData` — Data that defines surface to contour

[ ] (default) | matrix

Data that defines the surface to contour, specified as a matrix. `ZData` must be at least a 2-by-2 matrix.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`XDataSource` — Variable linked to `XData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZDataSource` — Variable linked to `ZData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `ZData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

`XDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for the `XData`, specified as one of these values:

- `'auto'` — Set the `XData` using the column indices of `ZData`.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property directly, or specify the input argument `X` to the `contour`, `contourf`, or `contour3` function.

`YDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for the `YData`, specified as one of these values:

- `'auto'` — Set the `YData` using the row indices of `ZData`.
- `'manual'` — Use manually specified values. To specify the values, set the `YData` property directly, or specify the input argument `Y` to the `contour`, `contourf`, or `contour3` function.

Visibility

`Visible` — Visibility of contour

`'on'` (default) | `'off'`

Visibility of contour, specified as one of these values:

- `'on'` — Display the contour.

- `'off'` — Hide the contour without deleting it. You still can access the properties of an invisible contour object.

`Clipping` — Clipping of contour to axes limits

`'on'` (default) | `'off'`

Clipping of contour to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the contour that are outside the axes limits.
- `'off'` — Display the entire contour, even if parts of it appear outside the axes limits. Parts of the contour might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the contour that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'contour'`

Type of graphics object, returned as the string `'contour'`.

`Tag` — User-specified tag

`''` (default) | any string

Tag to associate with the contour, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with contour

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the contour object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the contour.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the contour object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the contour from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the contour object in the legend as one entry (default).
   - `'off'` — Do not include the contour object in the legend.
   - `'children'` — Include only children of the contour object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of contour

axes object | group object | transform object

Parent of contour, specified as an axes, group, or transform object.

`Children` — Children of contour

empty `GraphicsPlaceholder` array

The contour has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of contour object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The contour object handle is always visible.
- `'off'` — The contour object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The contour object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the contour at the command-line, but allows callback functions to access it.

If the contour object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the contour. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The contour object — You can access properties of the contour object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the contour. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the contour when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the contour.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The Visible property must be set to `'on'`. The HitTest property determines if the contour responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the contour passes the click to the object below it in the current view of the figure window. The HitTest property of the contour has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the contour. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the contour that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the contour object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for contour

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for contour, specified as one of these values:

- `'off'` — Click the contour plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the contour plot to select it, that is, anywhere within the rectangle that encloses the contour plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the contour is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the contour tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the contour. Setting the `CreateFcn` property on an existing contour has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during contour creation. MATLAB executes the callback after creating the contour and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The contour object — You can access properties of the contour object from within the callback function. You also can access the contour object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the contour. MATLAB executes the callback before destroying the contour so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The contour object — You can access properties of the contour object from within the callback function. You also can access the contour object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of contour

`'off'` (default) | `'on'`

Deletion status of contour, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the contour begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the contour no longer exists.

Check the value of the BeingDeleted property to verify that the contour is not about to be deleted before querying or modifying it.

See Also

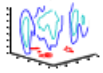
`contour` | `contour3` | `contourf` | `ezcontour` | `ezcontourf` | `ezmeshc` | `ezsurfc` | `meshc` | `surfc`

More About

- "Access Property Values"
- "Graphics Object Properties"

contourslice

Draw contours in volume slice planes



Syntax

```
contourslice(X,Y,Z,V,Sx,Sy,Sz)
contourslice(X,Y,Z,V,Xi,Yi,Zi)
contourslice(V,Sx,Sy,Sz)
contourslice(V,Xi,Yi,Zi)
contourslice(...,n)
contourslice(...,cvals)
contourslice(...,[cv cv])
contourslice(...,'method')
contourslice(axes_handle,...)
h = contourslice(...)
```

Description

`contourslice(X,Y,Z,V,Sx,Sy,Sz)` draws contours in the *x*-, *y*-, and *z*-axis aligned planes at the points in the vectors `Sx`, `Sy`, `Sz`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V` and must be monotonic and represent a Cartesian, axis-aligned grid (such as the data produced by `meshgrid`). The color at each contour is determined by the volume `V`, which must be an m-by-n-by-p volume array.

`contourslice(X,Y,Z,V,Xi,Yi,Zi)` draws contours through the volume `V` along the surface defined by the 2-D arrays `Xi,Yi,Zi`. The surface should lie within the bounds of the volume.

`contourslice(V,Sx,Sy,Sz)` and `contourslice(V,Xi,Yi,Zi)` (omitting the `X`, `Y`, and `Z` arguments) assume `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p]= size(v)`.

`contourslice(...,n)` draws `n` contour lines per plane, overriding the automatic value.

`contourslice(...,cvals)` draws `length(cval)` contour lines per plane at the values specified in vector `cvals`.

`contourslice(...,[cv cv])` computes a single contour per plane at the level `cv`.

`contourslice(...,'method')` specifies the interpolation method to use. *`method`* can be `linear`, `cubic`, or `nearest`. `nearest` is the default except when the contours are being drawn along the surface defined by `Xi`, `Yi`, `Zi`, in which case `linear` is the default. (See `interp3` for a discussion of these interpolation methods.)

`contourslice(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = contourslice(...)` returns a vector of handles to `patch` objects that are used to implement the contour lines.

Examples

Contour Slices of Fluid Flow

Store the matrices `X`, `Y`, `Z`, and `V` from the `flow` data set.

```
[X,Y,Z,V] = flow;
```

Create nine contour plots in the y-z plane, no plots in the x-z plane, and one plot in the x-y plane by specifying `Sx` as a vector of nine elements, `Sy` as an empty vector, and `Sz` as a scalar.

```
Sx = 1:9;
Sy = [];
Sz = 0;
```

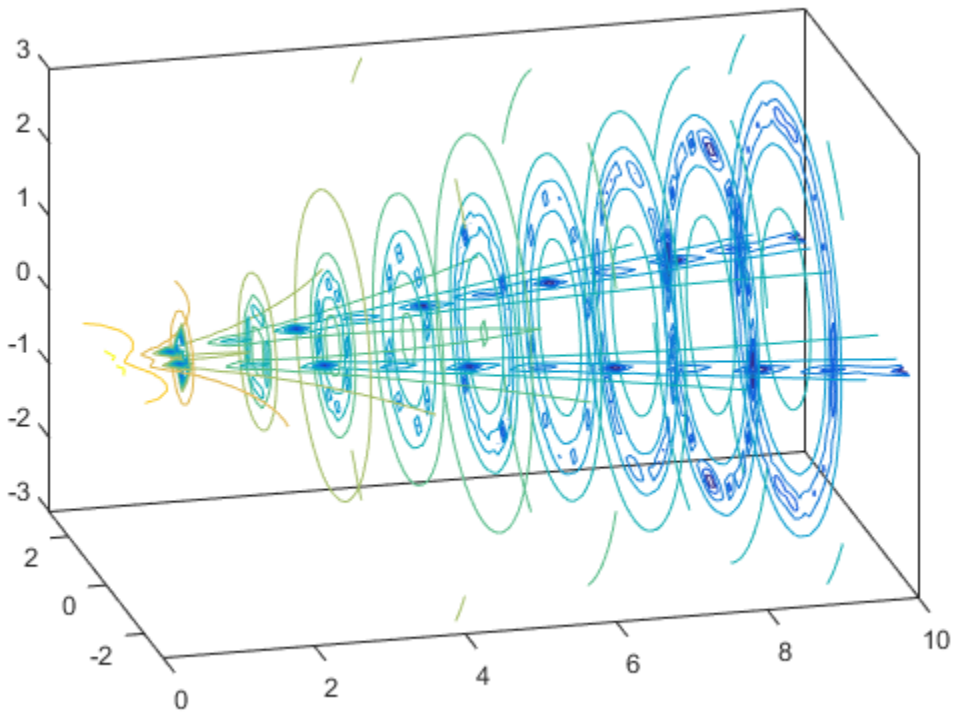
Draw 10 contour lines between -8 and 2 by specifying `cvals` as a 10-element vector of linearly spaced values between -8 and 2.

```
cvals = linspace(-8,2,10);
```

Create the contour slice plots and set the axis limits. Set the data aspect ratio, change the camera position, and display the black box outline.

```
figure
contourslice(X,Y,Z,V,Sx,Sy,Sz,cvals)
```

```
axis([0,10,-3,3,-3,3])
daspect([1,1,1])
campos([0,-20,7])
box on
```



Contour Slices Along Spherical Surface

Set up matrices `X`, `Y`, and `Z` using the `meshgrid` function.

```
x = -2:0.2:2;
y = -2:0.25:2;
z = -2:0.16:2;
[X,Y,Z] = meshgrid(x,y,z);
```

Use `X`, `Y`, and `Z` to define `V` as a matrix of volume data.

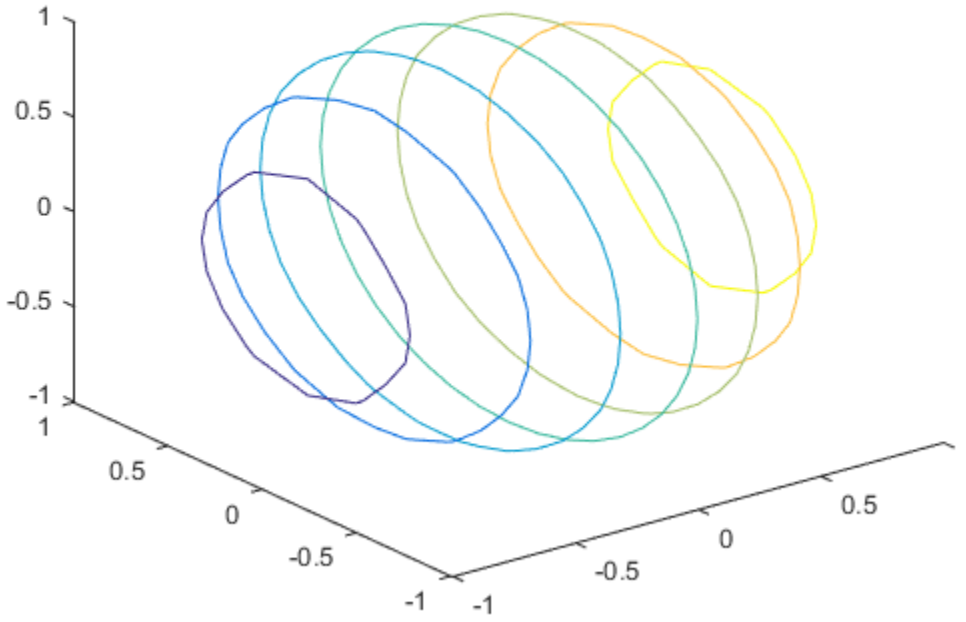
```
V = X.*exp(-X.^2-Y.^2-Z.^2);
```

Return matrices `Xi`, `Yi`, and `Zi` from the `sphere` function.

```
[Xi,Yi,Zi] = sphere;
```

Draw contours through the volume `V` along the surface defined by `Xi`, `Yi`, and `Zi`. Change the plot view to a 3-D view.

```
contourslice(X,Y,Z,V,Xi,Yi,Zi)
view(3)
```



See Also

`isosurface` | `slice` | `smooth3` | `subvolume` | `reducevolume`

Introduced before R2006a

matlab.unittest.constraints

Summary of classes in MATLAB Constraints Interface

Description

Constraints specify business rules against which to qualify a calculated value. Use constraints in conjunction with the `matlab.unittest.qualifications` qualification methods `assertThat`, `assumeThat`, `fatalAssertThat`, or `verifyThat`. Constraints determine whether or not a calculated (actual) value satisfies the constraint. Constraints also provide diagnostics. The `matlab.unittest.constraints` package consists of the following classes.

Constraint Implementations

Fundamental Constraint-Related Interfaces

matlab.unittest.constraints.BooleanConstraint	Interface class for boolean combinations of constraints
matlab.unittest.constraints.Constraint	Fundamental interface class for comparisons

General Purpose

matlab.unittest.constraints.Eventually	Poll for value to asynchronously satisfy constraint
matlab.unittest.constraints.HasField	Constraint specifying structure containing particular field

matlab.unittest.constraints.IsAnything
: Constraint specifying any value

matlab.unittest.constraints.IsFalse
: Constraint specifying false value

matlab.unittest.constraints.IsEqualTo
: General constraint to compare for equality

matlab.unittest.constraints.IsSameHandleAs
: Constraint specifying handle instance same as another

matlab.unittest.constraints.IsTrue
: Constraint specifying true value

matlab.unittest.constraints.ReturnsTrue
: Constraint specifying function handle that returns true

Errors and Warnings

matlab.unittest.constraints.IssuesNoWarnings
: Constraint specifying function that issues no warnings

matlab.unittest.constraints.IssuesWarnings
: Constraint specifying function that issues expected warning profile

matlab.unittest.constraints.Throws
: Constraint specifying function handle that throws MException

Inequalities

matlab.unittest.constraints.IsGreaterThan
: Constraint specifying value greater than another value

matlab.unittest.constraints.IsGreaterThanOrEqualTo
: Constraint specifying value greater than or equal to another value

matlab.unittest.constraints.IsLessThan
: Constraint specifying value less than another value

matlab.unittest.constraints.IsLessThanOrEqualTo
: Constraint specifying value less than or equal to another value

Array Size

matlab.unittest.constraints.HasElementCount
: Constraint specifying expected number of elements

matlab.unittest.constraints.HasLength
: Constraint specifying expected length of array

matlab.unittest.constraints.HasSize
: Constraint specifying expected size of array

matlab.unittest.constraints.IsEmpty
: Constraint specifying empty value

matlab.unittest.constraints.IsScalar
: Constraint specifying scalar value

Type

matlab.unittest.constraints.IsInstanceOf
: Constraint specifying inclusion in given class hierarchy

matlab.unittest.constraints.IsOfClass
: Constraint specifying class type

Strings

matlab.unittest.constraints.ContainsSubstring
: Constraint specifying string containing substring

matlab.unittest.constraints.EndsWithSubstring
: Constraint specifying string ending with substring

matlab.unittest.constraints.IsSubstringOf
: Constraint specifying substring of another string

matlab.unittest.constraints.Matches
: Constraint specifying string matches regular expression

matlab.unittest.constraints.StartsWithSubstring
: Constraint specifying string starting with substring

Finiteness

matlab.unittest.constraints.HasInf
: Constraint specifying array containing any infinite value

matlab.unittest.constraints.HasNaN
: Constraint specifying array containing NaN value

matlab.unittest.constraints.IsFinite
: Constraint specifying finite value

Numeric Attributes

matlab.unittest.constraints.IsReal
: Constraint specifying real valued array

matlab.unittest.constraints.IsSparse
: Constraint specifying sparse array

Actual Value Proxies

matlab.unittest.constraints.AnyCellOf
: Test if any element of cell array meets constraint

matlab.unittest.constraints.AnyElementOf
: Test if any element of array meets constraint

matlab.unittest.constraints.EveryCellOf
: Test if all elements of cell array meet constraint

matlab.unittest.constraints.EveryElementOf
: Test if all elements of array meet constraint

Tolerances

matlab.unittest.constraints.AbsoluteTolerance
: Absolute numeric tolerance

matlab.unittest.constraints.RelativeTolerance
: Relative numeric tolerance

matlab.unittest.constraints.Tolerance
: Abstract interface class for tolerances

Comparators

matlab.unittest.constraints.CellComparator
: Comparator for cell arrays

matlab.unittest.constraints.LogicalComparator
: Comparator for two logical values

matlab.unittest.constraints.NumericComparator
: Comparator for numeric data types

matlab.unittest.constraints.ObjectComparator
: Comparator for MATLAB or Java objects

matlab.unittest.constraints.PublicPropertyComparator
: Comparator for public properties of MATLAB objects

matlab.unittest.constraints.StringComparator
: Comparator for two strings

matlab.unittest.constraints.StructComparator

Comparator for MATLAB structure arrays

matlab.unittest.constraints.AbsoluteTolerance class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Tolerance

Absolute numeric tolerance

Description

This numeric `Tolerance` assesses the magnitude of the difference between actual and expected values. For the tolerance to be satisfied, `abs(expVal - actVal) <= absTol` must be true.

Construction

`AbsoluteTolerance(tolVals)` creates an absolute tolerance object that assesses the magnitude of the difference between the actual and expected values.

The data types of the inputs to the `AbsoluteTolerance` constructor determines which data types the tolerance supports. For example, `AbsoluteTolerance(10*eps)` constructs an `AbsoluteTolerance` for comparing double-precision numeric arrays, while `AbsoluteTolerance(int8(2))` constructs an `AbsoluteTolerance` for comparing numeric arrays of type `int8`. If the actual and expected values being compared contain more than one numeric data type, the tolerance only applies to the data types specified by the values passed into the constructor.

To specify different tolerance values for different data types, you can pass multiple tolerance values to the constructor. For example, `AbsoluteTolerance(10*eps, 10*eps('single'), int8(1))` constructs an `AbsoluteTolerance` object applies the following absolute tolerances:

- `10*eps` applies an absolute tolerance of `10*eps` for double-precision numeric arrays.
- `10*eps('single')` applies an absolute tolerance of `10*eps` for single-precision numeric arrays.
- `int8(1)` applies an absolute tolerance of `1` for numeric arrays of type `int8`.

You can specify more than one tolerance for a particular data type by combining tolerances with the & and | operators. To combine two tolerances, the sizes of the tolerance values for each data type must be compatible.

Input Arguments

`tolVals`

Numeric tolerances, specified as a comma-separated list of numeric arrays. Each input argument contains the tolerance specification for a particular data type. Each numeric array can be a scalar or array the same size as the actual and expected values.

Default:

Properties

`Values`

Numeric tolerances, specified by the `tolVals` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test with Absolute Tolerance

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance

testCase = TestCase.forInteractiveUse;
```

Assert that the difference between an actual value, `4.1`, and an expected value, `4.5`, is less than `0.5`.

```
testCase.assertThat(4.1, IsEqualTo(4.5, ...
    'Within', AbsoluteTolerance(0.5)))
```

```
Interactive assertion passed.
```

Specify Absolute Tolerance for Different Data Types

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance

testCase = TestCase.forInteractiveUse;
```

Create the following actual and expected cell arrays.

```
act = {'abc', 123, single(106), int8([1, 2, 3])};
exp = {'abc', 122, single(105), int8([2, 4, 6])};
```

Test whether the arrays satisfy the `AbsoluteTolerance` constraint within a value of `2`.

```
testCase.verifyThat(act, IsEqualTo(exp, ...
    'Within', AbsoluteTolerance(2)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    Path to failure: <Value>{3}
    --> The values are not equal using "isequaln".
    --> The tolerance does not support single values so it was not used.
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        106       105         1        0.00952381

    Actual Value:
```

```
           106
    Expected Value:
           105

Actual cell:
        'abc'    [123]    [106]    [1x3 int8]
Expected cell:
        'abc'    [122]    [105]    [1x3 int8]
```

The test fails because the tolerance is only applied to the `double` data type.

Create a tolerance object that specifies different tolerances for different data types.

```
tolObj = AbsoluteTolerance(2, single(3), int8([2, 3, 5]));
```

A tolerance of `2` is a applied to `double` valued data. A tolerance of `3` is applied to `single` valued data. A tolerance of `[2 3 5]` is applied to corresponding array elements of `int8` valued data.

Verify that the expected and actual values satisfy the `AbsoluteTolerance` constraint.

```
testCase.verifyThat(act, IsEqualTo(exp, 'Within', tolObj))
```

```
Interactive verification passed.
```

Combine Absolute and Relative Tolerances

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Define an actual value approximation for `pi`.

```
act = 3.14;
```

Construct a tolerance object to test that the difference between the actual and expected values is within `0.001` and within `0.25%`.

```
tolObj = AbsoluteTolerance(0.001) & RelativeTolerance(0.0025);
```

Verify that the actual value is within the tolerance of the expected value of `pi`.

```
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> AndTolerance failed.
        --> AbsoluteTolerance failed.
            --> The error was not within absolute tolerance.
        --> RelativeTolerance passed.
        --> Failure table:
                    Index    Actual        Expected               Error
                    _____    ______    _________________    ____________________    ____

                    1        3.14      3.14159265358979    -0.00159265358979299    -0.0

Actual Value:
       3.140000000000000
Expected Value:
       3.141592653589793
```

The actual value does not satisfy the `AbsoluteTolerance` constraint.

Construct a constraint that is satisfied if the values are within `0.001` or `0.25%`, and then retest the actual value.

```
tolObj = AbsoluteTolerance(0.001) | RelativeTolerance(0.0025);
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))

Interactive verification passed.
```

Combine Absolute and Relative Tolerances to Test Small and Large Values

Combine tolerances so when you test the equality of values, an absolute (floor) tolerance dominates when the values are near zero, and a relative tolerance dominates for larger values.

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
```

```
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Define two structures containing electromagnetic properties of a vacuum. One structure, `approxVacuumProps`, contains approximate values for the permeability and speed of light in a vacuum.

```
approxVacuumProps.Permeability = 1.2566e-06; % Approximate
approxVacuumProps.Permitivity = 8.854187817*10^-12;
approxVacuumProps.LightSpeed = 2.9979e+08; % Approximate

baselineVacuumProps.Permeability = 4*pi*10^-7;
baselineVacuumProps.Permitivity = 8.854187817*10^-12;
baselineVacuumProps.LightSpeed = 1/sqrt(...
    baselineVacuumProps.Permeability*baselineVacuumProps.Permitivity);
```

Test that the relative difference between the approximate and baseline values is within `eps*1e11`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    Path to failure: <Value>.Permeability
    --> The values are not equal using "isequaln".
    --> RelativeTolerance failed.
        --> The error was not within relative tolerance.
        --> Failure table:
                    Index      Actual            Expected                  Error
                    _____    __________    ______________________    ____________________

                    1        1.2566e-06    1.25663706143592e-06    -3.70614359173257e-

    Actual Value:
             1.256600000000000e-06
    Expected Value:
             1.256637061435917e-06
```

```
Actual struct:
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 299790000
Expected struct:
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the relative difference in the permeabilities is not within the tolerance. The difference between the two values is small, but the numbers are close to zero, so the difference relative to their size is not small enough to satisfy the tolerance.

Construct a tolerance object to test that the absolute difference between the approximate and baseline values is within `1e-4`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', AbsoluteTolerance(1e-4)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    Path to failure: <Value>.LightSpeed
    --> The values are not equal using "isequaln".
    --> AbsoluteTolerance failed.
        --> The error was not within absolute tolerance.
        --> Failure table:
                    Index     Actual          Expected              Error             R
                    _____    _________    _________________    _________________    ___

                    1        299790000    299792458.010503    -2458.01050287485    -8.1

    Actual Value:
           299790000
    Expected Value:
             2.997924580105029e+08

Actual struct:
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
```

```
          LightSpeed: 299790000
Expected struct:
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the absolute difference in the speed of light is not within the tolerance. The difference between the two values is small relative to their size, but too large to satisfy the tolerance.

Construct a logical disjunction of tolerance objects to test that the absolute difference between the approximate and baseline values is within `1e-4` or the relative difference is within `eps*1e11`. The test uses this tolerance so permeability values that are close to zero satisfy the absolute (floor) tolerance, and speed of light values that are large, satisfy the relative tolerance.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)| AbsoluteTolerance(1e-4)))
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.RelativeTolerance` | `matlab.unittest.constraints.IsEqualTo`

matlab.unittest.constraints.AnyCellOf class

Package: matlab.unittest.constraints

Test if any element of cell array meets constraint

Description

The `AnyCellOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of a cell array, which ensures that a passing result occurs if at least one element of the cell array satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`AnyCellOf(actVal)` creates a proxy instance that tests if any element of a provided cell array, `actVal`, meets a constraint. The test passes if at least one element individually satisfies the constraint.

Input Arguments

`actVal`

Actual value to test against constraint

Properties

`ActualValue`

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Any Cell Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.AnyCellOf

testCase = TestCase.forInteractiveUse;
```

Test that at least one cell of `actVal` is finite.

```
import matlab.unittest.constraints.IsFinite
actVal = {NaN, Inf, 5};
testCase.verifyThat(AnyCellOf(actVal), IsFinite)
```

```
Interactive verification passed.
```

Test that at least one cell of the actual value contains five elements.

```
import matlab.unittest.constraints.HasElementCount
testCase.verifyThat(AnyCellOf({42, [11 38], 1:5}), HasElementCount(5))
```

```
Interactive verification passed.
```

Test that at least one cell of the actual value matches the string `'tea'` regardless of case.

```
import matlab.unittest.constraints.Matches
testCase.verifyThat(AnyCellOf({'Coffee','Tea','Water'}), ...
    Matches('tea','IgnoringCase',true))
```

```
Interactive verification passed.
```

Test that at least one cell of the actual value is less than zero.

```
import matlab.unittest.constraints.IsLessThan
```

```
testCase.verifyThat(AnyCellOf({1, 5}), IsLessThan(0))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
All cells failed. The first cell failed because:
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual Value:
             1
    Maximum Value (Exclusive):
             0

Actual Value Cell Array:
       [1]    [5]
```

Neither actual value element is less than zero.

Test that neither cell of the actual value is empty.

```
import matlab.unittest.constraints.IsEmpty
testCase.verifyThat(AnyCellOf({inputParser.empty,''}), ~IsEmpty)

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
All cells failed. The first cell failed because:
--> Negated IsEmpty failed.
    --> The value must not be empty.
    --> The value has a size of [0  0].

    Actual Value:
          0x0 inputParser array with properties:

            FunctionName
            CaseSensitive
            KeepUnmatched
            PartialMatching
            StructExpand
            Parameters
```

```
            Results
            Unmatched
            UsingDefaults

Actual Value Cell Array:
        []    ''
```

Both actual value elements are empty.

See Also

AnyElementOf | EveryCellOf | EveryElementOf | matlab.unittest.qualifications

matlab.unittest.constraints.AnyElementOf class

Package: matlab.unittest.constraints

Test if any element of array meets constraint

Description

The `AnyElementOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of an array, which ensures that a passing result occurs when at least one element of the array satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`AnyElementOf(actVal)` creates a proxy instance that tests if any element of a provided array, `actVal`, meets a constraint. The test passes if at least one element individually satisfies the constraint.

Tips

- `AnyElementOf` checks if any element in the provided array satisfies an associated constraint. However, there are some constraints, such as `HasNaN` and `HasInf`, that natively validate if any of the elements satisfy a condition. In these situations, use of `AnyElementOf` is unnecessary and impedes qualification performance.

Input Arguments

`actVal`

Actual value to test against constraint

Properties

`ActualValue`

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Any Element Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.AnyElementOf

testCase = TestCase.forInteractiveUse;
```

Test that at least one element of `actVal` is finite.

```
import matlab.unittest.constraints.IsFinite
actVal = [NaN, Inf, 5];
testCase.verifyThat(AnyElementOf(actVal), IsFinite)
```

```
Interactive verification passed.
```

Test that at least one element of the actual value is complex.

```
import matlab.unittest.constraints.IsReal
testCase.verifyThat(AnyElementOf([1+0i 4i]), ~IsReal)
```

```
Interactive verification passed.
```

Test that at least one element of the actual value array is less than zero.

```
import matlab.unittest.constraints.IsLessThan
```

```
testCase.verifyThat(AnyElementOf([1 5]), IsLessThan(0))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
All elements failed. The first element failed because:
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual Value:
             1
    Maximum Value (Exclusive):
             0

Actual Value Array:
         1     5
```

Neither actual value element is less than zero.

See Also

AnyCellOf | EveryCellOf | EveryElementOf | matlab.unittest.qualifications

matlab.unittest.constraints.BooleanConstraint class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Interface class for boolean combinations of constraints

Description

The `BooleanConstraint` interface class provides an interface for boolean combinations of `Constraints`. Any constraint that derives from `BooleanConstraint` can be combined and negated using the `and` (`&`), `or` (`|`), and `not` (`~`) operators.

Classes that derive from the `BooleanConstraint` interface class must implement everything required by the standard `Constraint` interface. When a given constraint is negated, the diagnostics must be written in a different form than for a standard (non-negated) failure. Therefore, classes deriving from the `BooleanConstraint` class must implement a method to provide a `Diagnostic` object for the negated case, in addition to the non-negated case.

In exchange for meeting these requirements, all `BooleanConstraint` implementations inherit the appropriate MATLAB overloads for `and`, `or`, and `not` so that they can be combined with other `BooleanConstraint` objects or negated.

Methods

getNegativeDiagnosticFor	Produce negated diagnostic for value

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Boolean HasSameSizeAs Constraint

Create a custom constraint that determines if a given value is the same size as an expected value.

In a file in your working folder, create a file `HasSameSizeAs.m`. The constructor accepts a value to compare to the actual size. This value is stored within the `ValueWithExpectedSize` property. It is recommended that `BooleanConstraint` implementations be immutable, so set the property `SetAccess=immutable`.

```
classdef HasSameSizeAs < matlab.unittest.constraints.BooleanConstraint

    properties(SetAccess=immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
    end
end
```

Include these methods in the `methods` block in `HasSameSizeAs.m`. Since the `BooleanConstraint` class is a subclass of `Constraint`, classes that derive from it must implement the `satisfiedBy` and `getDiagnosticFor` methods. For more information about these methods, see `matlab.unittest.constraints.Constraint`.

```
    methods
        function bool = satisfiedBy(constraint, actual)
            bool = isequal(size(actual), size(constraint.ValueWithExpectedSize));
        end
        function diag = getDiagnosticFor(constraint, actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.satisfiedBy(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
            else
                diag = StringDiagnostic(sprintf(...
                    'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                    int2str(size(actual)),...
                    int2str(size(constraint.ValueWithExpectedSize))));
            end
        end
    end
```

Include the `getNegativeDiagnosticFor` method in the a `methods` block with protected access in `HasSameSizeAs.m`. Classes that derive from `BooleanConstraint` must implement the `getNegativeDiagnosticFor` method. This method must provide a `Diagnostic` object that is expressed in the negative sense of the constraint.

```
    methods(Access=protected)
        function diag = getNegativeDiagnosticFor(constraint, actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.satisfiedBy(actual)
                diag = StringDiagnostic(sprintf(...
                    ['Negated HasSameSizeAs failed.\nSize [%s] of ' ...
                    'Actual Value and Expected Value were the same ' ...
                    'but should not have been.'], int2str(size(actual))));
            else
                diag = StringDiagnostic('Negated HasSameSizeAs passed.');
            end
        end
    end
```

In exchange for implementing the required methods, the constraint inherits the appropriate `and`, `or`, and `not` overloads so it can be combined with other `BooleanConstraint` objects or negated.

HasSameSizeAs Class Definition Summary

```
classdef HasSameSizeAs < matlab.unittest.constraints.BooleanConstraint
    properties(SetAccess=immutable)
        ValueWithExpectedSize
    end
    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
        function bool = satisfiedBy(constraint, actual)
            bool = isequal(size(actual), size(constraint.ValueWithExpectedSize));
        end
        function diag = getDiagnosticFor(constraint, actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.satisfiedBy(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
            else
                diag = StringDiagnostic(sprintf(...
                    'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                    int2str(size(actual)), ...
                    int2str(size(constraint.ValueWithExpectedSize))));
            end
        end
    end
    methods(Access=protected)
        function diag = getNegativeDiagnosticFor(constraint, actual)
            import matlab.unittest.diagnostics.StringDiagnostic
            if constraint.satisfiedBy(actual)
                diag = StringDiagnostic(sprintf(...
                    ['Negated HasSameSizeAs failed.\nSize [%s] of ' ...
                    'Actual Value and Expected Value were the same ' ...
                    'but should not have been.'], int2str(size(actual))));
            else
                diag = StringDiagnostic('Negated HasSameSizeAs passed.');
            end
        end
    end
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasLength

testCase = TestCase.forInteractiveUse;
```

Test a passing case.

```
testCase.verifyThat(zeros(5), HasLength(5) | ~HasSameSizeAs(repmat(1,5)))

Interactive verification passed.
```

The test passes because one of the `or` conditions, `HasLength(5)`, is true.

Test a failing case.

```
testCase.verifyThat(zeros(5), HasLength(5) & ~HasSameSizeAs(repmat(1,5)))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
AndConstraint failed.
--> + [First Condition]:
     |   HasLength passed.
--> AND
    + [Second Condition]:
     |   Negated HasSameSizeAs failed.
     |   Size [5  5] of Actual Value and Expected Value were the same but should not ha
    -+---------------------
```

The test fails because one of the `and` conditions, `~HasSameSizeAs(repmat(1,5))`, is false.

Boolean Combinations of Constraints

At the command prompt, create a test case for interactive testing and import several classes that subclass `BooleanConstraint`.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasElementCount
import matlab.unittest.constraints.HasLength
import matlab.unittest.constraints.HasInf
```

```
import matlab.unittest.constraints.HasNaN
import matlab.unittest.constraints.IsEmpty
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.IsGreaterThanOrEqualTo
import matlab.unittest.constraints.IsOfClass
import matlab.unittest.constraints.IsReal

testCase = TestCase.forInteractiveUse;
```

Test these passing cases.

```
testCase.verifyThat(3, IsReal & IsGreaterThanOrEqualTo(3))
testCase.verifyThat([1 2 3; 4 5 6], HasLength(3) & HasElementCount(6))
testCase.verifyThat([3 NaN 5], HasNaN | HasInf)
testCase.verifyThat(3, ~IsEqualTo(4))
testCase.verifyThat('Some char', IsOfClass(?char) & ~IsEmpty)
```

See Also

`Diagnostic` | `matlab.unittest.constraints`

getNegativeDiagnosticFor

Produce negated diagnostic for value

Syntax

```
diag = getNegativeDiagnosticFor(constObj, actVal)
```

Description

`diag = getNegativeDiagnosticFor(constObj, actVal)` produces a negated diagnostic for a value. The `getNegativeDiagnosticFor` method analyzes the provided value, `actVal`, against the constraint, `constObj`, and produces a `matlab.unittest.diagnostics.Diagnostic` object, `diag`, which corresponds to the negation of the constraint, `constObj`. This method is a protected method.

The diagnostics that this method produces are expressed in the negative sense of the constraint. For example, a hypothetical `IsTasty` constraint, when negated, should express that the actual value was "tasty", when it should not have been, and it should describe the details on why it was found to be tasty.

Like the `getDiagnosticFor` method of `Constraint`, the `getNegativeDiagnosticFor` is only called upon failures, and thus can afford a more detailed analysis than the `satisfiedBy` method.

Input Arguments

`constObj`

`BooleanConstraint` instance

`actVal`

Value for comparison

Examples

Implement `getNegativeDiagnosticFor` method

```
function diag = getNegativeDiagnosticFor(constraint, actual)
% getNegativeDiagnosticFor - produce a diagnostic when the constraint is
% incorrectly met
%
%   This method is called by the testing framework when the constraint has
%   been met but should not have been met because it was negated in a
%   boolean expression. It should produce a Diagnostic result that
%   describes the failure in the correct terms which express the
%   requirement that the constraint actually should not have been met.

import matlab.unittest.diagnostics.StringDiagnostic

if constraint.satisfiedBy(actual)
    % Create the negative diagnostic. This will show information such as the
    % constraint class name and display the raw actual and expected values.
    % Using the DiagnosticSense.NegativeDiagnostic enumeration also
    % produces language more appropriate for the negated case.
    diag = StringDiagnostic(sprintf(...
        ['Negated HasSameSizeAs failed.\nSize [%s] of ' ...
        'Actual Value and Expected Value were the same ' ...
        'but should not have been.', int2str(size(actual))));
else
    % Produce a passing diagnostic, with language appropriate for
    % the negated case.
    diag = StringDiagnostic('Negated HasSameSizeAs passed.');
end % if

end % function
```

See Also

`Diagnostic` | `getDiagnosticFor` | `satisfiedBy`

matlab.unittest.constraints.CellComparator class

Package: matlab.unittest.constraints

Comparator for cell arrays

Description

The `CellComparator` compares cell arrays.

Construction

`CellComparator` creates a comparator for cell arrays.

`CellComparator(compObj)` indicates a comparator, `compObj`, that defines the comparator used to compare values contained in the cell array. By default, a cell comparator supports only empty cell arrays.

`CellComparator(compObj,Name,Value)` provides a comparator with additional options specified by one or more Name,Value pair arguments.

`CellComparator(Name,Value)` provides a comparator for empty cell arrays with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`compObj`

Comparator object

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'IgnoringCase'

Indicator whether the comparator is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to case. The comparator uses this name-value pair only if the contents being compared are strings.

Default: `false`

'IgnoringWhitespace'

Indicator whether the comparator is insensitive to whitespace characters, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to whitespace characters. Whitespace characters consist of space, form feed, new line, carriage return, horizontal tab, and vertical tab. The comparator uses this name-value pair only if the contents being compared are strings.

Default: `false`

'Recursively'

Indicator of whether comparator operates recursively, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the comparator does not operate recursively on its data.

When the value is `true`, the data types the cell comparator supports are fully supported in recursion. For example:

```
comp1 = CellComparator(StringComparator)
comp2 = CellComparator(StringComparator,'Recursively', true)
```

Both `comp1` and `comp2` support cell arrays of strings. However, only `comp2` supports cell arrays that recursively contain either cell arrays or strings as their elements.

Default: `false`

'Within'

Tolerance to use for numerical comparison, specified as a `matlab.unittest.constraints.Tolerance` object. The comparator uses this name-value pair only if the contents being compared are numeric types.

Default: (empty)

Properties

`IgnoreCase`

Indicator whether the comparator is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`

`IgnoreWhitespace`

Indicator whether the comparator is insensitive to whitespace characters, specified in the name-value pair argument, `'IgnoringWhitespace'`

`Recursive`

Indicator of whether comparator operates recursively, specified in the name-value pair argument, `'Recursively'`

`Tolerance`

Specific tolerance used in construction of the comparator, specified as a `matlab.unittest.constraints.Tolerance` object in the name-value pair argument, `'Within'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Compare Cell Arrays

Create a test case for interactive testing.

```
import matlab.unittest.constraints.CellComparator
import matlab.unittest.constraints.StringComparator
import matlab.unittest.constraints.IsEqualTo

testCase = matlab.unittest.TestCase;
```

Use a `CellComparator` to test that two cell arrays are equal to each other.

```
actual = {'abc','def'};
expected = {'abc','def'};
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Using', CellComparator(StringComparator)))
```

```
Interactive verification passed.
```

By default, the `CellComparator` supports only comparison of empty cell arrays. Therefore, it is necessary to pass it a `StringComparator`.

Change the actual value and compare it to the expected value. To satisfy the constraint, construct it to ignore case and whitespace characters.

```
actual = {'ABC','D E F'};
testCase.verifyThat(actual, IsEqualTo(expected, 'Using', ...
    CellComparator(StringComparator, ...
    'IgnoringWhitespace', true, 'IgnoringCase',true)))
```

```
Interactive verification passed.
```

Test nested cell arrays of strings by constructing the comparator to operate recursively.

```
actual = {'abc',{'def','ghi'}};
expected = {'abc',{'def','ghi'}};

testCase.verifyThat(actual, IsEqualTo(expected, 'Using', ...
    CellComparator(StringComparator, 'Recursively', true)))
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsEqualTo` |
`matlab.unittest.constraints.Tolerance`

Introduced in R2013a

matlab.unittest.constraints.Constraint class

Package: matlab.unittest.constraints

Fundamental interface class for comparisons

Description

The `Constraint` interface class is the means by which `matlab.unittest` constraints encode comparison logic and the corresponding diagnostic information. Every comparison that conditionally can produce a failure inherits from the `Constraint` interface class.

Classes deriving from the `Constraint` interface class must provide a means to determine if a given value satisfies the constraint. To do this, implement the `satisfiedBy` method, which includes the definition of the underlying comparison logic. Classes deriving from the `Constraint` class also must provide a diagnostic for any given actual value. The testing framework uses the diagnostic when it encounters a qualification failure. To do this, implement the `getDiagnosticFor` method.

In exchange for meeting these requirements, all `Constraint` implementations are easily used with all qualification types through the `verifyThat`, `assertThat`, `assumeThat`, or `fatalAssertThat` methods. The qualifications use the comparison and diagnostic knowledge contained within the constraints. Also, the constraints can be used in situations where a test failure is not wanted, but the testing framework needs to reuse the comparison logic. For example, a constraint implementation may want to use the logic defined inside of another constraint. Since the constraint can interact with the other constraint directly, it can use the logic without the potential of causing a qualification failure.

Methods

getDiagnosticFor	Produce diagnostic for compared value
satisfiedBy	Determine whether value satisfies constraint

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create HasSameSizeAs Constraint

Create a custom constraint that determines if a given value is the same size as an expected value.

In a file in your working folder, create a `HasSameSizeAs.m`. The constructor accepts a value to compare to the actual size. This value is stored within the `ValueWithExpectedSize` property. Since, it is recommended that `Constraint` implementations are immutable, set the property `SetAccess=immutable`.

```
classdef HasSameSizeAs < matlab.unittest.constraints.Constraint

    properties(SetAccess=immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
    end
end
```

Classes that derive from `Constraint` must implement the `satisfiedBy` method. This method must contain the comparison logic and return a `boolean` value.

Include the `satisfiedBy` method in the `methods` block in `HasSameSizeAs.m`.

```
        function bool = satisfiedBy(constraint, actual)
            bool = isequal(size(actual), size(constraint.ValueWithExpectedSize));
        end
```

This method returns `true` if the actual size and expected size are equal.

Classes deriving from `Constraint` must implement the `getDiagnosticFor` method. This method must evaluate the actual value against the constraint and

provide a `Diagnostic` object. In this example, `getDiagnosticFor` returns a `StringDiagnostic`. Include the `getDiagnosticFor` method in the `methods` block in `HasSameSizeAs.m`.

```
        function diag = getDiagnosticFor(constraint, actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.satisfiedBy(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
            else
                diag = StringDiagnostic(sprintf(...
                    'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                    int2str(size(actual)),...
                    int2str(size(constraint.ValueWithExpectedSize))));
            end
        end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase

testCase = TestCase.forInteractiveUse;
```

Test a passing case.

```
testCase.verifyThat(zeros(5), HasSameSizeAs(repmat(1,5)))

Interactive verification passed.
```

Test a failing case.

```
testCase.verifyThat(zeros(5), HasSameSizeAs(ones(1,5)))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
HasSameSizeAs failed.
Actual Size: [5  5]
ExpectedSize: [1  5]
```

See Also

`assertThat` | `assumeThat` | `ConstraintDiagnostic` | `Diagnostic` | `fatalAssertThat` | `matlab.unittest.constraints` | `verifyThat`

getDiagnosticFor

Class: matlab.unittest.constraints.Constraint
Package: matlab.unittest.constraints

Produce diagnostic for compared value

Syntax

```
diag = getDiagnosticFor(constObj,actVal)
```

Description

`diag = getDiagnosticFor(constObj,actVal)` produces a diagnostic, `diag`, for a compared value, `actVal`. When creating a custom constraint, the class author must implement the `getDiagnosticFor` method so that it analyzes the value, `actVal`, against the constraint, `constObj`, and instantiates and returns a `matlab.unittest.diagnostics.Diagnostic` object.

Typically, the testing framework calls this method when it encounters a qualification failure. Therefore, the constraint author can afford to undertake a more detailed analysis in the `getDiagnosticFor` method than the `satisfiedBy` method.

Input Arguments

`actVal`

Value for comparison

`constObj`

`Constraint` instance

Output Arguments

`diag`

`Diagnostic` instance

Examples

See example for `matlab.unittest.constraints.Constraint`.

See Also

`ConstraintDiagnostic` | `Diagnostic` | `satisfiedBy`

satisfiedBy

Class: matlab.unittest.constraints.Constraint
Package: matlab.unittest.constraints

Determine whether value satisfies constraint

Syntax

```
TF = satisfiedBy(constObj,actVal)
```

Description

`TF = satisfiedBy(constObj,actVal)` determines whether a value, `actVal`, satisfies a constraint, `constObj`. The `satisfiedBy` method is used to determine qualification success or failure. It returns `true` or `false` (`logical` `0` or `1`). When creating a custom constraint, a class author must place comparison logic in this method.

Since the most common usage is for the passing case, the constraint author should optimize for speed in that case. It is only in the failing case that more expensive detailed analysis is helpful.

Input Arguments

`actVal`

Value to evaluate against the constraint

`constObj`

`Constraint` instance

Examples

See example for `matlab.unittest.constraints.Constraint`.

See Also

`getDiagnosticFor`

matlab.unittest.constraints.ContainsSubstring class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying string containing substring

Construction

`ContainsSubstring(substring)` creates a constraint that specifies a string containing a substring, `substring`. The constraint is satisfied only if the actual value contains an expected string.

`ContainsSubstring(substring,Name,Value)` provides a constraint with additional options specified by one or more Name,Value pair arguments. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`substring`

Text that must be contained within the actual value, specified as a string. `substring` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoringCase'`

Indicator if the constraint is insensitive to case, specified as `false` or `true` (`logical 0` or `1`)

Default: `false`

`'IgnoringWhitespace'`

Indicator if the constraint is insensitive to whitespace, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

Properties

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`. This property applies at all levels of recursion, such as nested structures.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`. This property applies at all levels of recursion, such as nested structures.

`Substring`

String that must be included in the actual value, specified in the input argument, `substring`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Contains Specified Substring

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
```

```
import matlab.unittest.constraints.ContainsSubstring

testCase = TestCase.forInteractiveUse;
```

Define the actual value string.

```
actVal = 'This Is One Long String';
```

Test the `actVal` contains the substring `'One'`.

```
testCase.verifyThat(actVal, ContainsSubstring('One'))

Interactive verification passed.
```

Test the `actVal` contains the substring `'long'`.

```
testCase.verifyThat(actVal, ContainsSubstring('long'))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
ContainsSubstring failed.
--> The string must contain the substring.

Actual String:
    This Is One Long String
Expected Substring:
    long
```

By default, the `ContainsSubstring` constraint is case sensitive.

Repeat the test ignoring case.

```
testCase.verifyThat(actVal, ContainsSubstring('long',...
    'IgnoringCase', true))

Interactive verification passed.
```

Test `actVal` contains the substring `'thisisone'`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat(actVal, ContainsSubstring('thisisone', ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Interactive verification passed.
```

Assert that `actVal` does not contain the substring `'longer'`.

```
testCase.assertThat(actVal, ~ContainsSubstring('longer',...
    'IgnoringCase', true))
```

```
Interactive verification passed.
```

See Also

`EndsWithSubstring` | `IsSubstringOf` | `Matches` | `StartsWithSubstring`

matlab.unittest.constraints.EndsWithSubstring class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying string ending with substring

Construction

`EndsWithSubstring(suffix)` creates a constraint specifying a string ending with a substring. The constraint is satisfied only if the actual value ends with an expected string, `suffix`.

`EndsWithSubstring(suffix,Name,Value)` provides a constraint with additional options specified by one or more Name,Value pair arguments. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`suffix`

Text that occurs at the end of the actual value, specified as a string. `suffix` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoringCase'`

Indicator if the constraint is insensitive to case, specified as `false` or `true` (`logical 0` or `1`)

Default: `false`

`'IgnoringWhitespace'`

Indicator if the constraint is insensitive to whitespace, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

Properties

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`. This property applies at all levels of recursion, such as nested structures.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`. This property applies at all levels of recursion, such as nested structures.

`Suffix`

Text that occurs at the end of the actual value, specified in the input argument, `suffix`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Ends with Specified Substring

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.EndsWithSubstring
```

```
testCase = TestCase.forInteractiveUse;
```

Define the actual value string.

```
actVal = 'This Is One Long String';
```

Test the `actVal` ends with the substring `'String'`.

```
testCase.verifyThat(actVal, EndsWithSubstring('String'))
```

```
Interactive verification passed.
```

Test the `actVal` ends with the substring `'InG'`.

```
testCase.verifyThat(actVal, EndsWithSubstring('InG'))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
EndsWithSubstring failed.
--> The string has an incorrect suffix.

Actual String:
    This Is One Long String
Expected Suffix:
    InG
```

By default, the `EndsWithSubstring` constraint is case sensitive.

Repeat the test ignoring case.

```
testCase.verifyThat(actVal, EndsWithSubstring('InG',...
    'IgnoringCase', true))
```

```
Interactive verification passed
```

Test the `actVal` ends with the substring `'longstring'`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat(actVal, EndsWithSubstring('longstring', ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Interactive verification passed.
```

Assert that `actVal` does not end with the substring `'long'`.

```
testCase.assertThat(actVal, ~EndsWithSubstring('long'))
```

```
Interactive verification passed.
```

See Also

`ContainsSubstring` | `IsSubstringOf` | `Matches` | `StartsWithSubstring`

matlab.unittest.constraints.Eventually class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Poll for value to asynchronously satisfy constraint

Construction

`outConstObj = Eventually(constObj)` creates a constraint, `outConstObj`, that polls for an actual value function handle to asynchronously satisfy the constraint specified in the `constObj` constraint. It is not satisfied if evaluation of the function handle does not produce a value that satisfies the constraint within 20 seconds. The testing framework invokes the `drawnow` function while the `Eventually` constraint waits for specified function to satisfy the constraint.

`outConstObj = Eventually(constObj,'WithTimeoutOf',timeOutVal)` creates a constraint that polls for the constraint to be satisfied within the timer period specified in `timeOutVal`.

Input Arguments

`constObj`

`Constraint` instance

`timeOutVal`

Maximum time to wait for passing behavior, specified in seconds

Default: `20` seconds

Properties

`TimeOut`

Maximum time to wait for passing behavior, specified by the `timeOutVal` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Verify Test Passes Eventually

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.Eventually
import matlab.unittest.constraints.IsGreaterThan
import matlab.unittest.constraints.IsLessThan

testCase = TestCase.forInteractiveUse;
```

Verify that, within the timeout period, a call to `toc` results in a value greater than 10 (seconds). The `Eventually` constraint repeatedly calls `toc` until either the constraint is satisfied or the elapsed time exceeds the timeout period. Repeated calls to `toc` result in the elapsed time since the last call to `tic`.

```
tic
testCase.verifyThat(@toc, Eventually(IsGreaterThan(10)))
```

```
Interactive verification passed.
```

The verification may take as long as 10 seconds for `toc` to reach a passing value. If you issue the call to `tic` and wait more than 10 seconds before issuing the `verifyThat` command, the verification returns immediately since `toc` already returns a value greater than 10.

Verify that, within the timeout period, `toc` does not return a negative value.

```
testCase.verifyThat(@toc, Eventually(IsLessThan(0)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Eventually failed.
--> The function did not pass after 20 seconds.
```

```
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual Value:
          20.286452356691139
    Maximum Value (Exclusive):
             0

Evaluated Function:
        @toc
```

This failure is expected since elapsed time is not going to be less than zero. However, `Eventually` polls `toc` for the duration of the timeout period.

Adjust the timeout period so `Eventually` polls for 5 seconds.

```
tic
testCase.verifyThat(@toc, Eventually(IsGreaterThan(10), ...
    'WithTimeoutOf', 5))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Eventually failed.
--> The function did not pass after 5 seconds.
--> IsGreaterThan failed.
    --> The value must be greater than the minimum value.

    Actual Value:
           5.076293030150061
    Minimum Value (Exclusive):
             10

Evaluated Function:
        @toc
```

If you didn't wait more than 5 seconds between calls to `tic` and `verifyThat`, the test fails because the elapsed time is not greater than 10 seconds within the modified timeout period.

See Also

`matlab.unittest.constraints.Constraint` | `drawnow`

matlab.unittest.constraints.EveryCellOf class

Package: matlab.unittest.constraints

Test if all elements of cell array meet constraint

Description

The `EveryCellOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of a cell array, which ensures that a passing result occurs when every element of the cell array satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`EveryCellOf(actVal)` creates a proxy instance that tests if every element of a provided cell array, `actVal`, meets a constraint. The test passes if all elements satisfy the constraint.

Tips

- `EveryCellOf` checks if every element in the provided cell array satisfies an associated constraint. However, there are some constraints, a prominent one being `IsEqualTo`, that natively validate if all elements in cell arrays satisfy a condition. In these situations, use of `EveryCellOf` is unnecessary and impedes qualification performance.

Input Arguments

`actVal`

Actual value to test against constraint

Properties

`ActualValue`

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Every Cell Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.EveryCellOf

testCase = TestCase.forInteractiveUse;
```

Test that every cell of `actVal` contains the substring `'ain'`.

```
import matlab.unittest.constraints.ContainsSubstring
actVal = {'Rain','Main','Plain'};
testCase.verifyThat(EveryCellOf(actVal), ContainsSubstring('ain'))
```

```
Interactive verification passed.
```

Test that every cell of the actual value array has two elements.

```
import matlab.unittest.constraints.HasElementCount
testCase.verifyThat(EveryCellOf({{'hello','world'}, {11 38}}), HasElementCount(2))
```

```
Interactive verification passed.
```

Test that every cell of the actual value array is empty.

```
import matlab.unittest.constraints.IsEmpty
```

```
testCase.verifyThat(EveryCellOf({inputParser.empty,''}), IsEmpty)

Interactive verification passed.
```

Test that every cell of the actual value array is finite.

```
import matlab.unittest.constraints.IsFinite
testCase.verifyThat(EveryCellOf({NaN, Inf, 5}), IsFinite)

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
At least one cell failed.

Failing indices:
         1     2
The first failing cell failed because:
--> IsFinite failed.
    --> The value must be finite.

    Actual Value:
           NaN

Actual Value Cell Array:
        [NaN]    [Inf]    [5]
```

Only the third element has a finite value.

Test that every cell of the actual value array is real.

```
import matlab.unittest.constraints.IsReal
testCase.verifyThat(EveryCellOf({1 4i}), IsReal)

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
At least one cell failed.

Failing indices:
         2
The first failing cell failed because:
```

```
--> IsReal failed.
    --> The value must be real.

    Actual Value:
          0.000000000000000 + 4.000000000000000i

Actual Value Cell Array:
        [1]    [0.000000000000000 + 4.000000000000000i]
```

The second element has an imaginary value.

See Also

`AnyCellOf` | `AnyElementOf` | `EveryElementOf` | `matlab.unittest.qualifications`

matlab.unittest.constraints.EveryElementOf class

Package: matlab.unittest.constraints

Test if all elements of array meet constraint

Description

The `EveryElementOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of an array, which ensures that a passing result occurs when every element of the array that satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`EveryElementOf(actVal)` creates a proxy instance that tests if every element of a provided array, `actVal`, meets a constraint. The test passes if all elements satisfy the constraint.

Tips

- `EveryElementOf` checks if every element in the provided array satisfies an associated constraint. However, there are some constraints, such as `IsEqualTo` and `IsGreaterThan`, `IsLessThan`, that natively validate if all elements in the array satisfy a condition. In these situations, use of `EveryElementOf` is unnecessary and impedes qualification performance.

Input Arguments

`actVal`

Actual value to test against constraint

Properties

`ActualValue`

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Every Element Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.EveryElementOf

testCase = TestCase.forInteractiveUse;
```

Test that every element of `actVal` is less than 55.

```
import matlab.unittest.constraints.IsLessThan
actVal = [1 1 2 3 5 8 13 21 34];
testCase.verifyThat(EveryElementOf(actVal), IsLessThan(55))
```

```
Interactive verification passed.
```

Test that every element of the actual value array is complex.

```
import matlab.unittest.constraints.IsReal
testCase.verifyThat(EveryElementOf([1+2i 4i]), ~IsReal)
```

```
Interactive verification passed.
```

Test that every element of the actual value array is less than zero.

```
import matlab.unittest.constraints.IsLessThan
```

```
testCase.verifyThat(EveryElementOf([1 -5]), IsLessThan(0))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
At least one element failed.

Failing indices:
         1
The first failing element failed because:
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual Value:
             1
    Maximum Value (Exclusive):
             0

Actual Value Array:
         1    -5
```

Only the second element is less than zero.

Test that every element of the actual value array has a `NaN` value.

```
import matlab.unittest.constraints.HasNaN
testCase.verifyThat(EveryElementOf([NaN 0/0 5]), HasNaN)

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
At least one element failed.

Failing indices:
         3
The first failing element failed because:
--> HasNaN failed.
    --> The value must be NaN.

    Actual Value:
             5
```

```
Actual Value Array:
       NaN   NaN     5
```

Only the third element has a `NaN` value.

See Also

`AnyCellOf` | `AnyElementOf` | `EveryCellOf` | `matlab.unittest.qualifications`

matlab.unittest.constraints.HasElementCount class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying expected number of elements

Construction

`HasElementCount(countVal)` provides a constraint that specifies an expected number of elements. The constraint is satisfied if the actual value array has the same number of elements specified as by `countVal`.

Input Arguments

`countVal`

Number of elements a value must have to satisfy the constraint.

Properties

`Count`

Number of elements a value must have to satisfy the constraint. Set this property through the constructor via the `countVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test for Expected Number of Elements

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasElementCount

testCase = TestCase.forInteractiveUse;
```

Verify a scalar has an element count of one.

```
testCase.verifyThat(3, HasElementCount(1))

Interactive verification passed.
```

Test the element count of the vector.

```
testCase.assertThat([42 7 13], HasElementCount(3))

Interactive assertion passed.
```

Test the element count of the matrix.

```
testCase.assertThat([1 2 3; 4 5 6], HasElementCount(5))

Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
HasElementCount failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             6
    Expected Number of Elements:
             5

Actual Value:
         1     2     3
         4     5     6
Assertion failed.
```

The matrix has six elements.

Test that a square identity matrix has the correct number of elements.

```
n = 7;
testCase.assumeThat(eye(n), HasElementCount(n^2))
```

```
Interactive assumption passed.
```

Verify the element count of a cell array of strings.

```
testCase.verifyThat({'aString', 'anotherString'}, HasElementCount(2))
```

```
Interactive verification passed.
```

Test the element count of a structure.

```
s.Field1 = 1;
s.Field2 = 2;
testCase.verifyThat(s, HasElementCount(2))
```

```
Interactive verification failed.

--------------------
Framework Diagnostic:
--------------------
HasElementCount failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             1
    Expected Number of Elements:
             2

Actual Value:
        Field1: 1
        Field2: 2
```

The structure has two fields, but it only has one element.

```
testCase.verifyThat(s, HasElementCount(1))
```

```
Interactive verification passed.
```

See Also

HasLength | HasSize | IsEmpty | numel

matlab.unittest.constraints.HasField class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying structure containing particular field

Construction

`HasField(fieldname)` provides a constraint specifying structure containing particular field, `fieldname`. The constraint is satisfied if the actual value is a structure and that structure contains a field named `fieldname`.

Input Arguments

`fieldname`

Name of the field that a structure must contain to satisfy the constraint, specified as a string.

Properties

`Field`

Name of the field that a structure must contain to satisfy the constraint. Set this property through the constructor via the `fieldname` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Structure Has Particular Field

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasField

testCase = TestCase.forInteractiveUse;
```

Define the following structure, `S`, with two fields.

```
S = struct('Tag', 123, 'Serial', 345);
```

Verify that the structure has a `'Tag'` field.

```
testCase.verifyThat(S, HasField('Tag'))
```

```
Interactive verification passed.
```

Verify that the structure has a `'tag'` field.

```
testCase.verifyThat(S, HasField('tag'))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
HasField failed.
--> The value did not have the expected field.

    Actual Fieldnames:
            'Tag'
            'Serial'
    Expected Fieldname:
            'tag'

Actual Value:
           Tag: 123
        Serial: 345
```

The verification fails because the field name comparison is case sensitive.

Verify that the structure has a `'Tag'` field.

```
testCase.verifyThat(S, HasField('Tag'))
```

```
Interactive verification passed.
```

Verify that the structure has both a `'Tag'` and a `'Serial'` field.

```
testCase.verifyThat(S, HasField('Tag') & HasField('Serial'))
```

```
Interactive verification passed.
```

Verify that the structure does not have a `'Name'` field.

```
testCase.verifyThat(S, ~HasField('Name'))
```

```
Interactive verification passed.
```

matlab.unittest.constraints.HasInf class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying array containing any infinite value

Construction

`HasInf` creates a constraint that is able to determine if any value of an actual value array is an infinite value. This constraint is satisfied only if the actual value array contains at least one infinite value.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Array Contains Infinite Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasInf

testCase = TestCase.forInteractiveUse;
```

Test that the value `Inf` satisfies the constraint.

```
testCase.verifyThat(Inf, HasInf)
```

```
Interactive verification passed.
```

Assert that an array contains an infinite value.

```
testCase.assertThat([0 1 1 2 3 5 8 13], HasInf)
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
HasInf failed.
--> At least one element must be Inf or -Inf.

Actual Value:
         0     1     1     2     3     5     8    13
Assertion failed.
```

The array does not contain any infinite values.

Verify that an array contains an infinite value.

```
testCase.verifyThat([-Inf 5 NaN], HasInf)
```

```
Interactive verification passed.
```

Assert that a complex number that is infinite in the imaginary part satisfies the constraint.

```
testCase.assertThat(42+Inf*1i, HasInf)
```

```
Interactive verification passed.
```

Verify that an array does not contain any infinite values.

```
testCase.verifyThat([NaN -7+NaN*1i], ~HasInf)
```

```
Interactive verification passed.
```

Negating the `HasInf` constraint does not ensure the value is finite, only that it does not contain any infinite values.

See Also

`HasNaN` | `IsFinite` | `isinf`

matlab.unittest.constraints.HasLength class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying expected length of array

Construction

`HasLength(lengthVal)` provides a constraint that specifies an expected length of an array. The constraint is satisfied if the largest dimension length of the actual value array has the same number of elements specified as by `lengthVal`.

Input Arguments

`lengthVal`

Length a value must have to satisfy the constraint.

Properties

`Count`

Length a value must have to satisfy the constraint. Set this property through the constructor via the `lengthVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test for Expected Array Length

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasLength

testCase = TestCase.forInteractiveUse;
```

Assert that a 2x5x3 array has an expected length.

```
testCase.assertThat(rand(2, 5, 3), HasLength(5))
```

```
Interactive assertion passed.
```

Verify that a cell array of strings has an expected length.

```
testCase.verifyThat({'SomeString', 'SomeOtherString'}, HasLength(2))
```

```
Interactive verification passed.
```

Verify that an identity matrix has an expected length.

```
testCase.verifyThat(eye(2), HasLength(4))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
HasLength failed.
--> The array has an incorrect length.

    Actual Length:
             2
    Expected Length:
             4

Actual Array:
         1     0
         0     1
```

The matrix has a length of 2.

See Also

HasElementCount | HasSize | IsEmpty | length

matlab.unittest.constraints.HasNaN class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying array containing NaN value

Construction

`HasNaN` creates a constraint that is able to determine if any value of an actual value array is `NaN`. This constraint is satisfied only if the actual value array contains at least one `NaN` value.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Array Contains NaN Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasNaN

testCase = TestCase.forInteractiveUse;
```

Test that the value `NaN` satisfies the constraint.

```
testCase.verifyThat(NaN, HasNaN)
```

```
Interactive verification passed.
```

Assert that an array contains a NaN value.

```
testCase.assertThat([0 1 1 2 3 5 8 13], HasNaN)
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
HasNaN failed.
--> At least one element must be NaN.

Actual Value:
         0     1     1     2     3     5     8    13
Assertion failed.
```

The array does not contain a NaN value.

Verify that an array contains a NaN value.

```
testCase.verifyThat([-Inf 5 NaN], HasNaN)
```

```
Interactive verification passed.
```

Assert that a complex number satisfies the constraint.

```
testCase.assertThat(42+NaN*1i, HasNaN)
```

```
Interactive assertion passed.
```

Verify that an array does not contain any `NaN` values.

```
testCase.verifyThat([Inf -7+Inf*1i], ~HasNaN)
```

```
Interactive verification passed.
```

Negating the `HasNaN` constraint does not ensure the value is finite, only that it does not contain any `NaN` values.

See Also

`HasInf` | `IsFinite` | `isnan`

matlab.unittest.constraints.HasSize class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying expected size of array

Construction

`HasSize(sizeVal)` provides a constraint that specifies an expected size of an array. The constraint is satisfied if the actual value array size is equal to the size specified by `sizeVal`.

Input Arguments

`sizeVal`

Size a value must have to satisfy the constraint.

Properties

`Count`

Size a value must have to satisfy the constraint. Set this property through the constructor via the `sizeVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test for Expected Array Size

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasSize

testCase = TestCase.forInteractiveUse;
```

Assert that a `2x5x3` array has an expected size.

```
testCase.assertThat(rand(2, 5, 3), HasSize([2 5 3]))
```

```
Interactive assertion passed.
```

Verify that a cell array of strings has an expected size.

```
testCase.verifyThat({'SomeString', 'SomeOtherString'}, HasSize([1 2]))
```

```
Interactive verification passed.
```

Verify that an identity matrix has an expected size.

```
testCase.verifyThat(eye(2), HasSize([4 1]))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
HasSize failed.
--> The value had an incorrect size.

    Actual Size:
             2     2
    Expected Size:
             4     1

Actual Value:
         1     0
         0     1
```

The matrix has a size of `2x2`.

See Also

`HasElementCount` | `HasLength` | `IsEmpty` | `size`

matlab.unittest.constraints.IsAnything class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying any value

Construction

`IsAnything` provides a constraint specifying any value. The constraint is satisfied by any value. It is the default constraint for selectors that do not require an input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Instantiate `IsAnything` Object

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsAnything

testCase = TestCase.forInteractiveUse;
```

Verify that the following values satisfy the `IsAnything` constraint: `NaN`, an `inputParser` object, a numeric array, and a complex number.

```
testCase.verifyThat(NaN, IsAnything)
testCase.verifyThat(inputParser, IsAnything)
testCase.verifyThat(1:10, IsAnything)
testCase.verifyThat(-Inf+5j, IsAnything)
```

```
Interactive verification passed.
Interactive verification passed.
```

```
Interactive verification passed.
Interactive verification passed.
```

Test that empty cells, arrays, and strings satisfy the `IsAnything` constraint.

```
testCase.verifyThat({}, IsAnything)
testCase.verifyThat([], IsAnything)
testCase.verifyThat('', IsAnything)
```

```
Interactive verification passed.
Interactive verification passed.
Interactive verification passed.
```

The constraint is satisfied even though the data are empty.

See Also

`matlab.unittest.selectors`

matlab.unittest.constraints.IsEmpty class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying empty value

Construction

`IsEmpty` provides a constraint that specifies an empty value. The constraint is satisfied if the actual value array is empty.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Empty

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEmpty

testCase = TestCase.forInteractiveUse;
```

Verify that an empty string satisfies the `IsEmpty` constraint.

```
testCase.verifyThat('', IsEmpty)
```

```
Interactive verification passed.
```

Assert that a vector is not empty.

```
testCase.assertThat([13 42], ~IsEmpty)
```

```
Interactive verification passed.
```

Verify that a matrix with a dimension of length zero is empty.

```
testCase.verifyThat(rand(2, 5, 0, 3), IsEmpty)
```

```
Interactive verification passed.
```

Assert that an empty object satisfies the `IsEmpty` constraint.

```
testCase.assertThat(MException.empty, IsEmpty)
```

```
Interactive assertion passed.
```

Verify that a cell array containing an empty numeric array is empty.

```
testCase.verifyThat({[]}, IsEmpty)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEmpty failed.
--> The value must be empty.
--> The value has a size of [1  1].

Actual Value:
        {[]}
```

The cell array is not empty, even though the only thing it contains is an empty array.

See Also

`HasCount` | `HasLength` | `HasSize` | `isempty`

matlab.unittest.constraints.IsFalse class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying false value

Construction

`IsFalse` provides a constraint specifying a false value. This constraint is satisfied only by a scalar logical with a value of `false`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test Actual Value Is False

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsFalse

testCase = TestCase.forInteractiveUse;
```

Test that `false` satisfies the `IsFalse` constraint.

```
testCase.verifyThat(false, IsFalse)
```

```
Interactive verification passed.
```

Test that the `IsFalse` constraint is not satisfied by `true`.

```
testCase.verifyThat(true, IsFalse)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsFalse failed.
--> The value must evaluate to "false".

Actual Value:
         1
```

The test fails because `true` returns `logical(1)`.

Test that the `IsFalse` constraint is not satisfied by the double `0`.

```
testCase.verifyThat(0, IsFalse)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsFalse failed.
--> The value must be logical. It is of type "double".

Actual Value:
         0
```

The `IsFalse` constraint is satisfied only by `logical(0)`.

Test that the `IsFalse` constraint is not satisfied by a logical array of zeros.

```
testCase.verifyThat([false false false], IsFalse)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     0     0
```

The `IsFalse` constraint is only satisfied if the value is scalar and `logical(0)`.

See Also

`IsTrue`

matlab.unittest.constraints.IsFinite class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying finite value

Construction

`IsFinite` creates a constraint that is able to determine if all values of an actual value array are finite. This constraint is satisfied only if the actual value array does not contain any infinite or `NaN` values.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Array Contains Only Finite Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsFinite

testCase = TestCase.forInteractiveUse;
```

Test that the value `17` satisfies the constraint.

```
testCase.verifyThat(17, IsFinite)
```

```
Interactive verification passed.
```

Assert that an array is completely finite.

```
testCase.assertThat([0 1 1 2 3 5 8 13], IsFinite)
```

```
Interactive assertion passed.
```

Verify that an array is completely finite.

```
testCase.verifyThat([-Inf 5 NaN], IsFinite)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsFinite failed.
--> All elements must be finite-valued.
    Failing indices:
              1     3

Actual Value:
      -Inf     5   NaN
```

The array contains an infinite value.

Test if a complex number that is infinite in the imaginary part satisfies the constraint.

```
testCase.assertThat(42+Inf*1i, IsFinite)
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsFinite failed.
--> The value must be finite.

Actual Value:
     42.000000000000000 +               Infi
Assertion failed.
```

Verify that an array does not contain all finite values.

```
testCase.verifyThat([NaN -7+NaN*1i], ~IsFinite)
```

```
Interactive verification passed.
```

See Also

`HasInf` | `HasNaN` | `isfinite`

matlab.unittest.constraints.IsGreaterThan class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying value greater than another value

Construction

`IsGreaterThan(floorVal)` creates a constraint specifying that an actual value is greater than another value. The constraint is satisfied if the actual value array is greater than the specified floor value, `floorVal`. The actual value is greater than `floorVal` only if the result of the expression `actual > floorVal` is nonempty and all values are true.

Input Arguments

`floorVal`

Largest value that fails the constraint.

Properties

`FloorValue`

Largest value that fails the constraint. Set this property through the constructor via the `floorVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Greater Than Provided Floor Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsGreaterThan

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is greater than two.

```
actVal = 3;
testCase.verifyThat(actVal, IsGreaterThan(2))
```

```
Interactive verification passed.
```

Test that the actual value is greater than three.

```
testCase.verifyThat(actVal, IsGreaterThan(3))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsGreaterThan failed.
--> The value must be greater than the minimum value.

Actual Value:
         3
Minimum Value (Exclusive):
         3
```

The actual value is equal to, not greater than, three.

Test that each element in the actual value array is greater than four.

```
actVal = [5 6 7];
testCase.verifyThat(actVal, IsGreaterThan(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is greater than four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsGreaterThan(4))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsGreaterThan failed.
--> Each element must be greater than the minimum value.

    Failing Indices:
              1     2     3     5

Actual Value:
         1     2     3
         4     5     6
Minimum Value (Exclusive):
         4
```

The matrix contains four elements with a value less than or equal to four.

Test that the actual value, 5, is greater than every element in an array.

```
testCase.verifyThat(5, IsGreaterThan([1 2 3]))

Interactive verification passed.
```

Test that elements in the actual value array are greater than the corresponding floor values.

```
testCase.verifyThat([5 -3 2], IsGreaterThan([4 -9 0]))

Interactive verification passed.
```

Repeat the test, this time negating the first actual value element.

```
testCase.verifyThat([-5 -3 2], IsGreaterThan([4 -9 0]))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsGreaterThan failed.
--> Each element must be greater than each corresponding element of the minimum value a
```

```
    Failing Indices:
              1

Actual Value:
        -5    -3     2
Minimum Value (Exclusive):
         4    -9     0
```

The negated element is less than four.

See Also

`gt` | `IsGreaterThanOrEqualTo` | `IsLessThan` | `IsLessThanOrEqualTo`

matlab.unittest.constraints.IsGreaterThanOrEqualTo class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying value greater than or equal to another value

Construction

`IsGreaterThanOrEqualTo(floorVal)` creates a constraint specifying that an actual value is greater than or equal to another value. The constraint is satisfied if the actual value array is greater than or equal to the specified floor value, `floorVal`. The actual value is greater than or equal to `floorVal` only if the result of the expression `actual >= floorVal` is nonempty and all values are true.

Input Arguments

`floorVal`

Minimum value to satisfy the constraint.

Properties

`FloorValue`

Minimum value to satisfy the constraint. Set this property through the constructor via the `floorVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Greater Than or Equal to Provided Floor Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsGreaterThanOrEqualTo

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is greater than or equal to two.

```
actVal = 3;
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(2))
```

Test that the actual value is greater than or equal to three.

```
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(3))
```

```
Interactive verification passed.
```

Test that each element in the actual value array is greater than or equal to four.

```
actVal = [5 6 7];
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is greater than or equal to four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(4))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsGreaterThanOrEqualTo failed.
--> Each element must be greater than or equal to the minimum value.

    Failing Indices:
              1     3     5
```

```
Actual Value:
         1     2     3
         4     5     6
Minimum Value (Inclusive):
         4
```

The matrix contains three elements that are greater than or equal to four.

Test that the actual value, 5, is greater than or equal to every element in an array.

```
testCase.verifyThat(5, IsGreaterThanOrEqualTo([1 2 3 5]))
```

```
Interactive verification passed.
```

Test that elements in the actual value array are greater than or equal to the corresponding floor values.

```
testCase.verifyThat([5 -3 0], IsGreaterThanOrEqualTo([4 -9 0]))
```

```
Interactive verification passed.
```

Repeat the test, this time negating the first actual value element.

```
testCase.verifyThat([-5 -3 0], IsGreaterThanOrEqualTo([4 -9 0]))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsGreaterThanOrEqualTo failed.
--> Each element must be greater than or equal to each corresponding element of the mi

    Failing Indices:
              1

Actual Value:
        -5    -3     0
Minimum Value (Inclusive):
         4    -9     0
```

The negated element is less than or equal to four.

See Also

`ge` | `IsGreaterThan` | `IsLessThan` | `IsLessThanOrEqualTo`

matlab.unittest.constraints.IsEqualTo class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

General constraint to compare for equality

Description

The `IsEqualTo` class creates a constraint that compares data for equality. The type of comparison it uses is governed by the data type of the expected value. First, the testing framework checks if the expected value is an object. This check is performed first because it is possible for the object to have overridden methods that are used in subsequent checks (e.g. `islogical`). The following list categorizes and describes the various tests.

Data Type	Equality Comparison Method
MATLAB & Java Objects	If the expected value is a MATLAB or Java object, the `IsEqualTo` constraint calls the `isequaln` method if it is defined on the expected value object, otherwise it calls `isequal`. If the check returns false and a supported tolerance is specified, the `IsEqualTo` constraint checks the actual and expected values for equivalent class, size, and sparsity before determining if the values are within the tolerance.
Logicals	If the expected value is a `logical`, the constraint checks the actual and expected values for equivalent sparsity. If the sparsity matches, the constraint compares the values with the `isequal` method. Otherwise, the constraint is not satisfied.
Numerics	If the expected value is `numeric`, the constraint checks the actual and expected values for equivalent class, size, and sparsity. If the all these checks match, the constraint uses the `isequaln` method for comparison. If `isequaln` returns `true`, the constraint is satisfied. If the complexity does not match or `isequaln` returns `false`, and a supported tolerance is supplied, the constraint uses the tolerance in the comparison. Otherwise, the constraint is not satisfied.
Strings	If the expected value is a `string`, the constraint uses the `strcmp` function to check the actual and expected values for equality. However, if the `IgnoreCase` property is true, the strings are compared using `strcmpi`. If the `IgnoreWhitespace` is `true`, all whitespace characters

Data Type	Equality Comparison Method
	are removed from the actual and expected strings before passing them to `strcmp` or `strcmpi`.
Structures	If the expected value is a `struct`, the constraint compares the field count of the actual and expected values. If not equal, the constraint is not satisfied. Otherwise, each field of the expected value `struct` must exist on the actual value `struct`. If any field names are different, the constraint is not satisfied. Then, the constraint recursively compares the fields in a depth first examination. The recursion continues until a fundamental data type is encountered (i.e. logical, numeric, string, or object), and then the values are compared as described above.
Cell Arrays	If the expected value is a cell array, the constraint checks the actual and expected values for size equality. If they are not equal in size, the constraint is not satisfied. Otherwise, each element of the array is recursively compared in a manner identical to fields in a structure, described above.

Construction

`IsEqualTo(expVal)` provides a general constraint to compare for equality.

`IsEqualTo(expVal,Name,Value)` provides a constraint with additional options specified by one or more Name,Value pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`expVal`

The expected value that will be compared to the actual value.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoringCase'`

Indicator if the constraint is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

`'IgnoringWhitespace'`

Indicator if the constraint is insensitive to whitespace, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

`'Using'`

Particular comparator to use for constraint construction, specified as a `matlab.unittest.constraints.Comparator` object

Default: (empty)

`'Within'`

Tolerance to use in constraint construction, specified as a `matlab.unittest.constraints.Tolerance` object

Default: (empty)

Properties

`Comparator`

Specific comparator used in construction of the constraint, specified as a `matlab.unittest.constraints.Comparator` object in the name-value pair argument, `'Using'`.

`Expected`

The expected value that will be compared to the actual value specified in the `expVal` input argument.

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`. This property applies at all levels of recursion, such as nested structures.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`. This property applies at all levels of recursion, such as nested structures.

`Tolerance`

Specific tolerance used in construction of the constraint, specified as a `matlab.unittest.constraints.Tolerance` object in the name-value pair argument, `'Within'`. This property applies at all levels of recursion, such as nested structures.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test Numerics for Equality

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Verify that an actual value of 5 is equal to the expected value.

```
expVal = 5;
testCase.verifyThat(expVal, IsEqualTo(5))
```

```
Interactive verification passed.
```

Assume that the actual value is 4.95. Verify that the difference between the actual value and expected value is less than 0.09.

```
testCase.verifyThat(expVal, IsEqualTo(4.95, 'Within', AbsoluteTolerance(0.09)))
```

```
Interactive verification passed.
```

Assume that the actual value is 4.9. Verify that the difference between the actual and expected value is less than 1%.

```
testCase.verifyThat(expVal, IsEqualTo(4.9, 'Within', RelativeTolerance(0.01)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> RelativeTolerance failed.
        --> The error was not within relative tolerance.
        --> Failure table:
                    Index    Actual    Expected          Error              RelativeErro
                    _____    ______    ________    __________________    ________________

                    1        5         4.9         0.0999999999999996    0.020408163265

Actual Value:
         5
Expected Value:
       4.900000000000000
```

The two values differ by more than 1%.

Test Floating Point Calculation with Tolerance

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance
```

```
testCase = TestCase.forInteractiveUse;
```

Test that `0.1*3 = 0.3`.

```
act = 0.1*3;
exp = 0.3;
testCase.verifyThat(act, IsEqualTo(exp))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected           Error                RelativeError
                _____    ______    ________    ____________________    _________________

                1        0.3       0.3         5.55111512312578e-17    1.85037170770859

Actual Value:
       0.300000000000000
Expected Value:
       0.300000000000000
```

This test fails due to round off error in floating point arithmetic.

Perform the comparison of floating point numbers using a tolerance. Test that `0.1*3 = 0.3` within a relative tolerance of `2*eps`.

```
testCase.verifyThat(act, IsEqualTo(exp, ...
    'Within', RelativeTolerance(2*eps)))
```

```
Interactive verification passed.
```

Test Strings for Equality

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Verify that two strings are equal.

```
expVal = 'Hello';
testCase.verifyThat(expVal, IsEqualTo('Hello'))
```

```
Interactive verification passed.
```

Change the case of the actual string and test for equality.

```
testCase.verifyThat(expVal, IsEqualTo('hello'))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> StringComparator failed.
    --> The strings are not equal

Actual String:
    Hello
Expected String:
    hello
```

Ignore case and test again.

```
testCase.verifyThat(expVal, IsEqualTo('hello', 'IgnoringCase', true))
```

```
Interactive verification passed.
```

Ignore whitespace and test two strings.

```
expVal = 'a bc';
testCase.verifyThat(expVal, IsEqualTo('abc', 'IgnoringWhitespace', true))
testCase.verifyThat(expVal, IsEqualTo('ab c', 'IgnoringWhitespace', true))
```

```
Interactive verification passed.
Interactive verification passed.
```

Test Objects for Equality Using Comparator

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
```

```
import matlab.unittest.constraints.RelativeTolerance
import matlab.unittest.constraints.PublicPropertyComparator

testCase = TestCase.forInteractiveUse;
```

Define actual and expected `timeseries` objects. Perturb one of the actual data points by 1%.

```
expected = timeseries(1:10);
actual = expected;
actual.Data(7) = 1.01*actual.Data(7);
```

Test that the actual and expected values are equal within a relative tolerance of 2%.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02)))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".
    --> The tolerance does not support timeseries values so it was not used.

Actual Object:
      timeseries

      Common Properties:
                Name: 'unnamed'
                Time: [10x1 double]
            TimeInfo: [1x1 tsdata.timemetadata]
                Data: [1x1x10 double]
            DataInfo: [1x1 tsdata.datametadata]

      More properties, Methods
Expected Object:
      timeseries

      Common Properties:
                Name: 'unnamed'
                Time: [10x1 double]
            TimeInfo: [1x1 tsdata.timemetadata]
```

```
                Data: [1x1x10 double]
            DataInfo: [1x1 tsdata.datametadata]

      More properties, Methods
```

Use the `PublicPropertyComparator` in the construction of the constraint.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02),...
    'Using', PublicPropertyComparator.supportingAllValues))
```

```
Interactive verification passed.
```

The test passes because the `PublicPropertyComparator` compares each public property individually instead of comparing the object all at once. In the former test, the `ObjectComparator` compares `timeseries` objects, and therefore relies on the `isequal` method of the `timeseries` class. Due to the perturbation in the actual `timeseries`, `isequal` returns `false`. The comparator does not apply the tolerance because the double-valued tolerance cannot apply directly to the `timeseries` object. In the latter test, the comparator applies the tolerance to each public property that contains double-valued data.

See Also

`matlab.unittest.constraints.Constraint` |
`matlab.unittest.constraints.Tolerance`

Introduced in R2013a

matlab.unittest.constraints.IsInstanceOf class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying inclusion in given class hierarchy

Construction

`IsInstanceOf(class)` provides a constraint specifying inclusion in a given class hierarchy. The constraint is satisfied if the actual value instance passes the "isa" relationship with `class` .

Input Arguments

`class`

Class name that the actual value must derive from or be an instance of to satisfy the constraint, specified as a fully qualified class name string or a `meta.class` instance.

Properties

`Class`

Class name that the actual value must derive from or be an instance of to satisfy the constraint. Set this property through the constructor via the `class` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Instance of Specified Class

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsInstanceOf

testCase = TestCase.forInteractiveUse;
```

Verify that the actual value, 5, is an instance of the double class.

```
testCase.verifyThat(5, IsInstanceOf('double'))
```

```
Interactive verification passed.
```

Repeat the test using an instance of `meta.class` instead of a string.

```
testCase.verifyThat(5, IsInstanceOf(?double))
```

```
Interactive verification passed.
```

Assert that zero is an instance of the logical class.

```
testCase.assertThat(0, IsInstanceOf('logical'))
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        logical

Actual Value:
         0
Assertion failed.
```

Verify that `@sin` is a function handle.

```
testCase.verifyThat(@sin, IsInstanceOf(?function_handle))
```

```
Interactive verification passed.
```

Verify that `name` is an instance of the `char` class.

```
name = 42;
```

```
testCase.verifyThat(name, IsInstanceOf('char'))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        char

Actual Value:
        42
```

Test That Derived Class Is Instance of Specified Class

In a file in your working folder, create a class, `DerivedExample`, that inherits from the `handle` class.

```
classdef DerivedExample < handle
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsInstanceOf

testCase = TestCase.forInteractiveUse;
```

Verify that an instance of the `DerivedExample` class is an instance of a `handle`.

```
exObj = DerivedExample;
testCase.verifyThat(exObj, IsInstanceOf(?handle))

Interactive verification passed.
```

Even though `exObj` is not an instance of the `handle` class, the verification passes because it derives from the `handle` class.

See Also

`isa` | `IsOfClass`

matlab.unittest.constraints.IsLessThan class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying value less than another value

Construction

`IsLessThan(ceilVal)` creates a constraint specifying that an actual value is less than another value. The constraint is satisfied if the actual value array is less than the specified ceiling value, `ceilVal`. The actual value is less than `ceilVal` only if the result of the expression `actual < ceilVal` is nonempty and all values are true.

Input Arguments

`ceilVal`

Smallest value that fails the constraint.

Properties

`CeilingValue`

Smallest value that fails the constraint. Set this property through the constructor via the `ceilVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Less Than Provided Ceiling Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsLessThan

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is less than four.

```
actVal = 3;
testCase.verifyThat(actVal, IsLessThan(4))
```

```
Interactive verification passed.
```

Test that the actual value is less than three.

```
testCase.verifyThat(actVal, IsLessThan(3))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsLessThan failed.
--> The value must be less than the maximum value.

Actual Value:
         3
Maximum Value (Exclusive):
         3
```

The actual value is equal to, not less than, three.

Test that each element in the actual value array is less than four.

```
actVal = [1 2 3];
testCase.verifyThat(actVal, IsLessThan(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is less than four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsLessThan(4))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsLessThan failed.
--> Each element must be less than the maximum value.

    Failing Indices:
              2     4     6

Actual Value:
         1     2     3
         4     5     6
Maximum Value (Exclusive):
         4
```

The matrix contains three elements that are greater than or equal to four.

Test that the actual value, 0, is less than every element in an array.

```
testCase.verifyThat(0, IsLessThan([1 2 3]))

Interactive verification passed.
```

Test that elements in the actual value array are less than the corresponding ceiling values.

```
testCase.verifyThat([4 -9 0], IsLessThan([5 -3 2]))

Interactive verification passed.
```

Repeat the test, this time negating the second actual value element.

```
testCase.verifyThat([4 9 0], IsLessThan([5 -3 2]))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsLessThan failed.
--> Each element must be less than each corresponding element of the maximum value arra
```

```
    Failing Indices:
              2

Actual Value:
         4     9     0
Maximum Value (Exclusive):
         5    -3     2
```

The negated element is greater than `-3`.

See Also

`IsGreaterThan` | `IsGreaterThanOrEqualTo` | `IsLessThanOrEqualTo` | `lt`

matlab.unittest.constraints.IsLessThanOrEqualTo class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying value less than or equal to another value

Construction

`IsLessThanOrEqualTo(ceilVal)` creates a constraint specifying that an actual value is less than or equal to another value. The constraint is satisfied if the actual value array is less than or equal to the specified ceiling value, `ceilVal`. The actual value is less than or equal to `ceilVal` only if the result of the expression `actual <= ceilVal` is nonempty and all values are true.

Input Arguments

`ceilVal`

Maximum value to satisfy the constraint.

Properties

`CeilingValue`

Maximum value to satisfy the constraint. Set this property through the constructor via the `ceilVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Less Than or Equal to Provided Ceiling Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsLessThanOrEqualTo

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is less than or equal to four.

```
actVal = 3;
testCase.verifyThat(actVal, IsLessThanOrEqualTo(4))
```

```
Interactive verification passed.
```

Test that the actual value is less than or equal to three.

```
testCase.verifyThat(actVal, IsLessThanOrEqualTo(3))
```

```
Interactive verification passed.
```

Test that each element in the actual value array is less than or equal to four.

```
actVal = [1 2 3 4];
testCase.verifyThat(actVal, IsLessThanOrEqualTo(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is less than or equal to four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsLessThanOrEqualTo(4))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsLessThanOrEqualTo failed.
--> Each element must be less than or equal to the maximum value.

    Failing Indices:
              4     6
```

```
Actual Value:
         1     2     3
         4     5     6
Maximum Value (Inclusive):
         4
```

The matrix contains two elements that are greater than four.

Test that the actual value, 1, is less than or equal to every element in an array.

```
testCase.verifyThat(1, IsLessThanOrEqualTo([1 2 3]))
```

```
Interactive verification passed.
```

Test that elements in the actual value array are less than the corresponding ceiling values.

```
testCase.verifyThat([4 -9 2], IsLessThanOrEqualTo([5 -3 2]))
```

```
Interactive verification passed.
```

Repeat the test, this time negating the second actual value element.

```
testCase.verifyThat([4 9 2], IsLessThanOrEqualTo([5 -3 2]))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsLessThanOrEqualTo failed.
--> Each element must be less than or equal to each corresponding element of the maximu

    Failing Indices:
              2

Actual Value:
         4     9     2
Maximum Value (Inclusive):
         5    -3     2
```

The negated element is greater than `-3`.

See Also

`IsGreaterThan` | `IsGreaterThanOrEqualTo` | `IsLessThan` | `le`

matlab.unittest.constraints.IsOfClass class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying class type

Construction

`IsOfClass(class)` provides a constraint specifying the class type. The constraint is satisfied if the actual value is the same class as `class` . The constraint is not satisfied if the actual value derives from `class`.

Input Arguments

`class`

Class name that must be matched by the actual value to satisfy the constraint, specified as a fully qualified class name string or a `meta.class` instance.

Properties

`Class`

Class name that must be matched by the actual value to satisfy the constraint. Set this property through the constructor via the `class` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Class Is Specified Class

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsOfClass

testCase = TestCase.forInteractiveUse;
```

Verify that the actual value, 5, is a double.

```
testCase.verifyThat(5, IsOfClass('double'))

Interactive verification passed.
```

Repeat the test using an instance of `meta.class` instead of a string.

```
testCase.verifyThat(5, IsOfClass(?double))

Interactive verification passed.
```

Assert that zero is an instance of the logical class.

```
testCase.assertThat(0, IsOfClass('logical'))

Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsOfClass failed.
--> The value's class is incorrect.

    Actual Class:
        double
    Expected Class:
        logical

Actual Value:
         0
Assertion failed.
```

Verify that `@sin` is a function handle.

```
testCase.verifyThat(@sin, IsOfClass(?function_handle))

Interactive verification passed.
```

Verify that `name` is an instance of the `char` class.

```
name = 42;
testCase.verifyThat(name, IsOfClass('char'))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsOfClass failed.
--> The value's class is incorrect.

    Actual Class:
        double
    Expected Class:
        char

Actual Value:
        42
```

Test That Derived Class Is Instance of Specified Class

In a file in your working folder, create a class, `DerivedExample`, that inherits from the `handle` class.

```
classdef DerivedExample < handle
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsOfClass

testCase = TestCase.forInteractiveUse
```

Verify that an instance of the `DerivedExample` class is an instance of a `handle`.

```
exObj = DerivedExample;
testCase.verifyThat(exObj, IsOfClass(?handle))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsOfClass failed.
```

```
--> The value's class is incorrect.

    Actual Class:
        DerivedExample
    Expected Class:
        handle

Actual Value:
      DerivedExample with no properties.
```

Even though `exObj` derives from the `handle` class, it is not an instance of the `handle` class.

Verify that an instance of the `DerivedExample` class is an instance of a `DerivedExample`.

```
testCase.verifyThat(exObj, IsOfClass(?DerivedExample))
```

```
Interactive verification passed.
```

See Also

`class` | `IsInstanceOf`

matlab.unittest.constraints.IsReal class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying real valued array

Construction

`IsReal` provides a constraint specifying a real valued array. This constraint is satisfied only if the actual value contains only real values.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Array Is Real Valued

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsReal

testCase = TestCase.forInteractiveUse;
```

Verify that the values `5` and `5+0i` are real.

```
testCase.verifyThat(5, IsReal)
testCase.verifyThat(5+0i, IsReal)
```

```
Interactive verification passed.
Interactive verification passed.
```

Test if the imaginary number is real.

```
testCase.verifyThat(sqrt(-1), IsReal)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsReal failed.
--> The value must be real.

Actual Value:
      0.000000000000000 + 1.000000000000000i
```

The actual value is imaginary.

Assert that an array contains only real values.

```
testCase.assertThat([0 1 1 2 3 5 8 13], IsReal)
```

```
Interactive assertion passed.
```

Test that the array, `arr`, is real.

```
arr = [NaN -Inf];
testCase.verifyThat(arr, IsReal)
```

```
Interactive verification passed.
```

Multiply the array by a complex number and test that the values are not real.

```
testCase.verifyThat(42i*arr, ~IsReal)
```

```
Interactive verification passed.
```

See Also

`isreal`

matlab.unittest.constraints.IsSameHandleAs class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying handle instance same as another

Construction

`IsSameHandle(h)` provides a constraint specifying a handle instance or group of instances is same as another.

The constraint is satisfied only if each element of the actual value is the same instance as each corresponding element of `h`.

Input Arguments

`h`

`handle` object or array of `handle` objects. The actual value array passed to the qualification must be the same size as `h`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test Handles for Equality

In a file in your working folder, create the following handle class for interactive testing.

```
classdef ExampleHandle < handle
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsSameHandleAs

testCase = TestCase.forInteractiveUse;
```

Instantiate two handles.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Verify that the handle, `h1`, is the same as `h1`.

```
testCase.verifyThat(h1, IsSameHandleAs(h1))
```

```
Interactive verification passed.
```

Test that `h1` is the same handle instance as `h2`.

```
testCase.verifyThat(h1, IsSameHandleAs(h2))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsSameHandleAs failed.
--> Values do not refer to the same handle.

Actual Value:
      ExampleHandle with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test that two arrays of handles are the same instances.

```
expArr = [h1 h2 h1];
actArr = [h1 h2 h1];

testCase.verifyThat(expArr, IsSameHandleAs(actArr))
```

```
Interactive verification passed.
```

The arrays satisfy the constraint even though the elements within a particular array are not the same instance as each other.

Verify that the constraint is not satisfied if it expects a single handle and the actual value is an array of the same instances.

```
testCase.verifyThat([h1 h1], IsSameHandleAs(h1))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsSameHandleAs failed.
--> Sizes do not match.
     Actual Value Size           : [1  2]
     Expected Handle Object Size : [1  1]

Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Similarly, the constraint is not satisfied a single handle instance if it expects an array of handles.

```
testCase.verifyThat(h2, IsSameHandleAs([h2 h2]))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsSameHandleAs failed.
--> Sizes do not match.
     Actual Value Size           : [1  1]
     Expected Handle Object Size : [1  2]

Actual Value:
      ExampleHandle with no properties.
Expected Handle Object:
      1x2 ExampleHandle array with no properties.
```

See Also

`handle` | `eq` | `IsEqualTo`

matlab.unittest.constraints.IsScalar class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying scalar value

Construction

`IsScalar` provides a constraint that specifies a scalar value. The constraint is satisfied if the actual value is a scalar.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Scalar

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsScalar

testCase = TestCase.forInteractiveUse;
```

Verify that a value of zero satisfies the `IsScalar` constraint.

```
testCase.verifyThat(0,IsScalar)
```

```
Interactive verification passed.
```

Assert that a single object is scalar.

```
testCase.assertThat(timeseries(1),IsScalar)
```

```
Interactive verification passed.
```

Verify that a vector is not scalar.

```
testCase.verifyThat([2 3],IsScalar)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsScalar failed.
--> The value must be a scalar.
--> The value has a size of [1  2].

Actual Value:
         2     3
```

Assert that an empty structure does not satisfy the `IsScalar` constraint.

```
testCase.assertThat(struct([]),IsScalar)
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsScalar failed.
--> The value must be a scalar.
--> The value has a size of [0  0].

Actual Value:
    0x0 struct array with no fields.
Assertion failed.
```

See Also

`HasElementCount` | `HasLength` | `HasSize` | `IsEmpty` | `isscalar`

Introduced in R2014b

matlab.unittest.constraints.IsSparse class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying sparse array

Construction

`IsSparse` creates a constraint specifying a sparse array. This constraint is satisfied only when the actual value is sparse.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Array Is Sparse

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsSparse

testCase = TestCase.forInteractiveUse;
```

Create an identity matrix, and test if it is sparse.

```
F = eye(7);
testCase.verifyThat(F, IsSparse)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
```

```
IsSparse failed.
--> The value must be sparse.

Actual Value:
         1     0     0     0     0     0     0
         0     1     0     0     0     0     0
         0     0     1     0     0     0     0
         0     0     0     1     0     0     0
         0     0     0     0     1     0     0
         0     0     0     0     0     1     0
         0     0     0     0     0     0     1
```

The matrix, `F`, is a full matrix.

Convert `F` to a sparse matrix and retest for sparsity.

```
S = sparse(F);
testCase.verifyThat(S, IsSparse)
```

```
Interactive verification passed.
```

See Also

`issparse`

matlab.unittest.constraints.IsSubstringOf class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying substring of another string

Construction

`IsSubstringOf(superstring)` creates a constraint specifying a substring of another string. The constraint is satisfied only if the actual value is contained within an expected string, `superstring`.

`IsSubstringOf(superstring,Name,Value)` provides a constraint with additional options specified by one or more Name,Value pair arguments. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`superstring`

Text that contains the actual value, specified as a string. `superstring` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoringCase'`

Indicator if the constraint is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

`'IgnoringWhitespace'`

Indicator if the constraint is insensitive to whitespace, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

Properties

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`. This property applies at all levels of recursion, such as nested structures.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`. This property applies at all levels of recursion, such as nested structures.

`Superstring`

String that includes the actual value, specified in the input argument, `superstring`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Is Substring of Specified String

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsSubstringOf
```

```
testCase = TestCase.forInteractiveUse;
```

Define the actual value string.

```
S = 'This Is One Long String';
```

Test that the actual value string, `'One'`, is contained in `S`.

```
testCase.verifyThat('One', IsSubstringOf(S))
```

```
Interactive verification passed.
```

Test that the actual value string, `'long'`, is contained in `S`.

```
testCase.verifyThat('long', IsSubstringOf(S))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsSubstringOf failed.
--> The string must be found within the superstring.

Actual String:
    long
Expected Superstring:
    This Is One Long String
```

By default, the `IsSubstringOf` constraint is case sensitive.

Repeat the test ignoring case.

```
testCase.verifyThat('long', IsSubstringOf(S,...
    'IgnoringCase', true))
```

```
Interactive verification passed.
```

Test that the actual value string, `'thisisone'`, is contained in `S`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat('thisisone', IsSubstringOf(S, ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Interactive verification passed.
```

Assert that the actual value string, `'longer'`, is not contained in S.

```
testCase.assertThat('Longer', ~IsSubstringOf(S))
```

```
Interactive assertion passed.
```

See Also

`ContainsSubstring` | `EndsWithSubstring` | `Matches` | `StartsWithSubstring`

matlab.unittest.constraints.IssuesNoWarnings class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying function that issues no warnings

Construction

`outConstObj = IssuesNoWarnings` creates a constraint, `outConstObj`, specifying a function that issues no warnings when the testing framework invokes it. The constraint is satisfied if no warnings are issued when the testing framework invokes the function.

`outConstObj = IssuesNoWarnings('WhenNargoutIs', numOutputs)` creates a constraint that can determine if the actual value is a function handle that issues no warnings when the testing framework invokes it with a particular number of output arguments, `numOutputs`.

Input Arguments

`numOutputs`

Number of outputs the constraint requests when invoking the function handle, specified as a non-negative, real, scalar integer.

Default: `0`

Properties

`FunctionOutputs`

Output arguments produced at invocation of the supplied function handle, specified as a cell array. This property provides access to output arguments. It is read only and the testing framework sets it when it invokes the function handle. The number of outputs is determined by the `Nargout` property.

`Nargout`

Number of output arguments the instance uses when executing functions. Set this property through the constructor via the `numOutputs` input argument.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Instantiate `IssuesNoWarnings` Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IssuesNoWarnings

testCase = TestCase.forInteractiveUse;
```

Verify that a call to `true` does not result in any warning.

```
testCase.verifyThat(@true, IssuesNoWarnings)
```

```
Interactive verification passed.
```

Verify that a call to `size` with an empty array does not result in any warning. Examine the output arguments.

```
issuesNoWarningsConstraint = IssuesNoWarnings('WhenNargoutIs', 2);
testCase.verifyThat(@() size([]), issuesNoWarningsConstraint)
[actualOut1, actualOut2] = issuesNoWarningsConstraint.FunctionOutputs{:};
```

```
Interactive verification passed.
```

Verify that the constraint is not satisfied if the actual value is not a function handle.

```
testCase.verifyThat(5, IssuesNoWarnings)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IssuesNoWarnings failed.
--> The value must be an instance of the expected type.
```

```
    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
```

Verify that the constraint is not satisfied if the actual value results in a warning.

```
testCase.verifyThat(@() warning('some:id', 'Message'), IssuesNoWarnings)
```

```
Warning: Message
> In @()warning('some:id','Message')
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 39
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 54
  In IssuesNoWarnings>IssuesNoWarnings.issuesNoWarnings at 132
  In IssuesNoWarnings>IssuesNoWarnings.satisfiedBy at 83
  In QualificationDelegate>QualificationDelegate.qualifyThat at 58
  In Verifiable>Verifiable.verifyThat at 228
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IssuesNoWarnings failed.
--> The function issued warnings.

    Warnings Issued:
         some:id

Evaluated Function:
        @()warning('some:id','Message')
```

See Also

matlab.unittest.constraints.Constraint | matlab.unittest.constraints.IssuesWarnings | matlab.unittest.constraints.Throws | warning

matlab.unittest.constraints.IssuesWarnings class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying function that issues expected warning profile

Description

The `IssuesWarnings` class creates a constraint that issues an expected warning profile. The constraint is satisfied only if the actual value is a function handle that issues a specific set of warnings. You specify warnings using warning identifiers.

By default, the constraint only confirms that when the testing framework invokes the function handle, MATLAB issues the specified set of warnings. It ignores the number of times the warnings are issued, in what order they are issued, and whether or not any unspecified warnings are issued. However, you can set parameters to respect the order, the count, and the warning set. Alternatively, you can specify the exact warning profile for comparison.

Construction

`outConstObj = IssuesWarnings(warnArr)` creates a constraint, `outConstObj`, specifying a function that issues expected warnings, `warnArr`.

`outConstObj = IssuesWarnings(expVal,Name,Value)` creates a constraint with additional options specified by one or more Name,Value pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`warnArr`

Warning identifiers expected when the testing framework invokes the function handle, specified as a cell array of warning identifiers. If `warnArr` is empty, the constructor throws an `MException`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Exactly'`

Indicator if the value is a function handle that must issue a warning profile that is an exact match, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance relies on specification of other parameters and default instance behavior to determine the strictness of its comparison. When set to `true`, the instance requires the warning profile to be exactly the same as the specified warning profile.

Default: `false`

`'RespectingCount'`

Indicator whether to respect element counts, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance is insensitive to the number of occurrences of members and ignores their frequency. When set to `true`, the instance is sensitive to the total number of set members. This means that, in addition to ensuring that all of the specified warnings are issued, this instance is not satisfied if the number of times that a particular warning issues differs from the number of times that warning is specified in `warnArr`.

Default: `false`

`'RespectingOrder'`

Indicator whether to respect the order of elements, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance is insensitive to the order of the set members. When set to `true`, the instance is sensitive to the order of the set members. This means that this instance is not satisfied if the order of the issued warnings differs from the order the warnings are specified in `warnArr`.

The order of a given set of warnings is determined by trimming the warning profiles to a profile with no repeated adjacent warnings. For example, the warning profile `{id:A, id:A, id:B, id:C, id:C, id:C, id:A, id:A, id:A}` is trimmed to `{id:A, id:B, id:C, id:A}`.

When this constraint respects order, the order of the warnings that are issued and expected must match the order of the expected warning profile. Warnings issued that are not listed in `warnArr` are ignored when determining order.

Default: `false`

`'RespectingSet'`

Indicator whether to respect set elements, specified as `false` or `true` (`logical 0` or `1`). When this value is `false`, the instance ignores additional set members. When set to `true`, the instance is sensitive to additional set members. This means that, in addition to ensuring that all of the specified warnings are issued, this instance is not satisfied if any extra, unspecified warnings are issued.

Default: `false`

`'WhenNargoutIs'`

Number of outputs the constraint should request when invoking the function handle, specified as a non-negative, real, scalar integer.

Default: `0`

Properties

`Exact`

Indicator of whether the constraint performs exact comparisons. Set this property through the constructor via the name-value pair argument, `'Exactly'`.

`ExpectedWarnings`

Expected warning identifiers. Set this read-only property through the constructor via the `warnArr` input argument.

`FunctionOutputs`

Output arguments produced at invocation of the supplied function handle, specified as a cell array. This property provides access to output arguments. It is read only and the testing framework sets it when it invokes the function handle. The number of outputs is determined by the `Nargout` property.

`Nargout`

Number of output arguments the instance uses when it executes functions. Set this property through the constructor via the name-value pair argument, `'WhenNargoutIs'`.

`RespectCount`

Indicator if the constraint respects the element counts, specified through the constructor via the name-value pair argument, `'RespectingCount'`.

`RespectOrder`

Indicator if the constraint respects the order of elements, specified through the constructor via the name-value pair argument, `'RespectingOrder'`.

`RespectSet`

Indicator if the constraint respects set elements, specified through the constructor via the name-value pair argument, `'RespectingSet'`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Instantiate `IssuesWarnings` Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IssuesWarnings

testCase = TestCase.forInteractiveUse;
```

Create a helper anonymous function for use in this example. Create several warning identifiers.

```
issueWarnings = @(idCell) cellfun(@(id) warning(id,'Message'), idCell);
```

```
firstID =   'first:id';
secondID =  'second:id';
thirdID =   'third:id';
```

Verify that the helper function issues a particular warning.

```
testCase.verifyThat(@() issueWarnings({firstID}),...
    IssuesWarnings({firstID}))
```

```
Interactive verification passed.
```

Verify the function issues a warning ignoring count, warning set, and order.

```
testCase.verifyThat(@() issueWarnings({firstID, thirdID, secondID,...
    firstID}), IssuesWarnings({secondID, firstID}))
```

```
Interactive verification passed.
```

Verify the function issues a warning while respecting the warning set.

```
testCase.verifyThat(@() issueWarnings({firstID, thirdID, secondID,...
    firstID}), IssuesWarnings({firstID, secondID, thirdID}, ...
    'RespectingSet', true))
```

```
Interactive verification passed.
```

Verify the function issues a warning while respecting the warning count.

```
testCase.verifyThat(@() issueWarnings({secondID, firstID, thirdID,...
    secondID}), IssuesWarnings({firstID, secondID, secondID}, ...
    'RespectingCount', true))
```

```
Interactive verification passed.
```

Verify the function issues a warning while respecting the warning order.

```
testCase.verifyThat(@() issueWarnings({firstID, secondID, secondID,...
    thirdID}), IssuesWarnings({firstID, secondID}, 'RespectingOrder', true))
```

```
Interactive verification passed.
```

Verify the function issues an exact match to the expected warning profile.

```
testCase.verifyThat(@() issueWarnings({firstID, secondID, secondID,...
    thirdID}), IssuesWarnings({firstID, secondID, secondID, thirdID}, ...
```

```
    'Exactly', true))

Interactive verification passed.
```

Verify that the constraint is not satisfied if the actual value is not a function handle.

```
testCase.verifyThat(5, IssuesWarnings({firstID}))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IssuesWarnings failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
```

Verify that the constraint is not satisfied if the function does not issue a warning.

```
testCase.verifyThat(@rand, IssuesWarnings({firstID}))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IssuesWarnings failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue any warnings.

    Expected Warning Profile:
         first:id

Evaluated Function:
```

```
        @rand
```

Verify that the constraint is not satisfied if the function issues a non-specified warning identifier.

```
testCase.verifyThat(@() issueWarnings({firstID}), IssuesWarnings({secondID}))
```

```
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID})
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 39
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 54
  In IssuesWarnings>IssuesWarnings.invoke at 409
  In IssuesWarnings>IssuesWarnings.issuesExpectedWarnings at 456
  In IssuesWarnings>IssuesWarnings.satisfiedBy at 242
  In QualificationDelegate>QualificationDelegate.qualifyThat at 58
  In Verifiable>Verifiable.verifyThat at 228
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IssuesWarnings failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue the correct warnings.

        Missing Warnings:
             second:id

    Actual Warning Profile:
         first:id
    Expected Warning Profile:
         second:id

Evaluated Function:
        @()issueWarnings({firstID})
```

Consider the following actual value and warning array.

```
actVal = @() issueWarnings({firstID, firstID, secondID, firstID});
```

```
warnArr = {firstID, secondID, firstID, firstID};
```

Test whether the warning array is exactly the same as the expected array.

```
testCase.verifyThat(actVal, IssuesWarnings(warnArr, 'Exactly', true))
```

```
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 39
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 54
  In IssuesWarnings>IssuesWarnings.invoke at 409
  In IssuesWarnings>IssuesWarnings.issuesExpectedWarnings at 456
  In IssuesWarnings>IssuesWarnings.satisfiedBy at 242
  In QualificationDelegate>QualificationDelegate.qualifyThat at 58
  In Verifiable>Verifiable.verifyThat at 228
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 39
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 54
  In IssuesWarnings>IssuesWarnings.invoke at 409
  In IssuesWarnings>IssuesWarnings.issuesExpectedWarnings at 456
  In IssuesWarnings>IssuesWarnings.satisfiedBy at 242
  In QualificationDelegate>QualificationDelegate.qualifyThat at 58
  In Verifiable>Verifiable.verifyThat at 228
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 39
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 54
  In IssuesWarnings>IssuesWarnings.invoke at 409
  In IssuesWarnings>IssuesWarnings.issuesExpectedWarnings at 456
  In IssuesWarnings>IssuesWarnings.satisfiedBy at 242
  In QualificationDelegate>QualificationDelegate.qualifyThat at 58
  In Verifiable>Verifiable.verifyThat at 228
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 39
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 54
```

```
  In IssuesWarnings>IssuesWarnings.invoke at 409
  In IssuesWarnings>IssuesWarnings.issuesExpectedWarnings at 456
  In IssuesWarnings>IssuesWarnings.satisfiedBy at 242
  In QualificationDelegate>QualificationDelegate.qualifyThat at 58
  In Verifiable>Verifiable.verifyThat at 228
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IssuesWarnings failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile must match exactly.
    --> The function handle did not issue the exact warning profile expected.

    Actual Warning Profile:
         first:id
         first:id
         second:id
         first:id
    Expected Warning Profile:
         first:id
         second:id
         first:id
         first:id

Evaluated Function:
        @()issueWarnings({firstID,firstID,secondID,firstID})
```

Test whether the warning array is the same as the expected array when respecting set, order and count.

```
testCase.verifyThat(actVal, IssuesWarnings(warnArr,...
    'RespectingSet',true,'RespectingOrder',true,'RespectingCount',true))

Interactive verification passed.
```

In this example, a constraint that specifies a warning profile that respects set, order and count is not the same as one that specifies an exact warning profile.

See Also

`matlab.unittest.constraints.IssuesNoWarnings` | `matlab.unittest.constraints.Throws` | `warning`

More About

- "Message Identifiers"

matlab.unittest.constraints.IsTrue class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying true value

Construction

`IsTrue` provides a constraint specifying a true value. This constraint is satisfied only by a scalar logical with a value of `true`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Tips

- For faster test execution, use `verifyTrue`, `assertTrue`, `assumeTrue`, or `fatalAssertTrue` instead of `IsTrue`.
- To display custom comparisons in the form of a function handle, use `ReturnsTrue` instead of `IsTrue`.

Examples

Test Actual Value Is True

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsTrue

testCase = TestCase.forInteractiveUse;
```

Test that `true` satisfies the `IsTrue` constraint.

```
testCase.verifyThat(true, IsTrue)
```

```
Interactive verification passed.
```

Test that the `IsTrue` constraint is not satisfied by `false`.

```
testCase.verifyThat(false, IsTrue)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsTrue failed.
--> The value must evaluate to "true".

Actual Value:
         0
```

The test fails because `false` returns `logical(0)`.

Test that the `IsTrue` constraint is not satisfied by the double `1`.

```
testCase.verifyThat(1, IsTrue)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsTrue failed.
--> The value must be logical. It is of type "double".

Actual Value:
         1
```

The `IsTrue` constraint is satisfied only by `logical(1)`.

Test that the `IsTrue` constraint is not satisfied by a logical array of ones.

```
testCase.verifyThat([true true true], IsTrue)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
```

```
---------------------
IsTrue failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         1     1     1
```

The `IsTrue` constraint is satisfied only if the value is scalar and `logical(1)`.

See Also

`IsFalse` | `ReturnsTrue`

matlab.unittest.constraints.LogicalComparator class

Package: matlab.unittest.constraints

Comparator for two logical values

Construction

`LogicalComparator` creates a comparator for two logical values. The comparator is satisfied if the actual and expected values have the same sparsity and the logical values are equivalent.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Compare Logical Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.LogicalComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Test the value of `true`.

```
testCase.assertThat(true, IsEqualTo(true, ...
    'Using', LogicalComparator))
```

```
Interactive assertion passed.
```

Test an array of `true` values.

```
testCase.assertThat([true true true], IsEqualTo(true, ...
```

```
    'Using', LogicalComparator))

Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> LogicalComparator failed.
    --> The logical values are not equal

Actual Logical Value:
         1     1     1
Expected Logical Value:
         1
Assertion failed.
```

The actual value must be a scalar logical to satisfy the constraint.

Compare the value of `1` to `true`.

```
testCase.verifyThat(1, IsEqualTo(true, 'Using', LogicalComparator))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> LogicalComparator failed.
    --> The logical values are not equal

Actual Logical Value:
         1
Expected Logical Value:
         1
```

Compare the value of `false` to `true`.

```
testCase.assertThat(false, IsEqualTo(true, 'Using', LogicalComparator))

Interactive assertion failed.

---------------------
Framework Diagnostic:
```

```
---------------------
IsEqualTo failed.
--> LogicalComparator failed.
    --> The logical values are not equal

Actual Logical Value:
        0
Expected Logical Value:
        1
Assertion failed.
```

See Also

`matlab.unittest.constraints.IsEqualTo`

Introduced in R2013a

matlab.unittest.constraints.Matches class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying string matches regular expression

Construction

`Matches(expr)` creates a constraint that specifies that a string matches a regular expression. The constraint is satisfied only if the actual value matches the given regular expression, `expr`.

`Matches(expr,'IgnoringCase',caseInsensitive)` creates a constraint indicating whether to ignore case difference.

Input Arguments

`expr`

Regular expression that the actual value must match to satisfy the constraint, specified as a string. `expr` can include newline characters.

`caseInsensitive`

Indicator if the constraint is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

Properties

`Expression`

Regular expression that the actual value must match, specified in the input argument, `expr`.

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the input argument, `ignoreCase`. This property applies at all levels of recursion, such as nested structures.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Matches Regular Expression

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.Matches

testCase = TestCase.forInteractiveUse;
```

Test that the actual value string, `'Epsilon Eridani'`, matches `'eps'`.

```
testCase.verifyThat('Epsilon Eridani', Matches('^eps'))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Matches failed.
--> The string did not match the regular expression.

Actual String:
    Epsilon Eridani
Regular Expression:
    ^eps
```

To satisfy the constraint, configure it to be case insensitive.

```
testCase.verifyThat('Epsilon Eridani', Matches('^eps', ...
```

```
    'IgnoringCase', true))
```

```
Interactive verification passed.
```

Define the regular expression that the actual value must match.

```
expr = 'Some[Ss]?tring';
```

The `[Ss]?` contained in the regular expression indicates that either `'S'` or `'s'` matches at that location 0 or 1 times.

Test that the actual values, `'SomeString'` and `'Somestring'`, satisfy the constraint.

```
testCase.verifyThat('SomeString', Matches(expr))
testCase.verifyThat('Somestring', Matches(expr))
```

```
Interactive verification passed.
Interactive verification passed.
```

Test that the actual value `'Sometring'` satisfies the constraint.

```
testCase.verifyThat('Sometring', Matches(expr))
```

```
Interactive verification passed.
```

Test that the actual value `'somestring'` satisfies the constraint.

```
testCase.verifyThat('somestring', Matches(expr))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Matches failed.
--> The string did not match the regular expression.

Actual String:
    somestring
Regular Expression:
    Some[Ss]?tring
```

See Also

`ContainsSubstring` | `EndsWithSubstring` | `IsSubstringOf` | `regexp` | `StartsWithSubstring`

More About

- "Regular Expressions"

matlab.unittest.constraints.NumericComparator class

Package: matlab.unittest.constraints

Comparator for numeric data types

Construction

`NumericComparator` creates a comparator for numeric data types. The comparator is satisfied if inputs are of the same class with equivalent size, complexity, and sparsity, and the built-in `isequaln` function returns `true`.

`NumericComparator('Within',tolObj)` creates a comparator using a specified tolerance. In this case, `NumericComparator` first checks for equivalent class, size, and sparsity of the actual and expected values. If these checks fail, the comparator is not satisfied. If these checks pass and the `isequaln` or complexity check fails, `NumericComparator` delegates comparison to the supplied tolerance, `tolObj`.

Input Arguments

`tolObj`

`matlab.unittest.constraints.Tolerance` instance

Properties

`Tolerance`

Specific tolerance used in construction of the comparator, specified as a `Tolerance` object in the `tolObj` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Compare Numeric Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.NumericComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Use a numeric comparator to test that `1.618` is equal to `1.618`.

```
testCase.verifyThat(1.618, IsEqualTo(1.618,...
    'Using', NumericComparator))
```

```
Interactive verification passed.
```

Verify that `(1+sqrt(5))/2` is equal to `1.618`.

```
testCase.verifyThat((1+sqrt(5))/2, IsEqualTo(1.618, ...
    'Using', NumericComparator))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index       Actual           Expected         Error                  Rel
                _____    ________________    ________    ____________________    ______

                1        1.61803398874989    1.618       3.39887498947977e-05    2.1006

Actual Value:
       1.618033988749895
Expected Value:
       1.618000000000000
```

Retest using a relative tolerance of `0.25%`.

```
import matlab.unittest.constraints.RelativeTolerance

testCase.verifyThat((1+sqrt(5))/2, IsEqualTo(1.618, ...
    'Using', NumericComparator('Within', RelativeTolerance(0.0025))))

Interactive verification passed.
```

See Also

matlab.unittest.constraints.IsEqualTo | matlab.unittest.constraints.Tolerance | isequaln

Introduced in R2013a

matlab.unittest.constraints.ObjectComparator class

Package: matlab.unittest.constraints

Comparator for MATLAB or Java objects

Construction

`ObjectComparator` creates a comparator for MATLAB or Java objects. The comparator is satisfied if the built-in `isequal` function returns `true`. However, if the class of the expected value defines an `isequaln` method, the `ObjectComparator` uses that method for comparison instead of `isequal`.

`ObjectComparator('Within',tolObj)` creates a comparator using a specified tolerance. `ObjectComparator` first checks that a call to `isequal` or `isequaln` returns `true`. If the check fails, the `ObjectComparator` checks for equivalent class, size, and sparsity of the actual and expected values. If these checks pass, `ObjectComparator` delegates comparison to the supplied tolerance, `tolObj`. The value of this tolerance must be of the same class as the actual and expected values.

Input Arguments

`tolObj`

`Tolerance` instance

Properties

`Tolerance`

Specific tolerance used in construction of the comparator, specified as a `matlab.unittest.constraints.Tolerance` object in the `tolObj` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Compare MATLAB Objects

In a file, `MyInt.m`, in your working folder, create a subclass of `int8`.

```
classdef MyInt < int8
    methods
        function i = MyInt(value)
            i@int8(value);
        end
    end
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.ObjectComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Use an `ObjectComparator` to test that two instances of `MyInt` are equal to each other.

```
testCase.verifyThat(MyInt(10), ...
    IsEqualTo(MyInt(10), 'Using', ObjectComparator))
```

```
Interactive verification passed.
```

Test the equality of two instances of `MyInt` that are constructed with different input values.

```
testCase.verifyThat(MyInt(11), ...
    IsEqualTo(MyInt(10), 'Using', ObjectComparator))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".

Actual Object:
```

```
      MyInt:

      int8 data:
       11
Expected Object:
      MyInt:

      int8 data:
       10
```

One instance of `MyInt` has a value of `11`, and the other has a value of `10`.

Repeat the test and specify that values must be equal within an absolute tolerance of `1`.

```
import matlab.unittest.constraints.AbsoluteTolerance

testCase.verifyThat(MyInt(11), IsEqualTo(MyInt(10), ...
    'Using', ObjectComparator('Within', AbsoluteTolerance(MyInt(1)))))
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.Tolerance` | `isequal`

Introduced in R2013a

matlab.unittest.constraints.PublicPropertyComparator class

Package: matlab.unittest.constraints

Comparator for public properties of MATLAB objects

Description

The `PublicPropertyComparator` compares public properties of MATLAB objects.

The `PublicPropertyComparator` supports MATLAB objects or arrays of objects and recursively compares data structures contained in the public properties. The `PublicPropertyComparator` is different from the `isequal` function because it examines only the public properties of the objects.

Construction

`PublicPropertyComparator` creates a comparator for public properties of MATLAB objects. This comparator supports only objects with no public properties.

`PublicPropertyComparator(compObj)` indicates a comparator, `compObj`, that defines the comparator used to compare public properties. This comparator supports recursion only in the data types supported by `compObj`.

`PublicPropertyComparator(compObj,Name,Value)` provides a comparator with additional options specified by one or more Name,Value pair arguments.

`PublicPropertyComparator(Name,Value)` provides a comparator for MATLAB objects with no public properties with additional options specified by one or more Name,Value pair arguments.

`PublicPropertyComparator.supportingAllValues` creates a comparator for public properties of MATLAB objects. This comparator supports any value in recursion. `supportingAllValues` is a `Static` method of the `PublicPropertyComparator` class.

You typically pass this comparator to another constraint, such as `IsEqualTo`. You can use the `Name,Value` pairs of the `IsEqualTo`

constraint in combination with a comparator constructed with the `PublicPropertyComparator.supportingAllValues` syntax.

Input Arguments

`compObj`

Comparator object

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoringCase'`

Indicator if the comparator is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to case. The comparator uses this name-value pair only if the content being compared consists of strings.

Default: `false`

`'IgnoringWhitespace'`

Indicator if the comparator is insensitive to whitespace characters, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to whitespace characters. Whitespace characters consist of space, form feed, new line, carriage return, horizontal tab, and vertical tab. The comparator uses this name-value pair only if the contents being compared consists of strings.

Default: `false`

`'Recursively'`

Indicator of whether comparator operates recursively, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the comparator does not operate recursively on its data.

When the value is `true`, the data types that the public property comparator supports are fully supported in recursion.

Default: `false`

`'Within'`

Tolerance to use for numerical comparison, specified as a `matlab.unittest.constraints.Tolerance` object. The comparator uses this name-value pair only if the contents being compared consist of numeric types.

Default: (empty)

Properties

`IgnoreCase`

Indicator if the comparator is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`

`IgnoreWhitespace`

Indicator if the comparator is insensitive to whitespace characters, specified in the name-value pair argument, `'IgnoringWhitespace'`

`Recursive`

Indicator of whether comparator operates recursively, specified in the name-value pair argument, `'Recursively'`

`Tolerance`

Specific tolerance used in construction of the comparator, specified as a `matlab.unittest.constraints.Tolerance` object in the name-value pair argument, `'Within'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Instantiate `PublicPropertyComparator` Object

In a file in your working folder, construct this `Employee` class.

```
classdef Employee
    properties (SetAccess=immutable)
        Name
    end
    properties (Access=private)
        Location
    end
    methods
        function obj = Employee(name,location)
            obj.Name = name;
            obj.Location = location;
        end
    end
end
```

At the command prompt, create two instances of the `Employee` class.

```
e1 = Employee('sam','Building A');
e2 = Employee('Sam','Building B');
```

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.PublicPropertyComparator
import matlab.unittest.constraints.StringComparator

testCase = TestCase.forInteractiveUse;
```

Construct a comparator and verify that `e1` and `e2` are equal.

```
compObj = PublicPropertyComparator;
testCase.verifyThat(e1, IsEqualTo(e2,'Using',compObj))
```

```
Error using matlab.unittest.constraints.ComparatorList/throwUnsupportedType (line 148)
No comparator in list supports the char below:
    Sam
List contains the following comparators:
   <empty>
```

```
Error in matlab.unittest.constraints.ComparatorList/getAndPrepareComparatorFor (line 19
               list.throwUnsupportedType(expVal);

Error in matlab.unittest.constraints.ComparatorList/satisfiedBy (line 129)
            comp = list.getAndPrepareComparatorFor(expVal);

Error in matlab.unittest.internal.constraints.FieldComparator/findFirstDifferingField (
                  bool = comp.satisfiedBy(actField, expField);

Error in matlab.unittest.internal.constraints.FieldComparator/fieldsHaveSameContents (l
            bool = comparator.findFirstDifferingField(actVal, expVal);

Error in matlab.unittest.internal.constraints.FieldComparator/satisfiedBy (line 22)
            bool = haveSameClass(actVal, expVal) && ...

Error in matlab.unittest.constraints.IsEqualTo/satisfiedBy (line 165)
            bool = comp.supports(actual) && comp.satisfiedBy(actual, constraint.Expecte

Error in matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (lin
               result = constraint.satisfiedBy(actual);

Error in matlab.unittest.qualifications.Verifiable/verifyThat (line 206)
            qualifyThat(verifiable.VerificationDelegate, actual, constraint, varargin{
```

The test fails because, by default, the `PublicPropertyComparator` does not support strings.

Construct a comparator that supports strings. Specify that the comparison is not case-sensitive.

```
compObj = PublicPropertyComparator(StringComparator, 'IgnoringCase',true);
testCase.verifyThat(e1, IsEqualTo(e2,'Using',compObj))
```

```
Interactive verification passed.
```

Note that the test passes even though `e1.Location` and `e2.Location` are not the same. Since `Location` is a private property, the comparator does not compare its contents.

Instantiate `PublicPropertyComparator` Object to Support All Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
```

```
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.PublicPropertyComparator

testCase = TestCase.forInteractiveUse;
```

Test a passing case.

```
m1 = MException('Msg:ID','MsgText');
m2 = MException('Msg:ID','MsgText');
testCase.verifyThat(m1, IsEqualTo(m2, 'Using', ...
    PublicPropertyComparator.supportingAllValues))

Interactive verification passed.
```

Test a failing case.

```
m1 = MException('Msg:ID','MsgText');
m2 = MException('Msg:ID','msgtext');
testCase.verifyThat(m1, IsEqualTo(m2, 'Using', ...
    PublicPropertyComparator.supportingAllValues))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> StringComparator failed.
    Path to failure: <Value>.message
    --> The strings are not equal

    Actual String:
        MsgText
    Expected String:
        msgtext

Actual MException:
      MException with properties:

        identifier: 'Msg:ID'
           message: 'MsgText'
             cause: {}
             stack: [0x1 struct]
Expected MException:
      MException with properties:
```

```
        identifier: 'Msg:ID'
           message: 'msgtext'
             cause: {}
             stack: [0x1 struct]
```

Test a case that passes when the comparator ignores differences in case.

```
m1 = MException('Msg:ID','MsgText');
m2 = MException('Msg:ID','msgtext');
testCase.verifyThat(m1, IsEqualTo(m2, 'IgnoringCase', true,...
    'Using', PublicPropertyComparator.supportingAllValues))
```

```
Interactive verification passed.
```

Instantiate `PublicPropertyComparator` Object with Tolerance

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance
import matlab.unittest.constraints.PublicPropertyComparator

testCase = TestCase.forInteractiveUse;
```

Define actual and expected `timeseries` objects. Perturb one of the actual data points by 1%.

```
expected = timeseries(1:10);
actual = expected;
actual.Data(7) = 1.01*actual.Data(7);
```

Test that the actual and expected values are equal within a relative tolerance of 2%.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".
    --> The tolerance does not support timeseries values so it was not used.
```

```
Actual Object:
      timeseries

      Common Properties:
                Name: 'unnamed'
                Time: [10x1 double]
            TimeInfo: [1x1 tsdata.timemetadata]
                Data: [1x1x10 double]
            DataInfo: [1x1 tsdata.datametadata]

      More properties, Methods
Expected Object:
      timeseries

      Common Properties:
                Name: 'unnamed'
                Time: [10x1 double]
            TimeInfo: [1x1 tsdata.timemetadata]
                Data: [1x1x10 double]
            DataInfo: [1x1 tsdata.datametadata]

      More properties, Methods
```

Use the `PublicPropertyComparator` in the construction of the constraint.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02),...
    'Using', PublicPropertyComparator.supportingAllValues))
```

```
Interactive verification passed.
```

The test passes because the `PublicPropertyComparator` compares each public property individually instead of comparing the object all at once. In the former test, the `ObjectComparator` compares `timeseries` objects, and therefore relies on the `isequal` method of the `timeseries` class. Due to the perturbation in the actual `timeseries`, `isequal` returns `false`. The comparator does not apply the tolerance because the double-valued tolerance cannot apply directly to the `timeseries` object. In the latter test, the comparator applies the tolerance to each public property that contains double-valued data.

See Also

`matlab.unittest.constraints.IsEqualTo` |
`matlab.unittest.constraints.ObjectComparator`

Introduced in R2014a

matlab.unittest.constraints.RelativeTolerance class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Tolerance

Relative numeric tolerance

Description

This numeric `Tolerance` assesses the magnitude of the difference between actual and expected values, relative to the expected value. For the tolerance to be satisfied, `abs(expVal - actVal) <= relTol.*abs(expVal)` must be true.

Construction

`RelativeTolerance(tolVals)` creates a relative tolerance object that assesses the magnitude of the difference between actual and expected values, relative to the expected value.

The data types of the inputs to the `RelativeTolerance` constructor determine which data types the tolerance supports. For example, `RelativeTolerance(10*eps)` constructs a `RelativeTolerance` for comparing double-precision numeric arrays while `RelativeTolerance(int8(2))` constructs a `RelativeTolerance` for comparing numeric arrays of type `int8`. If the actual and expected values being compared contain more than one numeric data type, the tolerance only applies to the data types specified by the values passed into the constructor.

To specify different tolerance values for different data types, you can pass multiple tolerance values to the constructor. For example, `RelativeTolerance(10*eps, 10*eps('single'), int8(1))` constructs an `RelativeTolerance` that applies the following absolute tolerances:

- `10*eps` applies a relative tolerance of `10*eps` for double-precision numeric arrays.
- `10*eps('single')` applies a relative tolerance of `10*eps` for single-precision numeric arrays.
- `int8(1)` applies a relative tolerance of `1` for numeric arrays of type `int8`.

You can specify more than one tolerance for a particular data type by combining tolerances with the & and | operators. To combine two tolerances, the sizes of the tolerance values for each data type must be compatible.

Input Arguments

`tolVals`

Numeric tolerances, specified as a comma-separated list of numeric arrays. Each input argument contains the tolerance specification for a particular data type. Each numeric array can be a scalar or array the same size as the actual and expected values.

Default:

Properties

`Values`

Numeric tolerances, specified by the `tolVals` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test with Relative Tolerance

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Assert that the difference between an actual value, `4.1`, and an expected value, `4.5`, is less than 10%.

```
testCase.assertThat(4.1, IsEqualTo(4.5, ...
    'Within', RelativeTolerance(0.1)))
```

```
Interactive assertion passed.
```

Specify Relative Tolerance for Different Data Types

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Create the following actual and expected cell arrays.

```
act = {'abc', 123, single(106)};
exp = {'abc', 122, single(105)};
```

Test that the arrays satisfy the `RelativeTolerance` constraint within 2%.

```
testCase.verifyThat(act, IsEqualTo(exp, ...
    'Within', RelativeTolerance(0.02)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    Path to failure: <Value>{3}
    --> The values are not equal using "isequaln".
    --> The tolerance does not support single values so it was not used.
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        106       105         1        0.00952381

    Actual Value:
```

```
           106
    Expected Value:
           105

Actual cell:
        'abc'    [123]    [106]
Expected cell:
        'abc'    [122]    [105]
```

The test fails because the tolerance is only applied to the `double` data type.

Create a tolerance object that specifies different tolerances for different data types.

```
tolObj = RelativeTolerance(0.02, single(0.02));
```

A tolerance of `2%` is a applied to `double` and `single` valued data.

Verify that the expected and actual values satisfy the `RelativeTolerance` constraint.

```
testCase.verifyThat(act, IsEqualTo(exp, 'Within', tolObj))
```

```
Interactive verification passed.
```

Combine Relative and Absolute Tolerances

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Define an actual value approximation for `pi`.

```
act = 3.14;
```

Construct a tolerance object to test that the difference between the actual and expected values is within `0.001` and within `0.25%`.

```
tolObj = AbsoluteTolerance(0.001) & RelativeTolerance(0.0025);
```

Verify that the actual value is within the tolerance of the expected value of `pi`.

```
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> AndTolerance failed.
        --> AbsoluteTolerance failed.
            --> The error was not within absolute tolerance.
        --> RelativeTolerance passed.
        --> Failure table:
                    Index    Actual        Expected               Error
                    _____    ______    _________________    ____________________

                    1        3.14      3.14159265358979     -0.00159265358979299

Actual Value:
       3.140000000000000
Expected Value:
       3.141592653589793
```

The actual value does not satisfy the `AbsoluteTolerance` constraint.

Construct a constraint that is satisfied if the values are within `0.001` or `0.25%`, and then retest the actual value.

```
tolObj = AbsoluteTolerance(0.001) | RelativeTolerance(0.0025);
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))
```

```
Interactive verification passed.
```

Combine Absolute and Relative Tolerances to Test Small and Large Values

Combine tolerances so when you test the equality of values, an absolute (floor) tolerance dominates when the values are near zero, and a relative tolerance dominates for larger values.

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance
```

```
testCase = TestCase.forInteractiveUse;
```

Define two structures containing electromagnetic properties of a vacuum. One structure, `approxVacuumProps`, contains approximate values for the permeability and speed of light in a vacuum.

```
approxVacuumProps.Permeability = 1.2566e-06; % Approximate
approxVacuumProps.Permitivity = 8.854187817*10^-12;
approxVacuumProps.LightSpeed = 2.9979e+08; % Approximate

baselineVacuumProps.Permeability = 4*pi*10^-7;
baselineVacuumProps.Permitivity = 8.854187817*10^-12;
baselineVacuumProps.LightSpeed = 1/sqrt(...
    baselineVacuumProps.Permeability*baselineVacuumProps.Permitivity);
```

Test that the relative difference between the approximate and baseline values is within `eps*1e11`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    Path to failure: <Value>.Permeability
    --> The values are not equal using "isequaln".
    --> RelativeTolerance failed.
        --> The error was not within relative tolerance.
        --> Failure table:
                    Index      Actual            Expected                  Error
                    _____    __________    ____________________    ____________________

                    1        1.2566e-06    1.25663706143592e-06    -3.70614359173257e-

    Actual Value:
             1.256600000000000e-06
    Expected Value:
             1.256637061435917e-06

Actual struct:
```

```
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 299790000
Expected struct:
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the relative difference in the permeabilities is not within the tolerance. The difference between the two values is small, but the numbers are close to zero, so the difference relative to their size is not small enough to satisfy the tolerance.

Construct a tolerance object to test that the absolute difference between the approximate and baseline values is within `1e-4`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', AbsoluteTolerance(1e-4)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    Path to failure: <Value>.LightSpeed
    --> The values are not equal using "isequaln".
    --> AbsoluteTolerance failed.
        --> The error was not within absolute tolerance.
        --> Failure table:
                    Index     Actual          Expected              Error             R
                    _____    _________    _________________    _________________    ___

                    1        299790000    299792458.010503    -2458.01050287485    -8.1

    Actual Value:
           299790000
    Expected Value:
             2.997924580105029e+08

Actual struct:
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 299790000
Expected struct:
```

```
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the absolute difference in the speed of light is not within the tolerance. The difference between the two values is small relative to their size, but too large to satisfy the tolerance.

Construct a logical disjunction of tolerance objects to test that the absolute difference between the approximate and baseline values is within `1e-4` or the relative difference is within `eps*1e11`. The test uses this tolerance so permeability values that are close to zero satisfy the absolute (floor) tolerance, and speed of light values that are large, satisfy the relative tolerance.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)| AbsoluteTolerance(1e-4)))
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.AbsoluteTolerance` | `matlab.unittest.constraints.IsEqualTo`

matlab.unittest.constraints.ReturnsTrue class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying function handle that returns true

Construction

`ReturnsTrue` provides a constraint specifying that a function handle that returns true. The constraint is satisfied only by a function handle that returns a scalar logical with a value of `true`.

When negated using the tilde operator, `~`, this constraint not only passes when the function handle returns `false`, but also when the function handle returns any non-scalar value (such as `[true true]`) or any non-logical (such as integer valued1).

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Tips

- To display custom comparisons in the form of a function handle, use `ReturnsTrue` instead of `IsTrue`.

Examples

Test Actual Value Specified by Function Handle Returns True

These comparisons are shown for example only. There are other constraints that might better handle the particular comparisons.

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.ReturnsTrue

testCase = TestCase.forInteractiveUse;
```

Verify that the `ReturnsTrue` constraint is satisfied by the value returned by a handle to `true`.

```
testCase.verifyThat(@true, ReturnsTrue)
```

```
Interactive verification passed.
```

Verify that the `ReturnsTrue` constraint is not satisfied by a handle to `false`.

```
testCase.verifyThat(@false, ReturnsTrue)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
ReturnsTrue failed.
--> The function handle should have evaluated to "true".
--> Returned value:
             0

Actual Function Handle:
        @false
```

Verify that a call to `isequal` returns true.

```
testCase.verifyThat(@() isequal(1,1), ReturnsTrue)
```

```
Interactive verification passed.
```

Verify that a function that returns a double-valued 1 does not satisfy the `ReturnsTrue` constraint.

```
testCase.verifyThat(@() double(true), ReturnsTrue)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
ReturnsTrue failed.
```

```
--> The function handle should have returned a logical value. It was of type "double".
--> Returned value:
             1

Actual Function Handle:
        @()double(true)
```

Verify that the negation of a string comparison of `'a'` and `'b'` returns true.

```
testCase.verifyThat(@() ~strcmp('a','b'), ReturnsTrue)
```

```
Interactive verification passed.
```

Test if a comparison of `'a'` to the cell array `{'a','a'}` returns true.

```
testCase.verifyThat(@() strcmp('a',{'a','a'}), ReturnsTrue)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
ReturnsTrue failed.
--> The function handle should have returned a scalar. The return value had a size of |
--> Returned value:
             1     1

Actual Function Handle:
        @()strcmp('a',{'a','a'})
```

The constraint is not satisfied because the call to `strcmp` results a logical array, not a logical scalar.

See Also

`Constraint` | `IsTrue`

matlab.unittest.constraints.StartsWithSubstring class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.BooleanConstraint

Constraint specifying string starting with substring

Construction

`StartsWithSubstring(prefix)` creates a constraint specifying a string starting with a substring. The constraint is satisfied only if the actual value starts with an expected string, `prefix`.

`StartsWithSubstring(prefix,Name,Value)` provides a constraint with additional options specified by one or more Name,Value pair arguments. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`prefix`

Text at the start of the actual value, specified as a string. `prefix` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoringCase'`

Indicator if the constraint is insensitive to case, specified as `false` or `true` (`logical 0` or `1`)

Default: `false`

`'IgnoringWhitespace'`

Indicator if the constraint is insensitive to whitespace, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

Properties

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`. This property applies at all levels of recursion, such as nested structures.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`. This property applies at all levels of recursion, such as nested structures.

`Prefix`

Text at the start of the actual value, specified in the input argument, `prefix`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test That Actual Value Starts with Specified Substring

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.StartsWithSubstring
```

```
testCase = TestCase.forInteractiveUse;
```

Define the actual value string.

```
actVal = 'This Is One Long String';
```

Test that `actVal` starts with the substring `'This'`.

```
testCase.verifyThat(actVal, StartsWithSubstring('This'))
```

```
Interactive verification passed.
```

Test that `actVal` starts with the substring `'this is'`.

```
testCase.verifyThat(actVal, StartsWithSubstring('this is'))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
StartsWithSubstring failed.
--> The string has an incorrect prefix.

Actual String:
    This Is One Long String
Expected Prefix:
    this is
```

By default, the `StartsWithSubstring` constraint is case sensitive.

Repeat the test, this time ignoring case.

```
testCase.verifyThat(actVal, StartsWithSubstring('this is',...
    'IgnoringCase', true))
```

```
Interactive verification passed.
```

Test that `actVal` starts with the substring `'thisisone'`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat(actVal, StartsWithSubstring('thisisone', ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Interactive verification passed.
```

Assert that `actVal` does not start with the substring `'long'`.

```
testCase.assertThat(actVal, ~StartsWithSubstring('Long'))
```

```
Interactive assertion passed.
```

See Also

`ContainsSubstring` | `EndsWithSubstring` | `IsSubstringOf` | `Matches`

matlab.unittest.constraints.StringComparator class

Package: matlab.unittest.constraints

Comparator for two strings

Construction

`StringComparator` creates a comparator for two strings. The comparator is satisfied if the two strings are equal.

`StringComparator(Name,Value)` creates a comparator with additional options specified by one or more Name,Value pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoringCase'`

Indicator if the comparator is insensitive to case, specified as `false` or `true` (`logical 0` or `1`). When it is `false`, the comparator is sensitive to case.

Default: `false`

`'IgnoringWhitespace'`

Indicator if the comparator is insensitive to whitespace characters, specified as `false` or `true` (`logical 0` or `1`). When it is `false`, the comparator is sensitive to whitespace characters. Whitespace characters consist of space, form feed, new line, carriage return, horizontal tab, and vertical tab.

Default: `false`

Properties

`IgnoreCase`

Indicator if the comparator is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`

`IgnoreWhitespace`

Indicator if the comparator is insensitive to whitespace characters, specified in the name-value pair argument, `'IgnoringWhitespace'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Compare Cell Arrays

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.StringComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Verify that the actual and expected strings are equal using a string comparator.

```
expected = 'coffee';
actual = 'coffee';
testCase.verifyThat(actual,IsEqualTo(expected, ...
    'Using', StringComparator))
```

```
Interactive verification passed.
```

Change the actual string and repeat the comparison.

```
expected = 'coF Fee';
```

```
testCase.verifyThat(actual,IsEqualTo(expected, ...
    'Using', StringComparator))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> StringComparator failed.
    --> The strings are not equal

Actual String:
    coffee
Expected String:
    coF Fee
```

For the test to pass, construct a comparator that ignores case and whitespace characters.

```
testCase.verifyThat(actual,IsEqualTo(expected, ...
    'Using', StringComparator('IgnoringCase', true, ...
    'IgnoringWhitespace', true)))

Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsEqualTo` | `strcmp`

Introduced in R2013a

matlab.unittest.constraints.StructComparator class

Package: matlab.unittest.constraints

Comparator for MATLAB structure arrays

Construction

`StructComparator` creates a comparator for MATLAB structure arrays.

`StructComparator(compObj)` indicates a comparator, `compObj`, that defines the comparator used to compare values contained in the structure. By default, a `StructComparator` supports only empty structure arrays.

`StructComparator(compObj,Name,Value)` provides a comparator with additional options specified by one or more Name,Value pair arguments.

`StructComparator(Name,Value)` provides a comparator for empty structure arrays with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`compObj`

Comparator object

A comparator is passed into the `StructComparator` to provide support for data types during recursion. By default, the `StructComparator` supports only empty structure arrays.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'IgnoringCase'

Indicator if the comparator is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to case. The comparator uses this name-value pair only if the contents being compared are strings.

Default: `false`

'IgnoringWhitespace'

Indicator if the comparator is insensitive to whitespace characters, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to whitespace characters. Whitespace characters consist of space, form feed, new line, carriage return, horizontal tab, and vertical tab. The comparator uses this name-value pair only if the contents being compared are strings.

Default: `false`

'Recursively'

Indicator of whether comparator operates recursively, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the comparator does not operate recursively on its data.

When the value is `true`, the data types the `StructComparator` supports are fully supported in recursion. For example:

```
comp1 = StructComparator(NumericComparator);
comp2 = StructComparator(NumericComparator, 'Recursively', true);
```

Both `comp1` and `comp2` support structures that contain numeric values as their fields. However, only `comp2` supports structures that recursively contain either structures or numeric values as their fields.

Default: `false`

'Within'

Tolerance to use for numerical comparison, specified as a `matlab.unittest.constraints.Tolerance` object. This name-value pair is applicable to contents that are numeric types.

Default: (empty)

Properties

`IgnoreCase`

Indicator if the comparator is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`

`IgnoreWhitespace`

Indicator if the comparator is insensitive to whitespace characters, specified in the name-value pair argument, `'IgnoringWhitespace'`

`Recursive`

Indicator of whether comparator operates recursively, specified in the name-value pair argument, `'Recursively'`

`Tolerance`

Specific tolerance used in construction of the comparator, specified as a `matlab.unittest.constraints.Tolerance` object in the name-value pair argument, `'Within'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Compare Numeric Structures

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.StructComparator
import matlab.unittest.constraints.NumericComparator
```

```
testCase = TestCase.forInteractiveUse;
```

Create two equal structures.

```
s1 = struct('id',7,'score',7.3);
s2 = s1;
```

Test that the structures are equal. By default, the `StructComparator` supports only empty structures, so you need to configure the comparator with a `NumericComparator`.

```
testCase.verifyThat(s1, IsEqualTo(s2, 'Using', ...
    StructComparator(NumericComparator)))
```

```
Interactive verification passed.
```

Change the score of `s2` and compare the structures again.

```
s2.score = 7.6;
testCase.verifyThat(s1, IsEqualTo(s2, 'Using', ...
    StructComparator(NumericComparator)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    Path to failure: <Value>.score
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error       RelativeError
                _____    ______    ________    _____    __________________

                1        7.3       7.6         -0.3     -0.0394736842105263

    Actual Value:
           7.300000000000000
    Expected Value:
           7.600000000000000

Actual struct:
           id: 7
        score: 7.300000000000000
```

```
Expected struct:
           id: 7
        score: 7.600000000000000
```

Specify an absolute tolerance for the comparison.

```
testCase.verifyThat(s1, IsEqualTo(s2, 'Using', ...
    StructComparator(NumericComparator, 'Within', ...
    AbsoluteTolerance(0.5))))
```

```
Interactive verification passed.
```

Compare Character Structures

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.StructComparator
import matlab.unittest.constraints.StringComparator

testCase = TestCase.forInteractiveUse;
```

Create two structures. Make one of the fields a nested structure.

```
e1 = struct('name', struct('first','sam','last','smith'), ...
    'location','Building A');
e2 = e1;
```

Verify that the two structures are equal. Since the struct contains a nested structure, configure the constraint to operate recursively.

```
testCase.verifyThat(e1, IsEqualTo(e2, 'Using', ...
    StructComparator(StringComparator, 'Recursively', true)))
```

```
Interactive verification passed.
```

Change the first name field of the `e2` structure and repeat the comparison.

```
e2.name.first = ' SAM';
testCase.verifyThat(e1, IsEqualTo(e2, 'Using', ...
    StructComparator(StringComparator, 'Recursively', true)))
```

```
Interactive verification failed.

--------------------
```

```
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> StringComparator failed.
    Path to failure: <Value>.name.first
    --> The strings are not equal

    Actual String:
        sam
    Expected String:
         SAM

Actual struct:
            name: [1x1 struct]
        location: 'Building A'
Expected struct:
            name: [1x1 struct]
        location: 'Building A'
```

Configure the comparator to ignore case and whitespace characters.

```
testCase.verifyThat(e1, IsEqualTo(e2, 'Using', ...
    StructComparator(StringComparator, 'Recursively', true, ...
    'IgnoringCase', true, 'IgnoringWhitespace', true)))
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.Tolerance`

Introduced in R2013a

matlab.unittest.constraints.Throws class

Package: matlab.unittest.constraints
Superclasses: matlab.unittest.constraints.Constraint

Constraint specifying function handle that throws MException

Description

The `Throws` class creates a constraint that is satisfied only if the actual value is a function handle that throws a specific exception.

If the function throws an `MException` and the `ExpectedException` property of the constraint is an error identifier, a qualification failure occurs if the actual `MException` thrown has a different identifier. Alternately, if the `ExpectedException` property is a `meta.class`, the constraint is not satisfied if the actual `MException` thrown does not derive from the `ExpectedException`.

Construction

`outConstObj = Throws(excep)` provides a constraint, `outConstObj`, specifying a function handle that throws a particular `MException`, `excep`.

`outConstObj = Throws(excep,Name,Value)` provides a constraint with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`excep`

Error identifier or `meta.class` representing the specific type of expected exception. If `excep` is a `meta.class` but does not derive from `MException`, the `Throws` constructor throws an `MException`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'CausedBy'`

Expected causes, specified as a cell array of strings or an array of `meta.class` instances.

The testing results in a qualification failure if any causes specified in `CausedBy` are not found within the cause tree.

Default: `{}`

`'WhenNargoutIs'`

Number of outputs the constraint should request when invoking the function handle, specified as a non-negative, real, scalar integer.

Default: `0`

Properties

`ExpectedException`

Expected `MException` identifier or class. Set this read-only property through the constructor via the `excep` input argument.

`Nargout`

Number of output arguments the instance uses when executing functions. Set this property through the constructor via the name-value pair argument, `'WhenNargoutIs'`.

`RequiredCauses`

Expected causes for the function handle throwing an `MException`. Set this property through the constructor via the name-value pair argument, `'CausedBy'`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Instantiate Throws Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.Throws

testCase = TestCase.forInteractiveUse;
```

Verify that a function throws a specified error id.

```
testCase.verifyThat(@() error('SOME:error:id','Error!'), ...
    Throws('SOME:error:id'))
```

```
Interactive verification passed.
```

Verify that a function throws a specified exception class.

```
testCase.verifyThat(@() error('SOME:error:id','Error!'), ...
    Throws(?MException))
```

```
Interactive verification passed.
```

Verify that a function, when called with a specified number of outputs, throws a specified error.

```
testCase.verifyThat(@() disp('hi'), Throws('MATLAB:maxlhs', ...
    'WhenNargoutIs', 1))
```

```
Interactive verification passed.
```

Check causes by identifier.

```
me = MException('TOP:error:id','TopLevelError!');
causeBy = MException('causedBy:someOtherError:id','CausedByError!');
me  = me.addCause(causeBy);
testCase.verifyThat(@() me.throw, Throws('TOP:error:id','CausedBy', ...
    {'causedBy:someOtherError:id'}))
```

```
Interactive verification passed.
```

Check causes by class.

```
me = MException('TOP:error:id','TopLevelError!');
```

```
causeBy = MException('causedBy:someOtherError:id','CausedByError!');
me = me.addCause(causeBy);
testCase.verifyThat(@() me.throw, Throws('TOP:error:id','CausedBy', ...
    ?MException))

Interactive verification passed.
```

Verify that if the actual value is not a function handle, the constraint is not satisfied.

```
testCase.fatalAssertThat(5, Throws('some:id'))

Interactive fatal assertion failed.

----------------------
Framework Diagnostic:
----------------------
Throws failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
Fatal assertion failed.
```

Verify that if the function does not throw an exception, the constraint is not satisfied.

```
testCase.assumeThat(@rand, Throws(?MException))

Interactive assumption failed.

----------------------
Framework Diagnostic:
----------------------
Throws failed.
--> The function did not throw any exception.

    Expected Exception Type:
        MException

Evaluated Function:
        @rand
```

```
Assumption failed.
```

Verify that if the function issues a non-specified error identifier, the constraint is not satisfied.

```
testCase.verifyThat(@() error('SOME:id','Error!'), Throws('OTHER:id'))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Throws failed.
--> The function threw the wrong exception.

    Actual Exception:
        SOME:id
    Expected Exception:
        OTHER:id

Evaluated Function:
        @()error('SOME:id','Error!')
```

Verify that if the function throws an exception and the cause does not match the specified identifier, the constraint is not satisfied.

```
testCase.verifyThat(@() error('TOP:error:id','TopLevelError!'), ...
    Throws('TOP:error:id','CausedBy',{'causedBy:someOtherError:id'}))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Throws failed.
--> The following causes were not found in the exception tree:
    --> causedBy:someOtherError:id

Evaluated Function:
        @()error('TOP:error:id','TopLevelError!')
```

See Also

`MException` | `matlab.unittest.constraints.IssuesWarnings` | `error` | `matlab.unittest.constraints`

More About

- "Message Identifiers"

Introduced in R2013a

matlab.unittest.constraints.Tolerance class

Package: matlab.unittest.constraints

Abstract interface class for tolerances

Description

Tolerances define a notion of fuzzy equality for a set of data types and can be plugged in to the `IsEqualTo` constraint through the `'Within'` name-value pair argument.

Classes that derive from the `Tolerance` interface class must provide a tolerance definition. Use the `satisfiedBy` method to implement the tolerance definition. Classes that derive from the `Tolerance` class also must provide a diagnostic for two compared values. The testing framework uses the diagnostic when the compared values are outside of the allowable tolerance. Use the `getDiagnosticFor` method to implement this condition. Finally, classes that derive from the `Tolerance` class must provide a means to determine which data types the tolerance supports. Define the supported data types by implementing the `supports` method.

Methods

getDiagnosticFor	Produce diagnostic for two values specified to be within tolerance
satisfiedBy	Determine whether two values are within tolerance
supports	Determine whether tolerance supports specified data type

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Custom Tolerance Class

Determine if two DNA sequences have a Hamming distance within a specified tolerance. For two DNA sequences (strings) of the same length, the Hamming distance is the number of positions in which the nucleotides (letters) of one sequence differ from the other.

In a file, `DNA.m`, in your working folder, create a simple class for a DNA sequence.

```
classdef DNA
    properties(SetAccess=immutable)
        Sequence
    end

    methods
        function dna = DNA(sequence)
            validLetters = ...
                sequence == 'A' | ...
                sequence == 'C' | ...
                sequence == 'T' | ...
                sequence == 'G';

            if ~all(validLetters(:))
                error('Sequence contained a letter not found in DNA.')
            end
            dna.Sequence = sequence;
        end
    end
end
```

In a file in your working folder, create a tolerance class so that you can test that DNA sequences are within a specified Hamming distance. The constructor requires a `Value` property that defines the maximum Hamming distance.

```
classdef HammingDistance < matlab.unittest.constraints.Tolerance
    properties
        Value
    end

    methods
        function tolerance = HammingDistance(value)
            tolerance.Value = value;
```

```
        end
    end
end
```

In a `methods` block with the `HammingDistance` class definition, include the following method so that the tolerance supports DNA objects. Tolerance classes must implement a `supports` method.

```
    methods
        function tf = supports(~, value)
            tf = isa(value, 'DNA');
        end
    end
```

In a `methods` block with the `HammingDistance` class definition, include the following method that returns `true` or `false`. Tolerance classes must implement a `satisfiedBy` method. The testing framework uses this method to determine if two values are within the tolerance.

```
    methods
        function tf = satisfiedBy(tolerance, actual, expected)
            if ~isSameSize(actual.Sequence, expected.Sequence)
                tf = false;
                return
            end
            tf = hammingDistance(actual.Sequence,expected.Sequence) <= tolerance.Value;
        end
    end
```

In the `HammingDistance.m` file, define the following helper functions outside of the `classdef` block. The `isSameSize` function returns `true` if two DNA sequences are the same size, and the `hammingDistance` function returns the Hamming distance between two sequences.

```
function tf = isSameSize(str1, str2)
tf = isequal(size(str1), size(str2));
end

function distance = hammingDistance(str1, str2)
distance = nnz(str1 ~= str2);
end
```

The function returns a `Diagnostic` object with information about the comparison. In a `methods` block with the `HammingDistance` class definition, include the following

method that returns a `StringDiagnostic`. Tolerance classes must implement a `getDiagosticFor` method.

```
    methods
        function diag = getDiagnosticFor(tolerance, actual, expected)
            import matlab.unittest.diagnostics.StringDiagnostic

            if ~isSameSize(actual.Sequence, expected.Sequence)
                str = 'The DNA sequences must be the same length.';
            else
                str = sprintf('%s%d.\n%s%d.', ...
                    'The DNA sequences have a Hamming distance of ', ...
                    hammingDistance(actual.Sequence, expected.Sequence), ...
                    'The allowable distance is ', ...
                    tolerance.Value);
            end
            diag = StringDiagnostic(str);
        end
    end
```

HammingDistance Class Definition Summary

```
classdef HammingDistance < matlab.unittest.constraints.Tolerance
    properties
        Value
    end

    methods
        function tolerance = HammingDistance(value)
            tolerance.Value = value;
        end

        function tf = supports(~, value)
            tf = isa(value, 'DNA');
        end

        function tf = satisfiedBy(tolerance, actual, expected)
            if ~isSameSize(actual.Sequence, expected.Sequence)
                tf = false;
                return
            end
            tf = hammingDistance(actual.Sequence,expected.Sequence) <= tolerance.Value;
        end

        function diag = getDiagnosticFor(tolerance, actual, expected)
```

```
            import matlab.unittest.diagnostics.StringDiagnostic

            if ~isSameSize(actual.Sequence, expected.Sequence)
                str = 'The DNA sequences must be the same length.';
            else
                str = sprintf('%s%d.\n%s%d.', ...
                    'The DNA sequences have a Hamming distance of ', ...
                    hammingDistance(actual.Sequence, expected.Sequence), ...
                    'The allowable distance is ', ...
                    tolerance.Value);
            end
            diag = StringDiagnostic(str);
        end
    end
end

function tf = isSameSize(str1, str2)
tf = isequal(size(str1), size(str2));
end

function distance = hammingDistance(str1, str2)
distance = nnz(str1 ~= str2);
end
```

At the command prompt, create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Create two DNA objects.

```
sampleA = DNA('ACCTGAGTA');
sampleB = DNA('ACCACAGTA');
```

Verify that the DNA sequences are equal to each other.

```
testCase.verifyThat(sampleA, IsEqualTo(sampleB))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
```

```
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".

Actual Object:
      DNA with properties:

        Sequence: 'ACCTGAGTA'
Expected Object:
      DNA with properties:

        Sequence: 'ACCACAGTA'
```

Verify that the DNA sequences are equal to each other within a Hamming distance of 1.

```
testCase.verifyThat(sampleA, IsEqualTo(sampleB,...
    'Within', HammingDistance(1)))

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".
    --> The DNA sequences have a Hamming distance of 2.
        The allowable distance is 1.

Actual Object:
      DNA with properties:

        Sequence: 'ACCTGAGTA'
Expected Object:
      DNA with properties:

        Sequence: 'ACCACAGTA'
```

The sequences are not equal to each other within a tolerance of 1. The testing framework displays additional diagnostics from the `getDiagnosticFor` method.

Verify that the DNA sequences are equal to each other within a Hamming distance of 2.

```
testCase.verifyThat(sampleA, IsEqualTo(sampleB,...
    'Within', HammingDistance(2)))
```

```
Interactive verification passed.
```

getDiagnosticFor

Class: matlab.unittest.constraints.Tolerance
Package: matlab.unittest.constraints

Produce diagnostic for two values specified to be within tolerance

Syntax

```
diag = getDiagnosticFor(tolObj,actVal,expVal)
```

Description

`diag = getDiagnosticFor(tolObj,actVal,expVal)` produces a diagnostic, `diag`, for a value, `actVal`, evaluated against another value, `expVal`, within the tolerance defined by `tolObj`. When creating a custom tolerance, the class author must implement the `getDiagnosticFor` method so that it analyzes the two values, `actVal` and `expVal`, against the tolerance, `tolObj`, and instantiates and returns a `matlab.unittest.diagnostics.Diagnostic` object.

Typically, this diagnostic is used when the `getDiagnosticFor` method of `IsEqualTo` is invoked, and the result is incorporated into the diagnostic output of the `IsEqualTo` constraint.

Input Arguments

`actVal`

Value to determine if is within tolerance of `expVal`

`tolObj`

`Tolerance` instance

`expVal`

Expected value

See Also

`ConstraintDiagnostic` | `Diagnostic` | `satisfiedBy` | `supports`

satisfiedBy

Class: matlab.unittest.constraints.Tolerance
Package: matlab.unittest.constraints

Determine whether two values are within tolerance

Syntax

```
TF = satisfiedBy(tolObj,actVal,expVal)
```

Description

`TF = satisfiedBy(tolObj,actVal,expVal)` determines whether two values, `actVal` and `expVal`, are within the tolerance defined by `tolObj`. The `satisfiedBy` method is used to determine whether the tolerance is met. It returns `true` or `false` (`logical` `0` or `1`). When creating a custom tolerance, a class author uses this method to contain the tolerance definition.

Input Arguments

`actVal`

Value to determine if is within tolerance of `expVal`

`tolObj`

`Tolerance` instance

`expVal`

Expected value

See Also

`getDiagnosticFor` | `supports`

supports

Class: matlab.unittest.constraints.Tolerance
Package: matlab.unittest.constraints

Determine whether tolerance supports specified data type

Syntax

```
TF = supports(tolObj,typeVal)
```

Description

`TF = supports(tolObj,typeVal)` determines whether the tolerance supports a specific data type. It returns `true` or `false` (`logical` `0` or `1`).

The `supports` method provides the ability for a tolerance author to specify support for data types. Generally, the method operates by examining the type of `typeVal` to determine whether it is supported.

Input Arguments

`tolObj`

`Tolerance` instance

`typeVal`

Value used to determine tolerance support

See Also

`getDiagnosticFor` | `satisfiedBy`

contrast

Grayscale colormap for contrast enhancement

Syntax

```
cmap = contrast(X)
cmap = contrast(X,m)
```

Description

The `contrast` function enhances the contrast of an `image`. It creates a new gray colormap, `cmap`, that has an approximately equal intensity distribution. All three elements in each row are identical.

`cmap = contrast(X)` returns a gray colormap that is the same length as the current colormap. If there are `NaN` or `Inf` elements in `X` the length of the colormap increases.

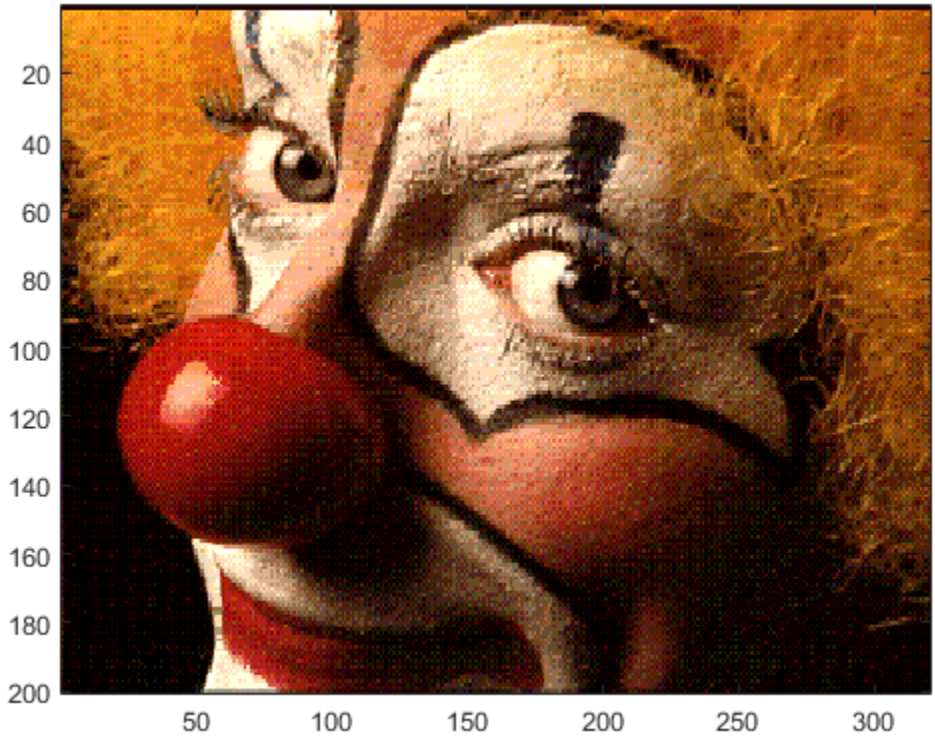
`cmap = contrast(X,m)` returns an `m`-by-3 gray colormap.

Examples

Display Image with Gray Colormap

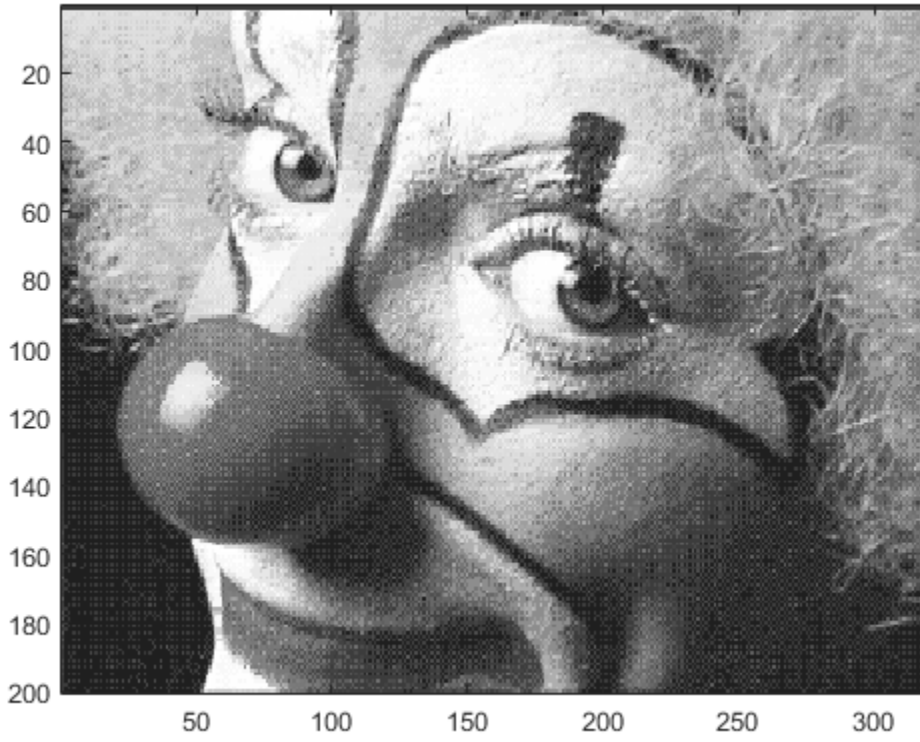
Load `clown` to get image `X` and its associated colormap, `map`. Display the image produced by `X` and `map`.

```
load clown
figure
image(X)
colormap(map)
```



Use `contrast` to return a gray colormap that is the same length as the current colormap, `map`. Display the image with the new colormap, `cmap`.

```
cmap = contrast(X);
colormap(cmap)
```



See Also

`brighten` | `image` | `colormap`

Introduced before R2006a

conv

Convolution and polynomial multiplication

Syntax

```
w = conv(u,v)
w = conv(u,v,shape)
```

Description

`w = conv(u,v)` returns the convolution of vectors `u` and `v`. If `u` and `v` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials.

`w = conv(u,v,shape)` returns a subsection of the convolution, as specified by `shape`. For example, `conv(u,v,'same')` returns only the central part of the convolution, the same size as `u`, and `conv(u,v,'valid')` returns only the part of the convolution computed without the zero-padded edges.

Examples

Polynomial Multiplication via Convolution

Create vectors `u` and `v` containing the coefficients of the polynomials $x^2 + 1$ and $2x + 7$.

```
u = [1 0 1];
v = [2 7];
```

Use convolution to multiply the polynomials.

```
w = conv(u,v)
```

```
w =

     2     7     2     7
```

`w` contains the polynomial coefficients for $2x^3 + 7x^2 + 2x + 7$.

Vector Convolution

Create two vectors and convolve them.

```
u = [1 1 1];
v = [1 1 0 0 0 1 1];
w = conv(u,v)
```

```
w =

     1     2     2     1     0     1     2     2     1
```

The length of `w` is `length(u)+length(v)-1`, which in this example is `9`.

Central Part of Convolution

Create two vectors. Find the central part of the convolution of `u` and `v` that is the same size as `u`.

```
u = [-1 2 3 -2 0 1 2];
v = [2 4 -1 1];
w = conv(u,v,'same')
```

```
w =

    15     5    -9     7     6     7    -1
```

`w` has a length of `7`. The full convolution would be of length `length(u)+length(v)-1`, which in this example would be 10.

Input Arguments

`u,v` — Input vectors

vectors

Input vectors, specified as either row or column vectors. The output vector is the same orientation as the first input argument, `u`. The vectors `u` and `v` can be different lengths or data types.

Data Types: `double` | `single`
Complex Number Support: Yes

`shape` — Subsection of convolution

`'full'` (default) | `'same'` | `'valid'`

Subsection of the convolution, specified as `'full'`, `'same'`, or `'valid'`.

`'full'`	Full convolution (default).
`'same'`	Central part of the convolution of the same size as `u`.
`'valid'`	Only those parts of the convolution that are computed without the zero-padded edges. Using this option, `length(w)` is `max(length(u)-length(v)+1,0)`, except when `length(v)` is zero. If `length(v) = 0`, then `length(w) = length(u)`.

More About

Convolution

The convolution of two vectors, `u` and `v`, represents the area of overlap under the points as `v` slides across `u`. Algebraically, convolution is the same operation as multiplying polynomials whose coefficients are the elements of `u` and `v`.

Let `m = length(u)` and `n = length(v)` . Then `w` is the vector of length `m+n-1` whose `k`th element is

$$w(k) = \sum_j u(j)v(k-j+1).$$

The sum is over all the values of `j` that lead to legal subscripts for `u(j)` and `v(k-j+1)`, specifically `j = max(1,k+1-n):1:min(k,m)`. When `m = n`, this gives

```
w(1) = u(1)*v(1)
w(2) = u(1)*v(2)+u(2)*v(1)
w(3) = u(1)*v(3)+u(2)*v(2)+u(3)*v(1)
```

```
...
w(n) = u(1)*v(n)+u(2)*v(n-1)+ ... +u(n)*v(1)
...
w(2*n-1) = u(n)*v(n)
```

See Also

conv2 | convmtx | convn | deconv | filter | xcorr

Introduced before R2006a

conv2

2-D convolution

Syntax

```
C = conv2(A,B)
C = conv2(h1,h2,A)
C = conv2(...,shape)
```

Description

`C = conv2(A,B)` computes the two-dimensional convolution of matrices `A` and `B`. If one of these matrices describes a two-dimensional finite impulse response (FIR) filter, the other matrix is filtered in two dimensions. The size of `C` is determined as follows: if `[ma,na] = size(A)`, `[mb,nb] = size(B)`, and `[mc,nc] = size(C)`, then `mc = max([ma+mb-1,ma,mb])` and `nc = max([na+nb-1,na,nb])`.

`C = conv2(h1,h2,A)` first convolves each column of `A` with the vector `h1` and then convolves each row of the result with the vector `h2`. The size of `C` is determined as follows: if `n1 = length(h1)` and `n2 = length(h2)`, then `mc = max([ma +n1-1,ma,n1])` and `nc = max([na+n2-1,na,n2])`.

`C = conv2(...,shape)` returns a subsection of the two-dimensional convolution, as specified by the `shape` parameter:

`'full'`	Returns the full two-dimensional convolution (default).
`'same'`	Returns the central part of the convolution of the same size as `A`.
`'valid'`	Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, `size(C) = max([ma-max(0,mb-1),na-max(0,nb-1)],0)`.

Note: All numeric inputs to `conv2` must be of type `double` or `single`.

Examples

Shape for Subsection of 2-D Convolution

For the `'same'` case, `conv2` returns the central part of the convolution. If there are an odd number of rows or columns, the "center" leaves one more at the beginning than the end.

This example first computes the convolution of A using the default (`'full'`) shape, then computes the convolution using the `'same'` shape. Note that the array returned using `'same'` corresponds to the red highlighted elements of the array returned using the default shape.

```
A = rand(3);
B = rand(4);
C = conv2(A,B)   % C is 6-by-6

C =
   0.1838  0.2374  0.9727  1.2644  0.7890  0.3750
   0.6929  1.2019  1.5499  2.1733  1.3325  0.3096
   0.5627  1.5150  2.3576  3.1553  2.5373  1.0602
   0.9986  2.3811  3.4302  3.5128  2.4489  0.8462
   0.3089  1.1419  1.8229  2.1561  1.6364  0.6841
   0.3287  0.9347  1.6464  1.7928  1.2422  0.5423

Cs = conv2(A,B,'same')   % Cs is the same size as A: 3-by-3
Cs =
   2.3576  3.1553  2.5373
   3.4302  3.5128  2.4489
   1.8229  2.1561  1.6364
```

Extract Edges from Raised Pedestal

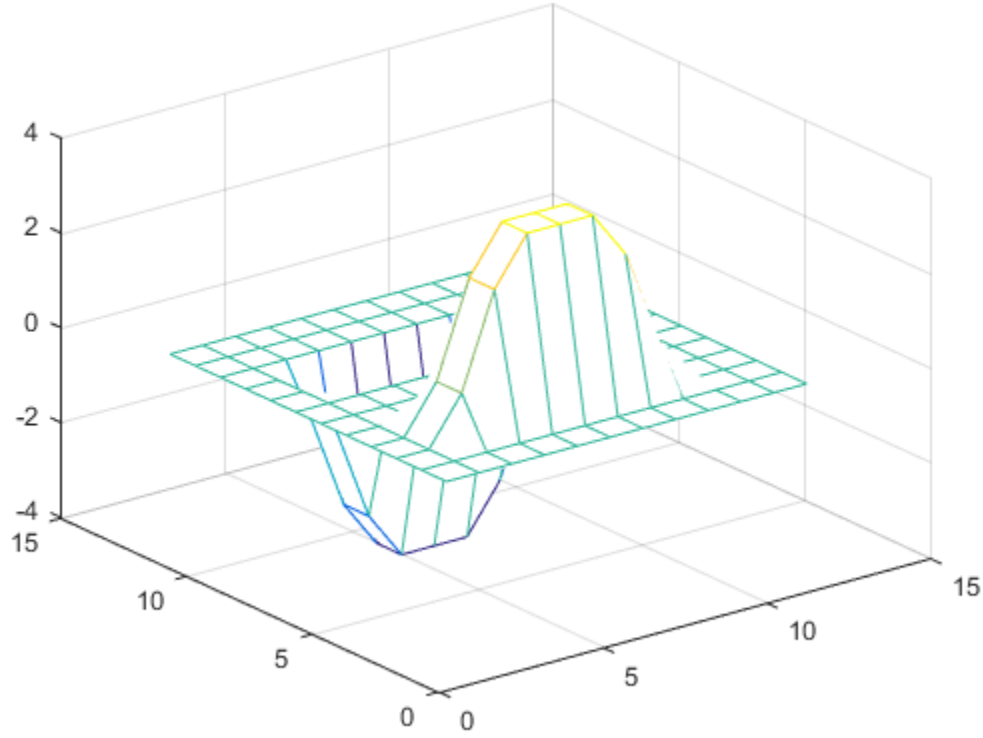
In image processing, the Sobel edge finding operation is a two-dimensional convolution of an input array with the special matrix:

```
s = [1 2 1; 0 0 0; -1 -2 -1];
```

These commands extract the horizontal edges from a raised pedestal.

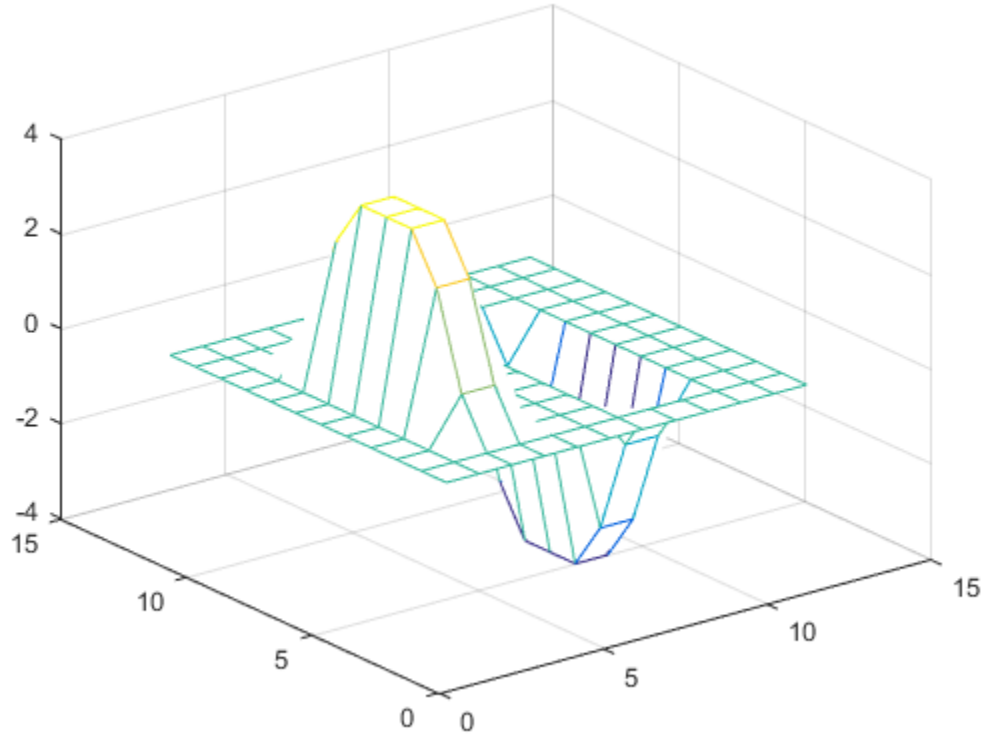
```
A = zeros(10);
```

```
A(3:7,3:7) = ones(5);
H = conv2(A,s);
figure, mesh(H)
```



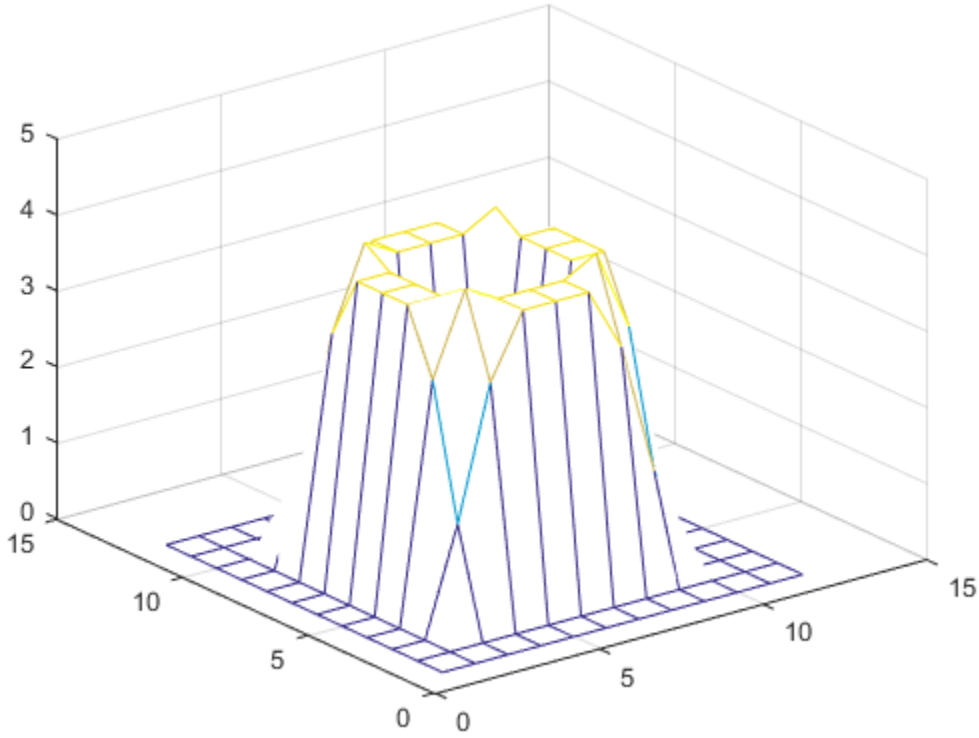
Transposing the filter `s` extracts the vertical edges of `A`.

```
V = conv2(A,s');
figure, mesh(V)
```



This figure combines both horizontal and vertical edges.

```
figure
mesh(sqrt(H.^2 + V.^2))
```



More About

Algorithms

`conv2` uses a straightforward formal implementation of the two-dimensional convolution equation in spatial form. If a and b are functions of two discrete variables, n_1 and n_2, then the formula for the two-dimensional convolution of a and b is

$$c(n_1,n_2) = \sum_{k_1=-\infty}^{\infty}\sum_{k_2=-\infty}^{\infty} a(k_1,k_2)b(n_1-k_1,n_2-k_2)$$

In practice however, `conv2` computes the convolution for finite intervals.

Note that matrix indices in MATLAB software always start at 1 rather than 0. Therefore, matrix elements `A(1,1)`, `B(1,1)`, and `C(1,1)` correspond to mathematical quantities $a(0,0)$, $b(0,0)$, and $c(0,0)$.

See Also

`conv` | `convn` | `filter2` | `xcorr2`

Introduced before R2006a

convhull

Convex hull

Note: Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `convhull`.

Syntax

```
K = convhull(X,Y)
K = convhull(X,Y,Z)
K = convhull(X)
K = convhull(...,'simplify', logicalvar)
[K,V] = convhull(...)
```

Definitions

`convhull` returns the convex hull of a set of points in 2-D or 3-D space.

Description

`K = convhull(X,Y)` returns the 2-D convex hull of the points (`X`,`Y`), where `X` and `Y` are column vectors. The convex hull `K` is expressed in terms of a vector of point indices arranged in a counterclockwise cycle around the hull.

`K = convhull(X,Y,Z)` returns the 3-D convex hull of the points (`X`,`Y`,`Z`), where `X`, `Y`, and `Z` are column vectors. `K` is a triangulation representing the boundary of the convex hull. `K` is of size `mtri`-by-3, where `mtri` is the number of triangular facets. That is, each row of `K` is a triangle defined in terms of the point indices.

`K = convhull(X)` returns the 2-D or 3-D convex hull of the points `X`. This variant supports the definition of points in matrix format. `X` is of size `mpts`-by-`ndim`, where `mpts`

is the number of points and `ndim` is the dimension of the space where the points reside, 2 ≦ `ndim` ≦ 3. The output facets are equivalent to those generated by the 2-input or 3-input calling syntax.

`K = convhull(...,'simplify', logicalvar)` provides the option of removing vertices that do not contribute to the area/volume of the convex hull, the default is false. Setting `'simplify'` to true returns the topology in a more concise form.

`[K,V] = convhull(...)` returns the convex hull `K` and the corresponding area/volume `V` bounded by `K`.

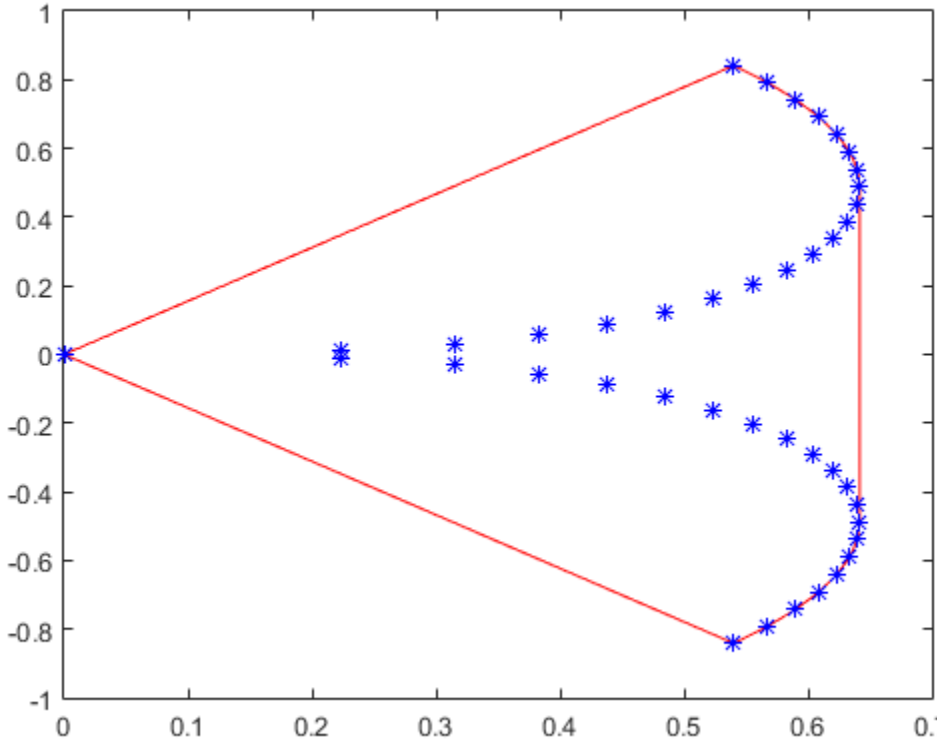
Visualization

Use `plot` to plot the output of `convhull` in 2-D. Use `trisurf` or `trimesh` to plot the output of convhull in 3-D.

Examples

Plot 2-D Convex Hull

```
xx = -1:.05:1;
yy = abs(sqrt(xx));
[x,y] = pol2cart(xx,yy);
k = convhull(x,y);
plot(x(k),y(k),'r-',x,y,'b*')
```



See Also

convexHull | voronoiDiagram | convhulln | delaunay | polyarea | voronoi

Introduced before R2006a

convhulln

N-D convex hull

Syntax

```
K = convhulln(X)
K = convhulln(X, options)
[K, v] = convhulln(...)
```

Description

`K = convhulln(X)` returns the indices `K` of the points in `X` that comprise the facets of the convex hull of `X`. `X` is an `m`-by-`n` array representing `m` points in N-dimensional space. If the convex hull has `p` facets then `K` is `p`-by-`n`.

`convhulln` uses Qhull.

`K = convhulln(X, options)` specifies a cell array of strings `options` to be used as options in Qhull. The default options are:

- `{'Qt'}` for 2-, 3-. and 4-dimensional input
- `{'Qt','Qx'}` for 5-dimensional input and higher.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see http://www.qhull.org/.

`[K, v] = convhulln(...)` also returns the volume `v` of the convex hull.

Visualization

Plotting the output of `convhulln` depends on the value of `n`:

- For `n = 2`, use `plot` as you would for `convhull`.
- For `n = 3`, you can use `trisurf` to plot the output. The calling sequence is

```
K = convhulln(X);
trisurf(K,X(:,1),X(:,2),X(:,3))
```

- You cannot plot `convhulln` output for `n > 3`.

Examples

The following example illustrates the `options` input for `convhulln`. The following commands

```
X = [0 0; 0 1e-10; 0 0; 1 1];
K = convhulln(X)
```

return a warning.

```
Warning: qhull precision warning:
The initial hull is narrow
(cosine of min. angle is 0.9999999999999998).
A coplanar point may lead to a wide facet.
Options 'QbB' (scale to unit box) or 'Qbb'
(scale last coordinate) may remove this warning.
Use 'Pp' to skip this warning.
```

To suppress the warning, use the option `'Pp'`. The following command passes the option `'Pp'`, along with the default `'Qt'`, to `convhulln`.

```
K = convhulln(X,{'Qt','Pp'})

K =

     1     4
     1     2
     4     2
```

More About

Algorithms

`convhulln` is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM *Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483.

See Also

`convexHull` | `convhull` | `delaunayn` | `dsearchn` | `tsearchn` | `voronoin`

Introduced before R2006a

convn

N-D convolution

Syntax

```
C = convn(A,B)
C = convn(A,B,'shape')
```

Description

`C = convn(A,B)` computes the N-dimensional convolution of the arrays `A` and `B`. The size of the result is `size(A)+size(B)-1`.

`C = convn(A,B,'shape')` returns a subsection of the N-dimensional convolution, as specified by the `shape` parameter:

`'full'`	Returns the full N-dimensional convolution (default).
`'same'`	Returns the central part of the result that is the same size as `A`.
`'valid'`	Returns only those parts of the convolution that can be computed without assuming that the array A is zero-padded. The size of the result is `max(size(A)-size(B) + 1, 0)`

See Also

`conv` | `conv2`

Introduced before R2006a

matlab.mixin.CustomDisplay class

Package: matlab.mixin

Display customization interface class

Description

This class provides an interface for customizing the MATLAB display of objects. Derive your class from `matlab.mixin.CustomDisplay` to add the custom display functionality to your class.

`matlab.mixin.CustomDisplay` implements three public sealed methods. `disp` and `display` provide a simple object display. The `details` method provides a standard formal display of object information.

For customizing object display, `matlab.mixin.CustomDisplay` defines a number of protected methods that you can override in your subclass. By overriding specific methods, you can customize specific aspects of the object display.

Note: You cannot use `matlab.mixin.CustomDisplay` to derive a custom display for enumeration classes. For an alternative approach, see "Overload the disp Function"

Methods

convertDimensionsToString	Return array dimensions as string
details	Fully detailed formal object display
disp	Simple informal object display
display	Print variable name and display object
displayEmptyObject	Display for empty object arrays

displayNonScalarObject	Display format for non-scalar objects
displayPropertyGroups	Display titles and property groups as defined
displayScalarHandleToDeletedObject	Display format for deleted scalar handles
displayScalarObject	Display format for scalar objects
getClassNameForHeader	Return class name for display
getDeletedHandleText	Returns text for handle to deleted object display
getDetailedFooter	Returns default detailed footer for object display
getDetailedHeader	Returns default detailed header for object display
getFooter	Build and return display footer text
getHandleText	Return string `'handle'` with link to documentation
getHeader	Build and return display header text
getPropertyGroups	Construct array of property groups
getSimpleHeader	Return simple header for object display

Attributes

`Abstract`	This class defines an interface that subclasses inherit. You cannot instantiate this class.
`HandleCompatibile`	You can use this class to derive both handle and value classes.

See Also

`matlab.mixin.util.PropertyGroup`

Related Examples

- "Custom Display Interface"

Introduced in R2013b

matlab.mixin.CustomDisplay.convertDimensionsToString

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Return array dimensions as string

Syntax

```
dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj)
```

Description

`dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj)` converts a size vector into a properly formatted string of dimensions for the nonscalar header.

Input Arguments

`obj`

MATLAB object

Default:

Output Arguments

`dimstr`

String representing the object's dimensions as determined by calling `size`.

Attributes

`Static`	`true`

`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.getHeader`

Related Examples

- "Custom Display Interface"

details

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Fully detailed formal object display

Syntax

```
details(obj)
```

Description

`details(obj)` displays the detailed state of obj. The detailed display is not affected by customizations made to classes derived from `matlab.mixin.CustomDisplay`.

Attributes

`Access = public`	You can call `details` on any object that is a class derived from `matlab.mixin.CustomDisplay`.
`Sealed`	You cannot override the `details` method in your sublcass of `matlab.mixin.CustomDisplay`

Input Arguments

`obj`

Instance of a class derived from `matlab.mixin.CustomDisplay`.

Default: none

Examples

Suppose `EmployeeInfo` is a subclass of `matlab.mixin.CustomDisplay`

```
details(e1)

  EmployeeInfo handle with properties:

          Name: 'Bill Tork'
      JobTitle: 'Software Engineer'
    Department: 'Product Development'
        Salary: 1000
      Password: 'bill123'

  Methods, Events, Superclasses
```

See Also

disp | display

disp

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Simple informal object display

Syntax

```
disp(obj)
```

Description

`disp(obj)` displays the contents of obj in one of four simple formats, based on the state of obj.

- Handle to deleted object
- Empty object array
- Scalar
- Nonscalar

To customize a display format, override one or more of the four handler methods.

Input Arguments

`obj`

Object of a class derived from `matlab.mixin.CustomDisplay`

Attributes

`Access`	`public`
`Sealed`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.displayScalarObject` | `matlab.mixin.CustomDisplay.displayNonScalarObject` | `matlab.mixin.CustomDisplay.displayEmptyObject` | `matlab.mixin.CustomDisplay.displayScalarHandleToDeletedObject` | `details` | `disp`

Related Examples

- “Custom Display Interface”

display

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Print variable name and display object

Syntax

```
display(obj)
```

Description

`display(obj)` prints the name of input argument obj as it appears in the workspace of the caller. If there is no assignment to a variable, `display` uses, `ans`. `display` then calls the `disp` method to print the object.

Input Arguments

`obj`

Object array of a class derived from `matlab.mixin.CustomDisplay`

Attributes

`Access`	`public`
`Sealed`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`details` | `disp`

Related Examples

- “Custom Display Interface”

displayEmptyObject

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Display for empty object arrays

Syntax

```
displayEmptyObject(obj)
```

Description

`displayEmptyObject(obj)` is called by `disp` when the object, obj, is empty. An object array is empty if one or more of its dimensions are zero. An empty object array is never scalar.

The default display of an empty object consists of a header and a list of property names. The header consists of the object's dimensions and the properties are shown in the order defined in the class definition. `displayEmptyObject` shows only those properties with public `GetAccess` and `Hidden` set to `false`.

Override this method to customize the appearance of an empty object array.

Input Arguments

`obj`

Object of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.displayScalarObject` |
`matlab.mixin.CustomDisplay.displayNonScalarObject` |
`matlab.mixin.CustomDisplay.displayScalarHandleToDeletedObject`

Related Examples

- “Custom Display Interface”

displayNonScalarObject

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Display format for non-scalar objects

Syntax

```
displayNonScalarObject(obj)
```

Description

`displayNonScalarObject(obj)` is called by the `disp` method when the object, obj, is nonscalar (`prod(size(obj)) > 1`)

The default display of a nonscalar object array consists of a header and a list of property names. The header consists of the object's dimensions and the properties are shown in the order defined in the class definition. `displayNonScalarObject` shows only those properties with public `GetAccess` and `Hidden` set to `false`.

Override this method to customize the display a nonscalar object array.

Input Arguments

`obj`

Object array of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.displayScalarObject` |
`matlab.mixin.CustomDisplay.displayEmptyObject` |
`matlab.mixin.CustomDisplay.displayScalarHandleToDeletedObject`

Related Examples

- “Custom Display Interface”

matlab.mixin.CustomDisplay.displayPropertyGroups

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Display titles and property groups as defined

Syntax

```
matlab.mixin.CustomDisplay.displayPropertyGroups(obj,
propertyGroupArray)
```

Description

`matlab.mixin.CustomDisplay.displayPropertyGroups(obj, propertyGroupArray)` displays titles and custom property lists as defined by the property groups.

Input Arguments

`obj`

MATLAB object

Default:

`propertyGroupArray`

Array of `matlab.mixin.util.PropertyGroup` objects.

Default:

Attributes

`Static`	`true`

`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`PropertyGroup`

Related Examples

- “Custom Display Interface”

displayScalarHandleToDeletedObject

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Display format for deleted scalar handles

Syntax

```
displayScalarHandleToDeletedObject(obj)
```

Description

`displayScalarHandleToDeletedObject(obj)` is called by the `disp` method when `obj` is:

- An instance of a handle class
- Scalar
- A handle to a deleted object

That is, the following expression is `true`.

```
isa(obj,'handle') && isscalar(obj) && ~isvalid(obj)
```

Override this method to customize the appearance of your object's display when it is deleted.

Input Arguments

`obj`

Object of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

Access	protected

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.displayScalarObject` |
`matlab.mixin.CustomDisplay.displayNonScalarObject` |
`matlab.mixin.CustomDisplay.displayEmptyObject`

Related Examples

- “Custom Display Interface”

displayScalarObject

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Display format for scalar objects

Syntax

```
displayScalarObject(obj)
```

Description

`displayScalarObject(obj)` is called by the `disp` method when the object, obj, is scalar (`prod(size(obj)) == 1`).

The default display of a scalar object consists of a header and a list of properties and their values. Properties are shown in the order they are defined in the class definition. `displayScalarObject` shows only those properties with public `GetAccess` and `Hidden` set to `false`.

Override this method to customize the display of a scalar object.

Input Arguments

`obj`

Object of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.displayNonScalarObject | matlab.mixin.CustomDisplay.displayEmptyObject | matlab.mixin.CustomDisplay.displayScalarHandleToDeletedObject`

Related Examples

- "Custom Display Interface"

matlab.mixin.CustomDisplay.getClassNameForHeader

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Return class name for display

Syntax

```
name = matlab.mixin.CustomDisplay.getClassNameForHeader(obj)
```

Description

`name = matlab.mixin.CustomDisplay.getClassNameForHeader(obj)` returns the class name of obj. If the display supports hypertext links, the text is linked to the help for the class of `obj`.

Use this method when building a custom display that includes the class name, but differs from the default header.

Input Arguments

`obj`

MATLAB object

Default:

Output Arguments

`name`

The simple class name, linked to the help for the class if the display supports hypertext links

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`getHeader`

Related Examples

- “Custom Display Interface”

matlab.mixin.CustomDisplay.getDeletedHandleText

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Returns text for handle to deleted object display

Syntax

```
handleText = matlab.mixin.CustomDisplay.getDeletedHandleText
```

Description

`handleText = matlab.mixin.CustomDisplay.getDeletedHandleText` returns the text:

```
'handle to deleted'
```

The text is linked to the documentation on deleted handle objects.

Output Arguments

`handleText`

String `'handle to deleted'`, linked if the display supports hypertext links

Attributes

Static	true
Access	Protected
Sealed	true
Hidden	true

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`getHeader`

Related Examples

- “Custom Display Interface”

matlab.mixin.CustomDisplay.getDetailedFooter

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Returns default detailed footer for object display

Syntax

```
headerText = matlab.mixin.CustomDisplay.getDetailedFooter(obj)
```

Description

`headerText = matlab.mixin.CustomDisplay.getDetailedFooter(obj)` returns the string containing:

```
Methods, Events, Superclass
```

Each link executes the respective command on `obj`.

Input Arguments

`obj`

MATLAB object

Default:

Output Arguments

`headerText`

String containing the linked phrase `'Methods, Events, Superclasses'` or the empty string if the display does not support hypertext links

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`events` | `getFooter` | `methods` | `superclasses`

Related Examples

- “Custom Display Interface”

matlab.mixin.CustomDisplay.getDetailedHeader

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Returns default detailed header for object display

Syntax

```
header = matlab.mixin.CustomDisplay.getDetailedHeader(obj)
```

Description

`header = matlab.mixin.CustomDisplay.getDetailedHeader(obj)` returns a string containing:

- Linked class name of `obj`
- Link to handle documentation if `obj` is a handle class
- The string `'with properties:'`

Input Arguments

`obj`

MATLAB object

Default:

Output Arguments

`header`

String containing the full detailed header, with properly inserted links if the display supports hypertext linking

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`details` | `handle`

Related Examples

- “Custom Display Interface”

getFooter

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Build and return display footer text

Syntax

```
s = getFooter(obj)
```

Description

`s = getFooter(obj)` returns the text used as the footer when displaying obj. This method is called once for the entire object array.

Override this method to create a custom footer. The overriding implementation must support all states of the object, including scalar, nonscalar, empty, and deleted (if obj is an instance of a handle class).

Input Arguments

`obj`

Object array of a class derived from `matlab.mixin.CustomDisplay`

Default:

Output Arguments

`s`

Footer text, returned as a string.

The default implementation returns an empty string

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.getHeader` |
`matlab.mixin.CustomDisplay.getPropertyGroups`

Related Examples

- "Custom Display Interface"

matlab.mixin.CustomDisplay.getHandleText

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Return string `'handle'` with link to documentation

Syntax

```
handleText = matlab.mixin.CustomDisplay.getHandleText
```

Description

`handleText = matlab.mixin.CustomDisplay.getHandleText` returns the string `'handle'`. If the display supports hypertext linking, the text is linked to documentation describing handle classes.

Output Arguments

`handleText`

String `'handle'`, linked to the handle documentation.

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`getHeader`

Related Examples

- “Custom Display Interface”

getHeader

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Build and return display header text

Syntax

```
s = getHeader(obj)
```

Description

`s = getHeader(obj)` returns the text used as the header when displaying obj. This method is called once for the entire object array.

Override this method to create a custom header. The overriding implementation must support all states of the object, including scalar, nonscalar, empty, and deleted (if obj is an instance of a handle class).

Input Arguments

`obj`

Object array of a class derived from `matlab.mixin.CustomDisplay`

Output Arguments

`s`

Header string, returned as a `char` array

The default implementation returns the following:

- If obj is scalar, returns *`classname`*, which is the simple name of the class (the nonpackage-qualified name).
- If obj is nonscalar, returns *`classname`* and dimensions.

- If obj is empty, returns an empty string.
- If obj is a deleted handle, returns the string `deleted` *`classname`* `handle`

`classname` is linked to MATLAB documentation for the class. Selecting the link displays the `helpPopup` window.

If you override this method, you might need to terminate s with a newline (`\n`) character.

Examples

Append Text to Default Header

Append the string, `'with Customized Display'`, to the header text.

Write a `getHeader` method.

```
methods (Access = protected)
   function header = getHeader(obj)
      if ~isscalar(obj)
         header = getHeader@matlab.mixin.CustomDisplay(obj);
      else
         headerStr = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
         headerStr = [headerStr,' with Customized Display'];
         header = sprintf('%s\n',headerStr);
      end
   end
end
```

Add `getHeader` method to class definition.

- “Custom Display Interface”

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.getFooter` |
`matlab.mixin.CustomDisplay.getPropertyGroups`

getPropertyGroups

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Construct array of property groups

Syntax

```
groups = getPropertyGroups(obj)
```

Description

`groups = getPropertyGroups(obj)` returns an array of `matlab.mixin.util.PropertyGroup` objects. MATLAB displays property groups separated by blank spaces.

Each default display state handler method calls this method once. The default implementation returns the properties in one group. These properties must have public `GetAccess` and not be defined as `Hidden`. If the object is scalar, MATLAB includes dynamic properties.

Override this method to construct one or more customized groups of properties to display.

Each group object array has the following fields:

- `Title` — String used as the header for the property group or an empty string if no title is used.
- `PropertyList` — The property list can be either:
  - A 1-by-1 `struct` of property name-property value pairs
  - A cell array of string property names.

Use the `struct` of name-value pairs if the object is scalar and you want to assign custom property values. Otherwise, use a cell array of string property names. If the object is scalar MATLAB adds the property values retrieved from the object.

Input Arguments

obj

Object array of a class derived from `matlab.mixin.CustomDisplay`

Default:

Output Arguments

groups

1xN array of `matlab.mixin.util.PropertyGroup` objects, where N is the number of groups

Examples

Custom Property Group

Customize the values returned by some properties.

Write a `getPropertyGroups` method.

```
methods (Access = protected)
   function propgrp = getPropertyGroups(obj)
      if ~isscalar(obj)
         propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
      else
         pd(1:length(obj.Password)) = '*';
         propList = struct('Department',obj.Department,...
            'JobTitle',obj.JobTitle,...
            'Name',obj.Name,...
            'Salary','Not avalable',...
            'Password',pd);
         propgrp = matlab.mixin.util.PropertyGroup(propList);
      end
   end
end
```

Add function to class definition.

- “Custom Display Interface”
- “Customize Property Display”

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.util.PropertyGroup` |
`matlab.mixin.CustomDisplay.getHeader` |
`matlab.mixin.CustomDisplay.getFooter`

matlab.mixin.CustomDisplay.getSimpleHeader

Class: matlab.mixin.CustomDisplay
Package: matlab.mixin

Return simple header for object display

Syntax

```
header = matlab.mixin.CustomDisplay.getSimpleHeader(obj)
```

Description

`header = matlab.mixin.CustomDisplay.getSimpleHeader(obj)` returns the default simple header for obj.

Input Arguments

`obj`

MATLAB object.

Default:

Output Arguments

`header`

String containing the linked class name and the phrase `'with properties'`

Attributes

`Static`	`true`

`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.CustomDisplay.getHeader`

Related Examples

- "Custom Display Interface"

matlab.mixin.Copyable class

Package: matlab.mixin

Superclass providing copy functionality for handle objects

Description

The `matlab.mixin.Copyable` class is an abstract handle class that provides a `copy` method for copying handle objects. The `copy` method makes a shallow copy of the object (that is, it shallow copies all non-dependent properties from the source to the destination object). In making a shallow copy, MATLAB does not call `copy` recursively on any handles contained in property values.

Subclass `matlab.mixin.Copyable` when you want to define handles classes that inherit a `copy` method. The `copy` method:

- Copies data without calling class constructors or property set functions and therefore produces no side effects.
- Enables subclasses to customize the copy behavior

Customizing Subclass Copy Behavior

The `copy` method provides the public, non-overrideable interface to copy behavior. `copy` takes an array of objects as input and returns an array of the same shape and dimensions.

`copyElement` is a protected method that the `copy` method uses to perform the copy operation on each object in the input array. `copyElement` is not `Sealed` so you can override it in your subclass to customize the behavior of the inherited `copy` method.

Implementing a Selective Deep Copy

This example overrides the `copyElement` method in a subclass of `matlab.mixin.Copyable` to implement a deep copy of a specific class of `handle` objects.

Consider the following classes:

- `ContainsHandles` — subclass of `matlab.mixin.Copyable` that contains `handle` objects in two properties
- `DeepCp` — subclass of `matlab.mixin.Copyable`
- `ShallowCp` — subclass of `handle`

Here are the simplified class definitions:

```
classdef ContainsHandles < matlab.mixin.Copyable
   properties
      Prop1
      Prop2
      DeepObj     % Contains a DeepCp object
      ShallowObj  % Contains a ShallowCp object
   end
   methods
      function obj = ContainsHandles(val1,val2,deepobj,shallowobj)
         if nargin > 0
            obj.Prop1 = val1;
            obj.Prop2 = val2;
            obj.DeepObj = deepobj;
            obj.ShallowObj = shallowobj;
          end
      end
   end
      methods(Access = protected)
       % Override copyElement method:
      function cpObj = copyElement(obj)
         % Make a shallow copy of all four properties
         cpObj = copyElement@matlab.mixin.Copyable(obj);
         % Make a deep copy of the DeepCp object
         cpObj.DeepObj = copy(obj.DeepObj);
      end
   end
end
```

The `DeepCp` class derives from `matlab.mixin.Copyable`:

```
classdef DeepCp < matlab.mixin.Copyable
   properties
      DpProp
   end
   methods
      function obj = DeepCp(val)
      ...
```

```
      end
   end
end
```

The `handle` class `ShallowCp` does not derive from `matlab.mixin.Copyable` and, therefore, has no copy method:

```
classdef ShallowCp < handle
   properties
      ShProp
   end
   methods
      function obj = ShallowCp(val)
       ...
      end
   end
end
```

Create a `ContainsHandles` object, which contains the two handle objects in its `DpProp` and `ShProp` properties:

```
>> sc = ShallowCp(7);
>> dc = DeepCp(7);
>> a = ContainsHandles(4,5,dc,sc);
>> a.DeepObj
ans =

  DeepCp with properties:

    DpProp: 7
>> a.ShallowObj.ShProp
ans =

  ShallowCp with properties:

    ShProp: 7
```

Make a copy of the `ContainsHandles` object:

```
>> b = copy(a);
```

The returned copy `b` contains a shallow copy of object `sc`, and a deep copy of object `dc`. That is, the `dc` object passed to `ContainsHandles` constructor is now a new, independent objects as a result of the copy operation. You can now change the `dc` object without affecting the copy. This is not the case for the shallow copied object, `sc`:

```
% Change the property values of the handle objects:
>> sc.ShProp = 5;
>> dc.DpProp = 5;
% Note that the deep copied object is not affected:
>> b.DeepObj

ans =

  DeepCp with properties:

    DpProp: 7
% The shallow copied object is still referencing the same data:
>> b.ShallowObj

ans =

  ShallowCp with properties:

    ShProp: 5
```

Overriding Copy Behavior in Hierarchies

The `copyElement` method in a superclass cannot access the private data in a subclass.

If you override `copyElement` in a subclass of `matlab.mixin.Copyable`, and then use this subclass as a superclass, you need to override `copyElement` in all subclasses that contain private properties. The override of `copyElement` in subclasses should call the `copyElement` in the respective superclass, as in the previous example.

The following simplified code demonstrates this approach:

```
classdef SuperClass < matlab.mixin.Copyable
   properties(Access = private)
      super_prop
   end
   methods
        ...

      function cpObj = copyElement(obj)
            ...
         cpObj = copyElement@mixin.matlab.Copyable(obj);
            ...
      end
   end
```

```
end


classdef SubClass1 < SuperClass
   properties(Access=private)
      sub_prop1
   end
   methods
      function cpObj = copyElement(obj)
         % Copy super_prop
         cpObj = copyElement@SuperClass(obj);
         % Copy sub_prop1 in subclass
         % Assignment can introduce side effects
         cpObj.sub_prop1 = obj.sub_prop1;
      end
   end
end
```

The override of `copyElement` in `SubClass1` copies the private subclass property because the superclass cannot access private data in the subclass.

Note: The assignment of `sub_prop1` in the override of `copyElement` in `SubClass1` calls the property set method, if one exists, possibly introducing side effects to the copy operation.

Copy Behaviors for Specific Inputs

Given a call to the `matlab.mixin.Copyable` `copy` method of the form:

```
B = copy(A);
```

Under the following conditions, produces the described results:

- `A` has dynamic properties — `copy` does not copy dynamic properties. You can implement dynamic-property copying in the subclass if needed.
- `A` has no non-Dependent properties — `copy` creates a new object with no property values without calling the class constructor to avoid introducing side effects.
- `A` contains deleted handles — `copy` creates deleted handles of the same class in the output array.
- `A` has attached listeners — `copy` does not copy listeners.

- A contains objects of enumeration classes — Enumeration classes cannot subclass `matlab.mixin.Copyable`.
- A `delete` method calls `copy` — `copy` creates a legitimate copy, obeying all the behaviors that apply in any other usage.

Note: You cannot derive an enumeration class from `matlab.mixin.Copyable` because the number of instances you can create are limited to the ones defined inside the enumeration block. See "Working with Enumerations" for more information about enumeration classes.

Methods

copy	Copy array of handle objects

Definitions

Deep Copy

Copy each property value and assign it to the new (copied) property. Recursively copy property values that reference handle objects to copy all of the underlying data.

Shallow Copy

Copy each property value and assign it to the new (copied) property. If a property value is a handle, copy the handle but not the underlying data.

Attributes

`ConstructOnLoad`	`true`

To learn about attributes of classes, see Class Attributes in the MATLAB Object-Oriented Programming documentation.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`handle`

How To

- Class Attributes

Introduced in R2011a

copy

Class: matlab.mixin.Copyable
Package: matlab.mixin

Copy array of handle objects

Syntax

```
B = copy(A)
```

Description

`B = copy(A)` copies each element in the array of handles `A` to the new array of handles `B`.

The `copy` method performs a copy according to the following rules:

- The `copy` method does not copy `Dependent` properties
- MATLAB does not call the `copy` method recursively on any handles contained in property values
- MATLAB does not call the class constructor or property set methods during the copy operation.
- `B` has the same number of elements and same size as `A`.
- `B` is the same class as `A`.
- If `A` is empty, `B` is also empty.
- If `A` is heterogeneous, `B` is also heterogeneous.
- If `A` contains deleted handle objects, copy creates deleted handles of the same class in `B`.
- Dynamic properties and listeners associated with objects in `A` are not copied to objects in `B`.
- You can call `copy` inside your class `delete` method.

Input Arguments

A

Handle object array

Default:

Output Arguments

B

Handle object array containing copies of the objects in A.

Attributes

Sealed	true

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.Copyable` | `handle` | `copyElement`

copyfile

Copy file or folder

Syntax

```
copyfile('source','destination')
copyfile('source','destination','f')
status = copyfile( ___ )
[status,message] = copyfile( ___ )
[status,message,messageid] = copyfile( ___ )
```

Description

`copyfile('source','destination')` copies the file or folder named `source` to the file or folder `destination`. The values for `source` and `destination` are 1 x n strings. Use full path names or path names relative to the current folder. To copy multiple files or folders, use one or more wildcard characters (`*`) after the last file separator in `source`. You cannot use a wildcard character in `destination`.

`copyfile('source','destination','f')` copies `source` to `destination`, even when `destination` is not writable. The state of the read-write attribute for `destination` does not change. You can use `f` with any syntax for `copyfile`.

`status = copyfile( ___ )` reports the outcome as a logical scalar, `status`. The value is `1` for success and `0` for failure.

`[status,message] = copyfile( ___ )` returns any warning or error message as a string to `message`. When `copyfile` succeeds, `message` is an empty string.

`[status,message,messageid] = copyfile( ___ )` returns any warning or error identifier as a string to `messageId`. When `copyfile` succeeds, `messageId` is an empty string.

Examples

Copy File to Another Folder

Copy `myFun.m` from the current folder to `d:/work/Projects/`.

```
copyfile('myFun.m','d:/work/Projects/')
```

Copy File to Its Current Folder

Copy `myFun.m` in the current folder, assigning it the name `myFun2.m`.

```
copyfile('myFun.m','myFun2.m')
```

Copy Files and Folders to a New Folder Using Wildcards

Copy files and subfolders whose names begin with `my`, from the `Projects` subfolder within the current folder to the folder `newProjects`, which is at the same level as the current folder:

```
copyfile('Projects/my*','../newProjects/')
```

Copy Files to a New, Nonexistent Folder

Copy the contents of the `Projects` subfolder within the current folder to the `I:/work/newProjects` folder, where `newProjects` does not exist.

```
copyfile('Projects','I:/work/newProjects')
```

Copy File that Overwrites a Read-Only File

Copy the contents of `myFun.m` from the current folder to `d:/work/restricted/myFun2.m`, where `myFun2.m` is read-only.

```
[status,message,messageId] = copyfile('myFun.m', ...
'd:/work/restricted/myFun2.m','f')
```

```
status =
     1

message =
     ''

messageId =
```

```
' '
```

The `status` of 1 and empty `message` and `messageId` strings confirm the copy was successful.

More About

Tips

- The timestamp for `destination` is the same as the timestamp for `source`.
- When `source` is a folder, `destination` must be a folder.
  - When `source` is a folder and `destination` does not exist, `copyfile` creates `destination` and copies the contents of `source` into `destination`.
  - When `source` is a folder and `destination` is an existing folder, `copyfile` copies the contents of `source` into `destination`.
  - When `source` is multiple files and `destination` does not exist, `copyfile` creates `destination`.
- "Specify File Names"
- "Manage Files and Folders"

See Also

`cd` | `delete` | `dir` | `fileattrib` | `filebrowser` | `fileparts` | `mkdir` | `movefile` | `rmdir`

Introduced before R2006a

copyobj

Copy graphics objects and their descendants

Syntax

```
new_handle = copyobj(h,p)
copyobj(___,'legacy')
```

Description

`copyobj` creates copies of graphics objects and assigns the objects to the new parent.

The new parent must be appropriate for the copied object (for example, you can copy an axes only to figure or uipanel). `copyobj` copies children as well.

`new_handle = copyobj(h,p)` copies one or more graphics objects identified by `h` and returns the handle of the new object or an array of new objects. The new graphics objects are children of the graphics objects specified by `p`.

`copyobj(___,'legacy')` copies object callback properties and object application data. This behavior is consistent with versions of `copyobj` before MATLAB release R2014b.

What is Not Copied

`copyobj` does not copy properties or objects that depend on their original context to operate properly. Objects with default context menus (such as legends and colorbars) create new context menus for the new object. Figures create new toolbars and menus for the new figure.

`copyobj` does *not* copy:

- Callback properties (except when using the `legacy` option)
- Application data associated with the object (except when using the `legacy` option)
- Context menu of legends, colorbars, or other objects that define default context menus.

- Default Figure toolbar and menus
- You cannot copy the same object more than once to the same parent in a single call to `copyobj`.

MATLAB changes the `Parent` property to the new parent and assigns the new objects a new handle.

Examples

Copy a surface to a new axes that is in a different figure.

```
h = surf(peaks);
colormap hsv
```

Create the destination figure and axes:

```
fig = figure;
ax = axes;
```

Copy the surface to the new axes and set properties that are not surface properties:

```
new_handle = copyobj(h,ax);
colormap(fig,hsv)
view(ax,3)
grid(ax,'on')
```

Note that while the surface is copied, the `colormap`, `view`, and `grid` are not copied.

More About

Tips

`h` and `p` can be scalars or vectors. When both are vectors, they must be the same length, and the output argument, `new_handle`, is a vector of the same length. In this case, `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p(i)`.

When `h` is a scalar and `p` is a vector, `h` is copied once to each of the parents in `p`. Each `new_handle(i)` is a copy of `h` with its `Parent` property set to `p(i)`, and `length(new_handle)` equals `length(p)`.

When `h` is a vector and `p` is a scalar, each `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p`. The length of `new_handle` equals `length(h)`.

Note: You must copy the associated axes when copying a legend or a colorbar.

When programming a UI, do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the recursion limit.

See Also

`findobj` | `gcf` | `gca` | `gco` | `get` | `set`

Introduced before R2006a

corrcoef

Correlation coefficients

Syntax

```
R = corrcoef(X)
R = corrcoef(x,y)
[R,P]=corrcoef(...)
[R,P,RLO,RUP]=corrcoef(...)
[...]=corrcoef(...,'param1',val1,'param2',val2,...)
```

Description

`R = corrcoef(X)` returns a matrix `R` of correlation coefficients calculated from an input matrix `X` whose rows are observations and whose columns are variables. The matrix `R = corrcoef(X)` is related to the covariance matrix `C = cov(X)` by

$$R(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}}.$$

`corrcoef(X)` is the zeroth lag of the normalized covariance function, that is, the zeroth lag of `xcov(x,'coeff')` packed into a square array.

`R = corrcoef(x,y)` where `x` and `y` are column vectors is the same as `corrcoef([x y])`. If `x` and `y` are not column vectors, `corrcoef` converts them to column vectors. For example, in this case `R=corrcoef(x,y)` is equivalent to `R=corrcoef([x(:) y(:)])`.

`[R,P]=corrcoef(...)` also returns `P`, a matrix of p-values for testing the hypothesis of no correlation. Each p-value is the probability of getting a correlation as large as the observed value by random chance, when the true correlation is zero. If `P(i,j)` is small, say less than `0.05`, then the correlation `R(i,j)` is significant.

`[R,P,RLO,RUP]=corrcoef(...)` also returns matrices `RLO` and `RUP`, of the same size as `R`, containing lower and upper bounds for a 95% confidence interval for each coefficient.

`[...]=corrcoef(...,'param1',val1,'param2',val2,...)` specifies additional parameters and their values. Valid parameters are the following.

`'alpha'`	A number between 0 and 1 to specify a confidence level of 100*(1 – `alpha`)%. Default is 0.05 for 95% confidence intervals.
`'rows'`	Either `'all'` (default) to use all rows, `'complete'` to use rows with no `NaN` values, or `'pairwise'` to compute `R(i,j)` using rows with no `NaN` values in either column `i` or `j`.

The p-value is computed by transforming the correlation to create a t statistic having `n`-2 degrees of freedom, where `n` is the number of rows of `X`. The confidence bounds are based on an asymptotic normal distribution of `0.5*log((1+R)/(1-R))`, with an approximate variance equal to `1/(n-3)`. These bounds are accurate for large samples when `X` has a multivariate normal distribution. The `'pairwise'` option can produce an `R` matrix that is not positive definite.

Examples

Generate random data having correlation between column 4 and the other columns.

```
x = randn(30,4);     % Uncorrelated data
x(:,4) = sum(x,2);   % Introduce correlation.
[r,p] = corrcoef(x)  % Compute sample correlation and p-values.
[i,j] = find(p<0.05);  % Find significant correlations.
[i,j]                % Display their (row,col) indices.

r =
    1.0000   -0.3566    0.1929    0.3457
   -0.3566    1.0000   -0.1429    0.4461
    0.1929   -0.1429    1.0000    0.5183
    0.3457    0.4461    0.5183    1.0000

p =
    1.0000    0.0531    0.3072    0.0613
    0.0531    1.0000    0.4511    0.0135
    0.3072    0.4511    1.0000    0.0033
    0.0613    0.0135    0.0033    1.0000

ans =
     4     2
```

```
     4     3
     2     4
     3     4
```

See Also

cov | mean | median | std | var | xcorr | xcov

cos

Cosine of argument in radians

Syntax

```
Y = cos(X)
```

Description

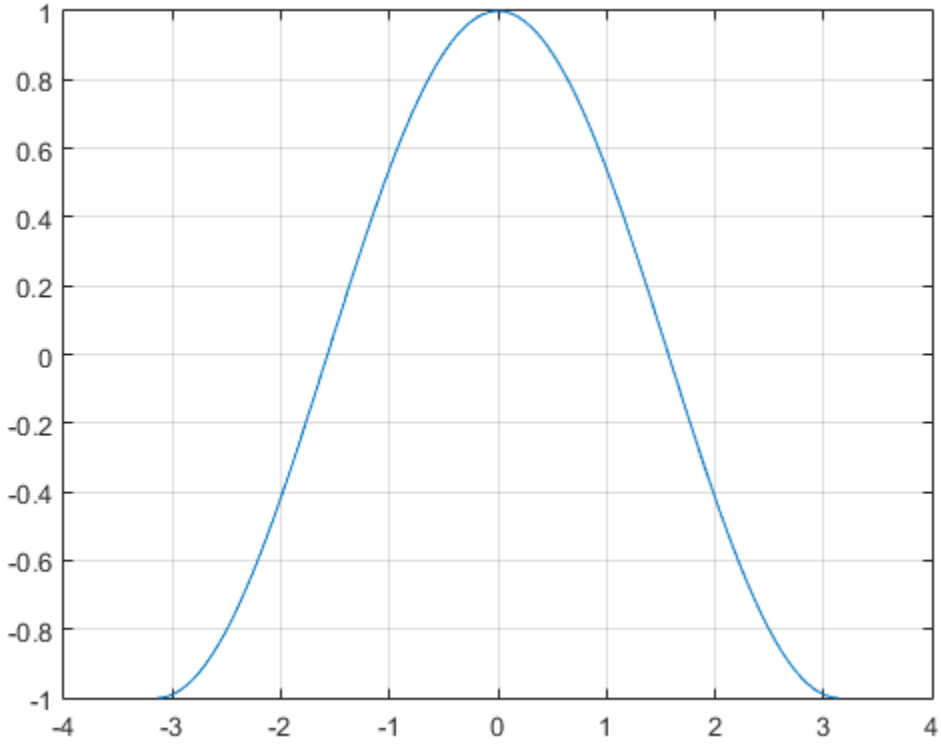
`Y = cos(X)` returns the cosine for each element of `X`. The `cos` function operates element-wise on arrays. The function accepts both real and complex inputs. For purely real values or imaginary values of X, `cos` returns real values in the interval [-1 ,1]. For complex values of X, `cos` returns complex values. All angles are in radians.

Examples

Plot Cosine Function

Plot the cosine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,cos(x))
grid on
```



Cosine of Vector of Complex Angles

Calculate the cosine of the complex angles in vector `x`.

```
x = [-i pi+i*pi/2 -1+i*4];
y = cos(x)
```

```
y =

   1.5431 + 0.0000i  -2.5092 - 0.0000i  14.7547 +22.9637i
```

Input Arguments

X — Input angle in radians

number | vector | matrix | multidimensional array

Input angle in radians, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosine of input angle

scalar value | vector | matrix | N-D array

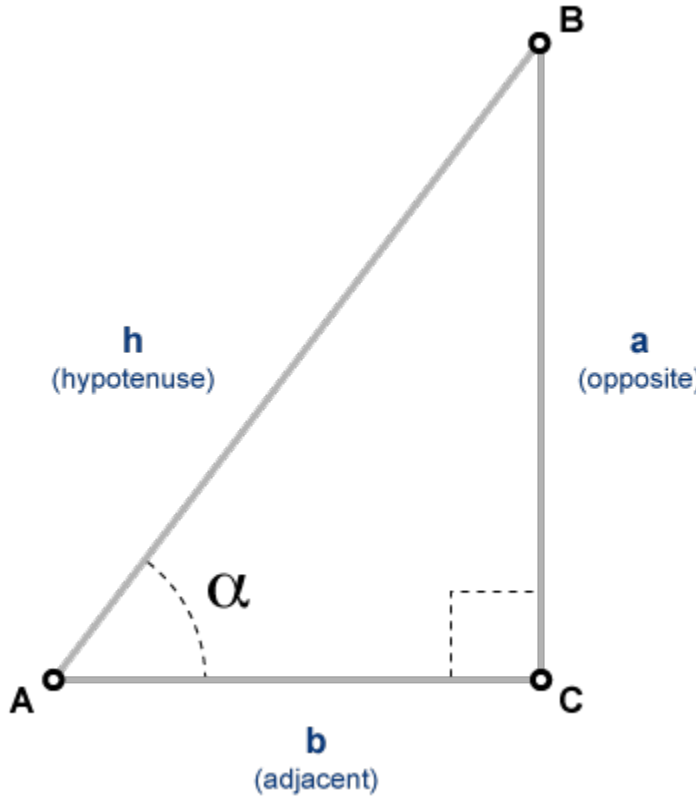
Cosine of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or N-D array.

More About

Cosine Function

The cosine of an angle, α, defined with reference to a right angled triangle is

$$\cos(\alpha) = \frac{\text{adjacent side}}{\text{hypotenuse}} = \frac{b}{h}.$$



The cosine of a complex angle, α, is

$$\cos(\alpha) = \frac{e^{i\alpha} + e^{-i\alpha}}{2}.$$

See Also

acos | acosd | cosd | cosh

Introduced before R2006a

cosd

Cosine of argument in degrees

Syntax

```
Y = cosd(X)
```

Description

`Y = cosd(X)` returns the cosine of the elements of `X`, which are expressed in degrees.

Examples

Cosine of 90 degrees compared to cosine of π/2 radians

```
cosd(90)
```

```
ans =

     0
```

```
cos(pi/2)
```

```
ans =

   6.1232e-17
```

Cosine of complex angles specified in degrees

Create an array of three complex angles and compute the cosine.

```
z = [180+i 45+2i 10+3i];
y = cosd(z)
```

```
y =
```

```
   -1.0002         0.7075 - 0.0247i   0.9862 - 0.0091i
```

Input Arguments

X — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cosd` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosine of angle

scalar value | vector | matrix | N-D array

Cosine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

See Also

`acos` | `acosd` | `cos`

Introduced before R2006a

cosh

Hyperbolic cosine

Syntax

```
Y = cosh(X)
```

Description

The `cosh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

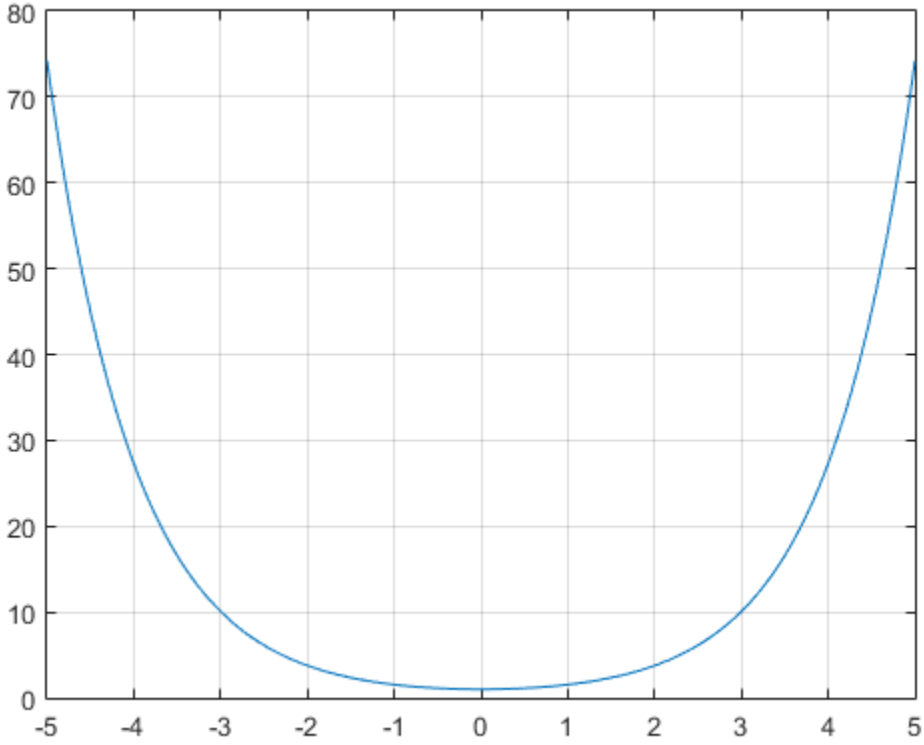
`Y = cosh(X)` returns the hyperbolic cosine for each element of `X`.

Examples

Graph of Hyperbolic Cosine

Graph the hyperbolic cosine function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,cosh(x)), grid on
```



Hyperbolic Cosine

The hyperbolic cosine of z is

$$\cosh(z) = \frac{e^z + e^{-z}}{2}.$$

See Also

`acosh` | `cos` | `sinh` | `tanh`

Introduced before R2006a

cot

Cotangent of angle in radians

Syntax

```
Y = cot(X)
```

Description

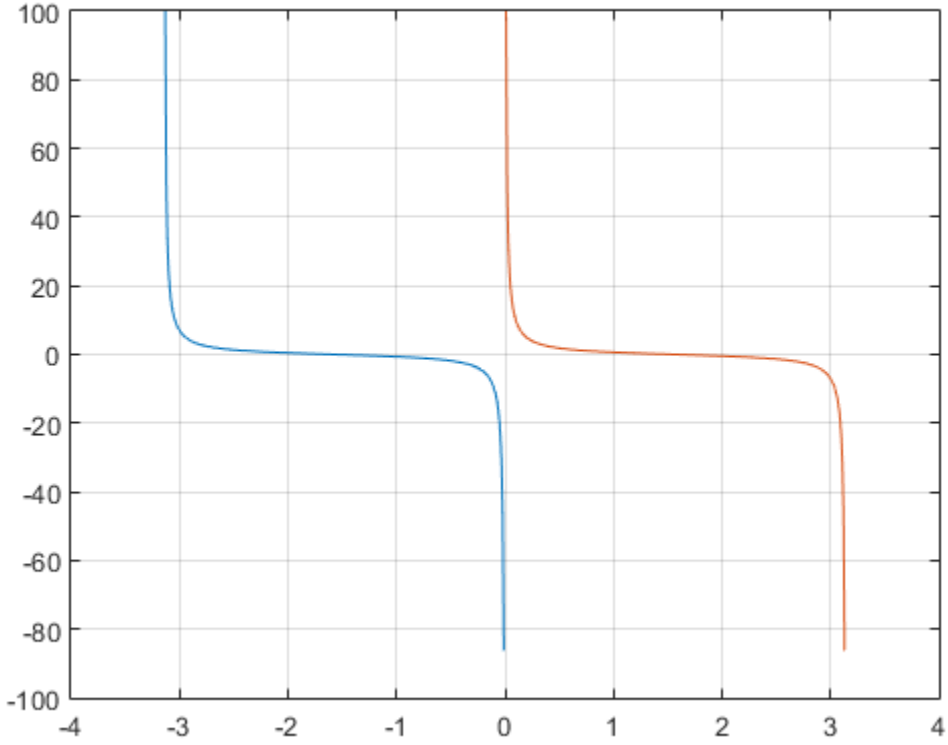
`Y = cot(X)` returns the cotangent of elements of X. The `cot` function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval [-Inf,Inf], `cot` returns real values in the interval [-Inf,Inf].. For complex values of X, `cot` returns complex values. All angles are in radians.

Examples

Plot Cotangent Function

Plot the cotangent function over the domain $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,cot(x1),x2,cot(x2)), grid on
```



Cotangent of Vector of Complex Angles

Calculate the cotangent of the complex angles in vector `x`.

```
x = [-i pi+i*pi/2 -1+i*4];
y = cot(x)
```

```
y =

   0.0000 + 1.3130i  -0.0000 - 1.0903i  -0.0006 - 0.9997i
```

Input Arguments

X — Input angle in radians

number | vector | matrix | multidimensional array

Input angle in radians, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cotangent of input angle

scalar value | vector | matrix | N-D array

Cotangent of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or N-D array.

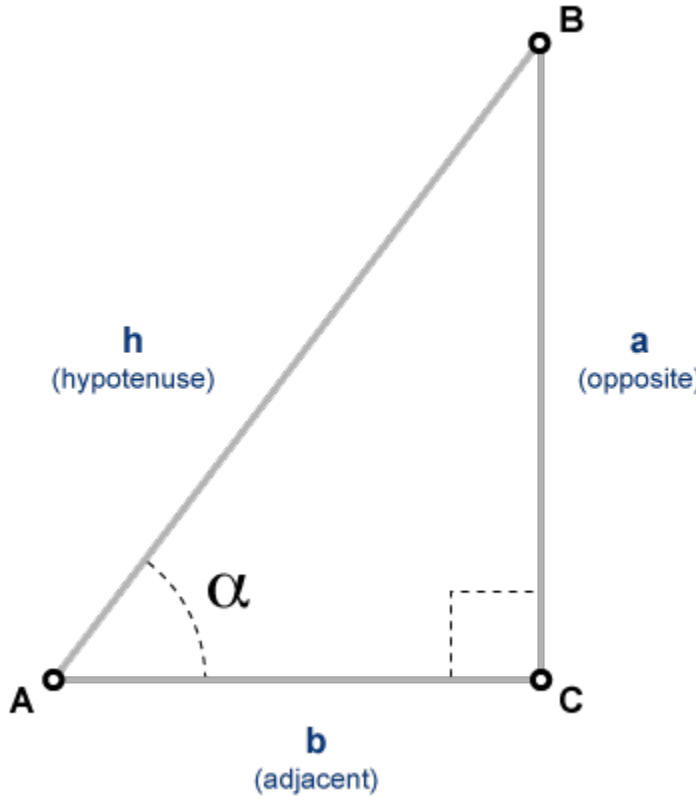
More About

Cotangent Function

The cotangent of an angle, α, defined with reference to a right angled triangle is

$$\cot(\alpha) = \frac{1}{\tan(\alpha)} = \frac{\text{adjacent side}}{\text{opposite side}} = \frac{b}{a}.$$

.



The cotangent of a complex angle α is

$$\cot(\alpha) = \frac{i\left(e^{i\alpha} + e^{-i\alpha}\right)}{\left(e^{i\alpha} - e^{-i\alpha}\right)}.$$

.

See Also

`acot` | `acotd` | `acoth` | `cotd` | `coth`

Introduced before R2006a

cotd

Cotangent of argument in degrees

Syntax

```
Y = cotd(X)
```

Description

`Y = cotd(X)` returns the cotangent of the elements of `X`, which are expressed in degrees.

Examples

Cotangent of angles approaching 90 and 180 degrees

Create a vector of input angles consisting of 90° and the next smaller and larger double precision numbers. Then compute the cotangent.

```
x1 = [90-eps(90) 90 90+eps(90)];
y1 = cotd(x1)
```

```
y1 =

   1.0e-15 *

    0.2480         0   -0.2480
```

`cotd` returns zero when the input angle is exactly 90°. Evaluation at the next smaller double-precision angle returns a slightly positive result. Likewise, the cotangent is slightly negative when the input angle is the next double-precision number larger than 90.

The behavior is similar for input angles near 180°.

```
x2 = [180-eps(180) 180 180+eps(180)];
y2 = cotd(x2)
```

```
y2 =

   1.0e+15 *

   -2.0159       Inf    2.0159
```

Cotangent of complex angle, specified in degrees

```
x = 35+5i;
y = cotd(x)

y =

   1.3958 - 0.2606i
```

Input Arguments

X — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cotd` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cotangent of angle

scalar value | vector | matrix | N-D array

Cotangent of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

See Also

`acot` | `acotd` | `cot`

Introduced before R2006a

coth

Hyperbolic cotangent

Syntax

```
Y = coth(X)
```

Description

The `coth` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

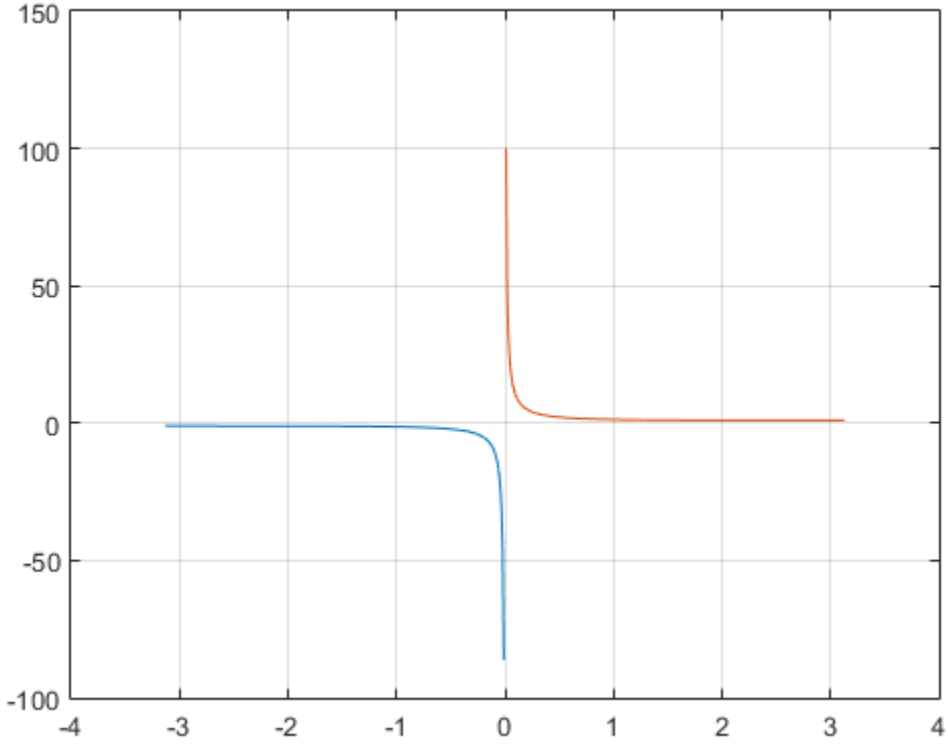
`Y = coth(X)` returns the hyperbolic cotangent for each element of `X`.

Examples

Graph of Hyperbolic Cotangent

Plot the hyperbolic cotangent over the domain $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,coth(x1),x2,coth(x2))
grid on
```



More About

Hyperbolic Cotangent

The hyperbolic cotangent of z is

$$\coth(z) = \frac{1}{\tanh(z)}.$$

See Also

`acoth` | `cot` | `sinh` | `cosh` | `tanh`

Introduced before R2006a

countcats

Count occurrences of categorical array elements by category

Syntax

```
B = countcats(A)
B = countcats(A,dim)
```

Description

`B = countcats(A)` returns the number of elements in each category of the categorical array, `A`.

- If `A` is a vector, then `countcats` returns the number of elements in each category.
- If `A` is a matrix, then `countcats` treats the columns of `A` as vectors and returns the category counts for each column of `A`.
- If `A` is a multidimensional array, then `countcats` acts along the first array dimension whose size does not equal `1`.

`B = countcats(A,dim)` returns the category counts along dimension dim.

For example, you can return the category counts of each row in a categorical array using `countcats(A,2)`.

Examples

Category Counts of Categorical Vector

Create a 1-by-5 categorical vector.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'})
```

```
A =

     plane      car      train      car      plane
```

`A` has three categories, `car`, `plane`, and `train`.

Find the number of elements in each category of A.

```
B = countcats(A)
```

```
B =

     2     2     1
```

The first element in B corresponds to the first category of A, which is `car`. The second element in B corresponds to the second category of A, which is `plane`. The third element of B corresponds to the third category of A, which is `train`.

Since A is a row vector, `countcats` returns a row vector.

Category Counts of Each Column in Array

Create a 3-by-2 categorical array, A, from a numeric array.

```
valueset = 1:3;
catnames = {'red' 'green' 'blue'};

A = categorical([1 3; 2 1; 3 1],valueset,catnames)
```

```
A =

     red         blue
     green       red
     blue        red
```

A has three categories, `red`, `green`, and `blue`.

Find the category counts of each column in A.

```
B = countcats(A)
```

```
B =

     1     2
     1     0
     1     1
```

The first row of B corresponds to the first category of A. The value, `red`, occurs once in the first column of A and twice in the second column.

The second row of B corresponds to the second category of A. The value, `green`, occurs once in the first column of A, and it does not occur in the second column.

The third row of B corresponds to the third category of A. The value, blue, occurs once in the first column of A and once in the second column.

Category Counts of Each Row in Array

Create a 3-by-2 categorical array, A, from a numeric array.

```
valueset = 1:3;
catnames = {'red' 'green' 'blue'};

A = categorical([1 3; 2 1; 3 1],valueset,catnames)
```

```
A =

     red         blue
     green       red
     blue        red
```

A has three categories, red, green, and blue.

Find the category counts of A along the second dimension.

```
B = countcats(A,2)
```

```
B =

     1     0     1
     1     1     0
     1     0     1
```

The first column of B corresponds to the first category of A. The value, red, occurs once in the first row of A, once in the second row, and once in the third row.

The second column of B corresponds to the second category of A. The value, green, occurs in only one element. It occurs in the second row of A.

The third column of B corresponds to the third category of A. The value, blue, occurs once in the first row of A and once in the third row.

Category Counts of Array Containing Undefined Elements

Create a 6-by-1 categorical array, A, from a numeric array.

```
valueset = 1:3;
catnames = {'red' 'green' 'blue'};
```

```
A = categorical([1;3;2;1;3;1],valueset,catnames)
```

```
A =

     red
     blue
     green
     red
     blue
     red
```

Remove the `blue` category.

```
A = removecats(A,'blue')
```

```
A =

     red
     <undefined>
     green
     red
     <undefined>
     red
```

`A` has two categories, `red` and `green`. Elements of `A` that were from the `blue` category are now undefined.

Find the number of elements in each category of `A`.

```
B = countcats(A)
```

```
B =

     3
     1
```

The first element in `B` corresponds to the first category of `A`. The value, `red`, occurs three times in `A`.

The second element in `B` corresponds to the second category of `A`. The value, `green`, occurs once in `A`.

`countcats` does not return any information on undefined elements.

Use the `summary` function to view the number of undefined elements in addition to the number of elements in each category of `A`.

```
summary(A)

     red              3
     green            1
     <undefined>      2
```

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

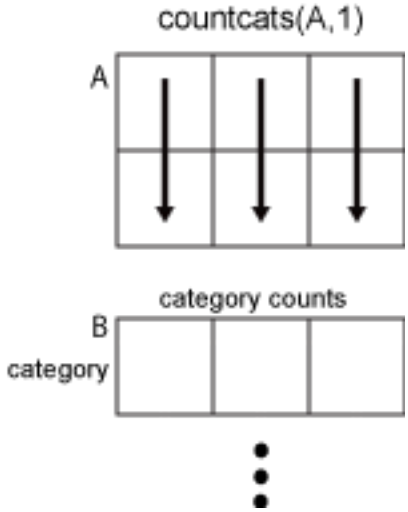
dim — Dimension to operate along

positive integer scalar

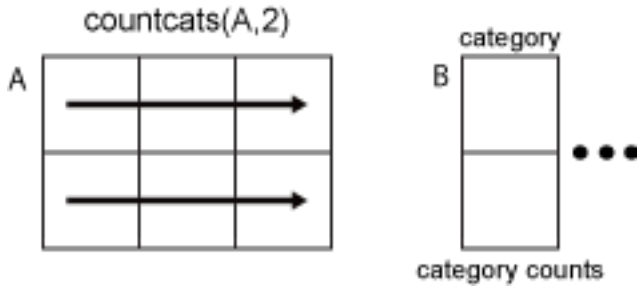
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional categorical array, A.

If `dim = 1`, then `countcats(A,1)` returns the category counts for each column of A.



If `dim = 2`, then `countcats(A,2)` returns the category counts of each row of A.



If `dim` is greater than `ndims(A)`, then `countcats(A)` returns an array the same size as `A` for each category. `countcats` returns `1` for elements in the corresponding category and `0` otherwise.

More About

Tips

- To find the number of undefined elements in a categorical array, `A`, you must use `summary` or `isundefined`.

See Also

`categories` | `iscategory` | `ismember` | `isundefined` | `summary`

cov

Covariance

Syntax

```
C = cov(A)
C = cov(A,B)
C = cov(___,w)
C = cov(___,nanflag)
```

Description

`C = cov(A)` returns the covariance.

- If `A` is a vector of observations, `C` is the scalar-valued variance.
- If `A` is a matrix whose columns represent random variables and whose rows represent observations, `C` is the covariance matrix with the corresponding column variances along the diagonal.
- `C` is normalized by the number of observations-`1`. If there is only one observation, it is normalized by 1.
- If `A` is a scalar, `cov(A)` returns `0`. If `A` is an empty array, `cov(A)` returns `NaN`.

`C = cov(A,B)` returns the covariance between two random variables `A` and `B`.

- If `A` and `B` are vectors of observations with equal length, `cov(A,B)` is the 2-by-2 covariance matrix.
- If `A` and `B` are matrices of observations, `cov(A,B)` treats `A` and `B` as vectors and is equivalent to `cov(A(:),B(:))`. `A` and `B` must have equal size.
- If `A` and `B` are scalars, `cov(A,B)` returns a 2-by-2 block of zeros. If `A` and `B` are empty arrays, `cov(A,B)` returns a 2-by-2 block of `NaN`.

`C = cov(___,w)` specifies the normalization weight for any of the previous syntaxes. When `w = 0` (default), `C` is normalized by the number of observations-`1`. When `w = 1`, it is normalized by the number of observations.

`C = cov( ___ ,nanflag)` specifies a condition for omitting `NaN` values from the calculation for any of the previous syntaxes. For example, `cov(A,'omitrows')` will omit any rows of `A` with one or more `NaN` elements.

Examples

Covariance of Matrix

Create a 3-by-4 matrix and compute its covariance.

```
A = [5 0 3 7; 1 -5 7 3; 4 9 8 10];
C = cov(A)
```

```
C =

    4.3333    8.8333   -3.0000    5.6667
    8.8333   50.3333    6.5000   24.1667
   -3.0000    6.5000    7.0000    1.0000
    5.6667   24.1667    1.0000   12.3333
```

Since the number of columns of `A` is 4, the result is a 4-by-4 matrix.

Covariance of Two Vectors

Create two vectors and compute their 2-by-2 covariance matrix.

```
A = [3 6 4];
B = [7 12 -9];
cov(A,B)
```

```
ans =

    2.3333    6.8333
    6.8333  120.3333
```

Covariance of Two Matrices

Create two matrices of the same size and compute their 2-by-2 covariance.

```
A = [2 0 -9; 3 4 1];
```

```
B = [5 2 6; -4 4 9];
cov(A,B)
```

```
ans =

   22.1667   -6.9333
   -6.9333   19.4667
```

Specify Normalization Weight

Create a matrix and compute the covariance normalized by the number of rows.

```
A = [1 3 -7; 3 9 2; -5 4 6];
C = cov(A,1)
```

```
C =

   11.5556    5.1111  -10.2222
    5.1111    6.8889    5.2222
  -10.2222    5.2222   29.5556
```

Covariance Excluding `NaN`

Create a matrix and compute its covariance, excluding any rows containing `NaN` values.

```
A = [1.77 -0.005 3.98; NaN -2.95 NaN; 2.54 0.19 1.01]
```

```
A =

    1.7700   -0.0050    3.9800
       NaN   -2.9500       NaN
    2.5400    0.1900    1.0100
```

```
C = cov(A,'omitrows')
```

```
C =

    0.2964    0.0751   -1.1435
    0.0751    0.0190   -0.2896
```

```
   -1.1435   -0.2896    4.4105
```

Input Arguments

A — Input array

vector | matrix

Input array, specified as a vector or matrix.

Data Types: `single` | `double`

B — Additional input array

vector | matrix

Additional input matrix, specified as a vector or matrix. `B` must be the same size as `A`.

Data Types: `single` | `double`

w — Normalization weight

0 (default) | 1

Normalization weight, specified as one of these values:

- `0` — The output is normalized by the number of observations `-1`. If there is only one observation, it is normalized by 1.
- `1` — The output is normalized by the number of observations.

Data Types: `single` | `double`

nanflag — NaN condition

'`includenan`' (default) | '`omitrows`' | '`partialrows`'

`NaN` condition, specified as one of these values:

- '`includenan`' — include all `NaN` values in the input prior to computing the covariance.
- '`omitrows`' — omit any row of input containing one or more `NaN` values prior to computing the covariance.
- '`partialrows`' — omit rows containing `NaN` only on a pair-wise basis for each two-column covariance calculation.

Data Types: `char`

Output Arguments

`C` — Covariance

scalar | matrix

Covariance, specified as a scalar or matrix.

- For single matrix input, `C` has size `[size(A,2) size(A,2)]` based on the number of random variables (columns) represented by `A`. The variances of the columns are along the diagonal. If `A` is a row or column vector, `C` is the scalar-valued variance.
- For two-vector or two-matrix input, `C` is the 2-by-2 covariance matrix between the two random variables. The variances are along the diagonal of `C`.

More About

Covariance

For two random variable vectors A and B, the covariance is defined as

$$\mathrm{cov}(A,B) = \frac{1}{N-1}\sum_{i=1}^{N}(A_i - \mu_A)^{*}(B_i - \mu_B)$$

where μ_A is the mean of A, μ_B is the mean of B, and `*` denotes the complex conjugate.

The *covariance matrix* of two random variables is the matrix of pair-wise covariance calculations between each variable,

$$C = \begin{pmatrix} \mathrm{cov}(A,A) & \mathrm{cov}(A,B) \\ \mathrm{cov}(B,A) & \mathrm{cov}(B,B) \end{pmatrix}.$$

For a matrix `A` whose columns are each a random variable made up of observations, the covariance matrix is the pair-wise covariance calculation between each column combination. In other words, $C(i,j) = \mathrm{cov}\left(A(:,i), A(:,j)\right)$.

Variance

For a random variable vector A made up of N scalar observations, the variance is defined as

$$V = \frac{1}{N-1}\sum_{i=1}^{N} |A_i - \mu|^2$$

where μ is the mean of A,

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i.$$

Some definitions of variance use a normalization factor of N instead of $N-1$, which can be specified by setting `w` to `1`. In either case, the mean is assumed to have the usual normalization factor N.

See Also

`corrcoef` | `mean` | `median` | `std` | `var` | `xcorr` | `xcov`

Introduced before R2006a

cplxpair

Sort complex numbers into complex conjugate pairs

Syntax

```
B = cplxpair(A)
B = cplxpair(A,tol)
B = cplxpair(A,[],dim)
B = cplxpair(A,tol,dim)
```

Description

`B = cplxpair(A)` sorts the elements along different dimensions of a complex array, grouping together complex conjugate pairs.

The conjugate pairs are ordered by increasing real part. Within a pair, the element with negative imaginary part comes first. The purely real values are returned following all the complex pairs. The complex conjugate pairs are forced to be exact complex conjugates. A default tolerance of `100*eps` relative to `abs(A(i))` determines which numbers are real and which elements are paired complex conjugates.

If `A` is a vector, `cplxpair(A)` returns `A` with complex conjugate pairs grouped together.

If `A` is a matrix, `cplxpair(A)` returns `A` with its columns sorted and complex conjugates paired.

If `A` is a multidimensional array, `cplxpair(A)` treats the values along the first non-singleton dimension as vectors, returning an array of sorted elements.

`B = cplxpair(A,tol)` overrides the default tolerance.

`B = cplxpair(A,[],dim)` sorts `A` along the dimension specified by scalar dim.

`B = cplxpair(A,tol,dim)` sorts `A` along the specified dimension and overrides the default tolerance.

Diagnostics

If there are an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message

```
Complex numbers can't be paired.
```

Introduced before R2006a

cputime

Elapsed CPU time

Syntax

```
cputime
```

Description

`cputime` returns the total CPU time (in seconds) used by your MATLAB application from the time it was started. This number can overflow the internal representation and wrap around.

Examples

The following code returns the CPU time used to run `surf(peaks(40))`.

```
t = cputime;
surf(peaks(40));
e = cputime-t

e =
     0.4667
```

More About

Tips

Although it is possible to measure performance using the `cputime` function, it is recommended that you use the `timeit` or `tic` and `toc` functions for this purpose exclusively. See Using tic and toc Versus the cputime Function in the MATLAB Programming Fundamentals documentation for more information.

See Also

`clock` | `etime` | `tic` | `timeit` | `toc`

Introduced before R2006a

RandStream.create

Create random number streams

Class

`RandStream`

Syntax

```
[s1,s2,...] = RandStream.create('gentype','NumStreams',n)
s = RandStream.create('gentype')
[ ... ] = RandStream.create('gentype', Name, Value,...)
```

Description

`[s1,s2,...] = RandStream.create('gentype','NumStreams',n)` creates `n` random number streams that use the uniform pseudorandom number generator algorithm specified by `gentype`. The streams are independent in a pseudorandom sense. The streams are not necessarily independent from streams created at other times. `RandStream.list` returns all possible values for `gentype` or see " Choosing a Random Number Generator" in the MATLAB Mathematics documentation for details on generator algorithms.

Note: Multiple streams are not supported by all generator types. Use either the multiplicative lagged Fibonacci generator (`mlfg6331_64`) or the combined multiple recursive generator (`mrg32k3a`) to create multiple streams.

`s = RandStream.create('gentype')` creates a single random stream. The `RandStream` constructor is a more concise alternative when you need to create a single stream.

`[ ... ] = RandStream.create('gentype', Name, Value,...)` allows you to specify optional Name, Value pairs to control creation of the stream. The parameters are:

`NumStreams`	Total number of streams of this type that will be created across sessions or labs. Default is 1.
`StreamIndices`	Stream indices that should be created in this call. Default is `1:N`, where `N` is the value given with the `'NumStreams'` parameter.
`Seed`	Nonnegative scalar integer with which to initialize all streams. Default is 0. Seeds must be an integer between 0 and $2^{32} - 1$ or `'shuffle'` to create a seed based on the current time.
`NormalTransform`	Transformation algorithm used by `randn(S, ...)` to generate normal pseudorandom values. Options are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.
`CellOutput`	Logical flag indicating whether or not to return the stream objects as elements of a cell array. Default is false.

Typically, you call `RandStream.create` once to create multiple independent streams in a single pass. Alternatively, you can create each stream from separate calls to `RandStream.create`, but you must specify the appropriate values for `gentype`, `'NumStreams'`, `'Seed'`, and `'StreamIndices'` to ensure their independence:

- Specify the same set of values for `gentype`, `'NumStreams'`, and `'Seed'` in each case.
- Specify a different value for `'StreamIndices'` that is between `1` and the `'NumStreams'` value in each case.

Examples

Create three independent streams.

```
[s1,s2,s3] = RandStream.create('mrg32k3a','NumStreams',3);
r1 = rand(s1,100000,1);
r2 = rand(s2,100000,1);
r3 = rand(s3,100000,1);
```

```
corrcoef([r1,r2,r3])
```

Create one stream from a set of three independent streams and designate it as the global stream.

```
s2 = RandStream.create('mrg32k3a','NumStreams',3,'StreamIndices',2);
RandStream.setGlobalStream(s2);
```

See Also

`RandStream` | `RandStream.list`

createClassFromWsdl

Create MATLAB class based on WSDL document

Compatibility

`createClassFromWsdl` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
createClassFromWsdl(source)
```

Description

`createClassFromWsdl(source)` creates a MATLAB class based on the service name defined in source.

`createClassFromWsdl` creates a class folder, `@servicename`, in the current folder. The class folder contains:

- A method file for each Web service operation.
- A display method, `display.m`.
- A constructor, `servicename.m`.

Examples

Display Standardized Test Scores

This example illustrates how to use the function. It does not use an actual WSDL document; therefore, you cannot run it. Retrieve information from a database that provides standardized test scores. Assume the WSDL document is located at `http://examplestandardtests.com/scoreswebservice?WSDL`.

Create the MATLAB class, `@TestScoreWebService`, in the current folder.

```
createClassFromWsdl('http://examplestandardtests.com/scoreswebservice?WSDL')
```

```
Retrieving document at 'http://examplestandardtests.com/scoreswebservice?WSDL'
```

Create the service.

```
svc = TestScoreWebService
```

```
endpoint: 'http://examplestandardtests.com/scoreswebservice'
    wsdl: 'http://examplestandardtests.com/scoreswebservice?WSDL'
```

Display the class methods.

```
dir @TestScoreWebService
```

```
display.m
StudentNames.m
Tests.m
TestScoreWebService.m
```

Display the calling syntax for the `StudentNames` function.

```
help StudentNames
```

```
 StudentNames(obj)

    Get names of students who took tests

      Output:
        Names = (string)
```

Get the names. MATLAB creates a structure with the names of the test takers.

```
students = StudentNames(svc)
```

```
students =

    StudentInfo: [125x1 struct]
```

View the data for the first student.

```
students.StudentInfo(1)
```

```
StudentNameLast: 'Benjamin'
StudentNameFirst: 'Ali'
```

- “Specify Proxy Server Settings for Connecting to the Internet”

Input Arguments

`source` — Web Services Description Language (WSDL) document

string

Web Services Description Language (WSDL) document, specified as a string. The name must include the location of the document, using one of the following:

- URL
- Full path
- Relative path

Example: `'http://examplestandardtests.com/scoreswebservice?WSDL'`

See Also

`createSoapMessage` | `matlab.wsdl.createWSDLClient` | `xmlread`

Introduced before R2006a

createSoapMessage

Create SOAP (Simple Object Access Protocol) message to send to server

Compatibility

`createSoapMessage` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
message = createSoapMessage(namespace,method,values,names,types)
message = createSoapMessage(namespace,method,values,names,types,
style)
```

Description

`message = createSoapMessage(namespace,method,values,names,types)` creates a SOAP message.

`message = createSoapMessage(namespace,method,values,names,types, style)` creates message with specified style.

Examples

Retrieve Book Information from Library Database

This example assumes the library is on a local intranet and does not use an actual endpoint; therefore, you cannot run it.

Retrieve the name of the author of a book titled “In the Fall.” The relative path of the library service is `urn:LibraryCatalog`. To get the author's name, use the `getAuthor` function, which takes the book name as the input value. The `getAuthor` parameter is `nameToLookUp`. The XML data type for title is `{http://www.w3.org/2001/XMLSchema}string`. The SOAP message style is `rpc` by default.

Create the SOAP message.

```
message = createSoapMessage(...
  'urn:LibraryCatalog',...
  'getAuthor',...
  {'In the Fall'},...
  {'nameToLookUp'},...
  {'{http://www.w3.org/2001/XMLSchema}string'})
```

```
message =

[#document: null]
```

This response does not necessarily indicate that the message is valid, although certain input problems produce error messages.

Send the message to the server for processing, and get the author's name back. The server endpoint is `http://test/soap/services/LibraryCatalog`. The server method is `urn:LibraryCatalog#getAuthor`.

```
response = callSoapService(...
  'http://test/soap/services/LibraryCatalog',...
  'urn:LibraryCatalog#getAuthor',...
  message)
```

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<getAuthorResponse xmlns="urn:LibraryCatalog">
<ns1:getAuthorReturn xmlns:ns1="http://latestversion.soap.test">
Kate Alvin
</ns1:getAuthorReturn>
</getAuthorResponse>
</soapenv:Body>
</soapenv:Envelope>
```

MATLAB returns the message in a single line, displayed here on separate lines for legibility.

Extract the author's name.

```
author = parseSoapResponse(response)
```

```
author = Kate Alvin
```

MATLAB automatically converted the XML string data type to `char`.

Input Arguments

`namespace` — Location of Web service

string

Location of Web service, specified as a string in the form of a valid Uniform Resource Identifier (URI).

Example: `'urn:LibraryCatalog'`

`method` — Name of Web service operation

string

Name of Web service operation, specified as a string.

Example: `'getAuthor'`

`values` — Input arguments for `method`

cell array

Input arguments for `method`, specified as a cell array.

Example: `{'In the Fall'}`

`names` — Parameter for `method`

cell array

Parameter for `method`, specified as a cell array.

Example: `{'nameToLookUp'}`

`types` — XML data types for `values`

cell array

XML data types for `values`, specified as a cell array.

Example: `{'{http://www.w3.org/2001/XMLSchema}string'}`

`style` — Style for structuring SOAP message

`'rpc'` (default) | `'document'`

Style for structuring the SOAP message, specified as one of these values. Use a style supported by the service specified in namespace.

`'rpc'`	Remote Procedure Call (RPC) encoding
`'document'`	Document-style encoding

Output Arguments

`message` — Java document object model (DOM)

string

Java document object model (DOM), returned as a string. Use message as input to `callSoapService` function.

See Also

`callSoapService` | `matlab.wsdl.createWSDLClient` | `parseSoapResponse` | `urlread` | `xmlread`

Introduced before R2006a

cross

Cross product

Syntax

```
C = cross(A,B)
C = cross(A,B,dim)
```

Description

`C = cross(A,B)` returns the cross product of `A` and `B`.

- If `A` and `B` are vectors, then they must have a length of 3.
- If `A` and `B` are matrices or multidimensional arrays, then they must have the same size. In this case, the `cross` function treats `A` and `B` as collections of three-element vectors. The function calculates the cross product of corresponding vectors along the first array dimension whose size equals 3.

`C = cross(A,B,dim)` evaluates the cross product of arrays `A` and `B` along dimension, `dim`. `A` and `B` must have the same size, and both `size(A,dim)` and `size(B,dim)` must be 3. The `dim` input is a positive integer scalar.

Examples

Cross Product of Vectors

Create two 3-D vectors.

```
A = [4 -2 1];
B = [1 -1 3];
```

Find the cross product of `A` and `B`.

```
C = cross(A,B)
```

```
C =
```

```
    -5   -11    -2
```

The result, `C`, is a vector that is perpendicular to both `A` and `B`.

Use dot products to verify that `C` is perpendicular to `A` and `B`.

```
dot(C,A)==0 & dot(C,B)==0
```

```
ans =

     1
```

The result is logical `1` (`true`).

Cross Product of Matrices

Create two matrices containing random integers.

```
rng(0)
A = randi(15,3,5)
B = randi(25,3,5)
```

```
A =

    13    14     5    15    15
    14    10     9     3     8
     2     2    15    15    13


B =

     4    20     1    17    10
    11    24    22    19    17
    23    17    24    19     5
```

Find the cross product of `A` and `B`.

```
C = cross(A,B)
```

```
C =

   300   122  -114  -228  -181
  -291  -198  -105   -30    55
    87   136   101   234   175
```

The result, C, contains five independent cross products between the columns of A and B. For example, C(:,1) is equal to the cross product of A(:,1) with B(:,1).

Cross Product of Multidimensional Arrays

Create two 3-by-3-by-3 multidimensional arrays of random integers.

```
rng(0)
A = randi(10,3,3,3);
B = randi(25,3,3,3);
```

Find the cross product of A and B, treating the rows as vectors.

```
C = cross(A,B,2)
```

```
C(:,:,1) =

   -34    12    62
    15    72  -109
   -49     8     9


C(:,:,2) =

   198  -164  -170
    45     0   -18
   -55   190  -116


C(:,:,3) =

  -109   -45   131
     1   -74    82
    -6   101  -121
```

The result is a collection of row vectors. For example, C(1,:,1) is equal to the cross product of A(1,:,1) with B(1,:,1).

Find the cross product of A and B along the third dimension (dim = 3).

```
D = cross(A,B,3)
```

```
D(:,:,1) =

   -14   179  -106
   -56    -4   -75
```

```
     2   -37    10


D(:,:,2) =

   -37  -162   -37
    50  -124   -78
     1    63   118


D(:,:,3) =

    62  -170    56
    46    72   105
    -2   -53  -160
```

The result is a collection of vectors oriented in the third dimension. For example, `C(1,1,:)` is equal to the cross product of `A(1,1,:)` with `B(1,1,:)`.

Input Arguments

`A,B` — Input arrays

numeric arrays

Input arrays, specified as numeric arrays.

Data Types: `single | double`
Complex Number Support: Yes

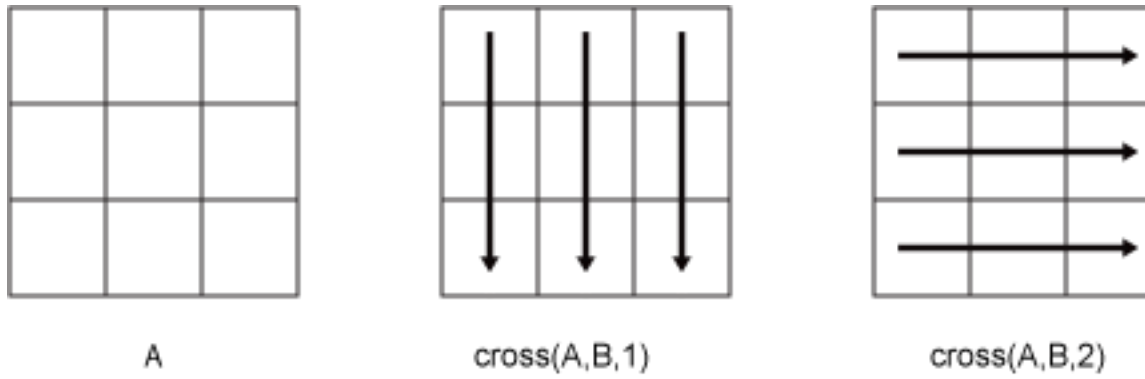
`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. The size of dimension `dim` must be 3. If no value is specified, the default is the first array dimension whose size equals 3.

Consider two 2-D input arrays, `A` and `B`:

- `cross(A,B,1)` treats the columns of `A` and `B` as vectors and returns the cross products of corresponding columns.
- `cross(A,B,2)` treats the rows of `A` and `B` as vectors and returns the cross products of corresponding rows.



`cross` returns an error if `dim` is greater than `ndims(A)`.

More About

Cross Product

The cross product between two 3-D vectors produces a new vector that is perpendicular to both.

Consider the two vectors

$$A = a_1\hat{i} + a_2\hat{j} + a_3\hat{k},$$
$$B = b_1\hat{i} + b_2\hat{j} + b_3\hat{k}.$$

In terms of a matrix determinant involving the basis vectors $\hat{i}$, $\hat{j}$, and $\hat{k}$, the cross product of A and B is

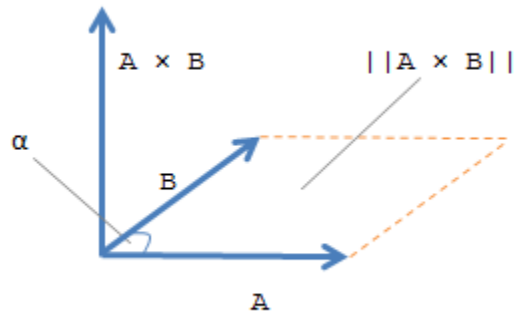
$$C = A \times B = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

$$= (a_2b_3 - a_3b_2)\hat{i} + (a_3b_1 - a_1b_3)\hat{j} + (a_1b_2 - a_2b_1)\hat{k}.$$

Geometrically, $A \times B$ is perpendicular to both A and B. The magnitude of the cross product, $\|A \times B\|$, is equal to the area of the parallelogram formed using A and B as sides. This area is related to the magnitudes of A and B as well as the angle between the vectors by

$$\|A \times B\| = \|A\| \|B\| \sin \alpha \ .$$

Thus, if A and B are parallel, then the cross product is zero.



See Also

`dot` | `kron`

Introduced before R2006a

csc

Cosecant of input angle in radians

Syntax

```
Y = csc(X)
```

Description

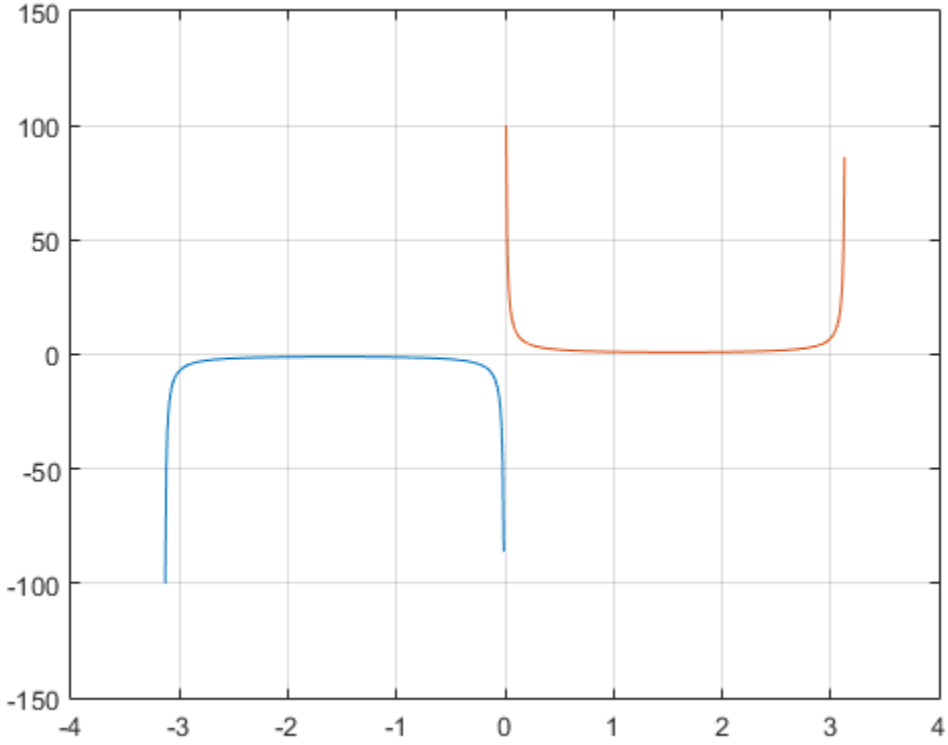
`Y = csc(X)` returns the cosecant of the elements of `X`. The `csc` function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval [-Inf, Inf], `csc` returns real values in the interval [-Inf ,-1] and [1,Inf]. For complex values of X, `csc` returns complex values. All angles are in radians.

Examples

Plot Cosecant Function

Plot the cosecant function over the domain $-\pi < x < 0$ and $0 < x < \pi$ as shown.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csc(x1),x2,csc(x2)), grid on
```



Cosecant of Vector of Complex Angles

Calculate the cosecant of the complex angles in vector `x`.

```
x = [-i pi+i*pi/2 -1+i*4];
y = csc(x)
```

```
y =

   0.0000 + 0.8509i   0.0000 + 0.4345i  -0.0308 - 0.0198i
```

Input Arguments

X — Input angle in radians

number | vector | matrix | multidimensional array

Input angle in radians, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosecant of input angle

scalar value | vector | matrix | N-D array

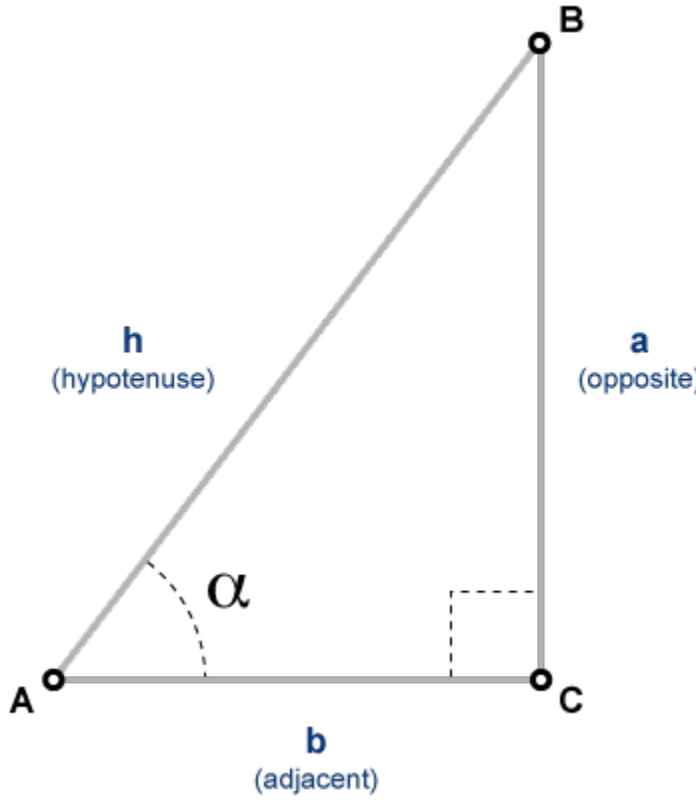
Cosecant of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or N-D array.

More About

Cosecant Function

The cosecant of an angle, α, defined with reference to a right angled triangle is

$$\csc(\alpha) = \frac{1}{\sin(\alpha)} = \frac{\text{hypotenuse}}{\text{opposite side}} = \frac{h}{a}.$$



The cosecant of a complex angle, α, is

$$\csc(\alpha) = \frac{2i}{e^{i\alpha} - e^{-i\alpha}}.$$

See Also

acsc | acscd | acsch | cscd | csch

Introduced before R2006a

cscd

Cosecant of argument in degrees

Syntax

```
Y = cscd(X)
```

Description

`Y = cscd(X)` returns the cosecant of the elements of `X`, which are expressed in degrees.

Examples

Cosecant of 180 degrees compared to cosecant of π radians

`cscd(180)` is infinite, whereas `csc(pi)` is large but finite.

```
cscd(180)
```

```
ans =

   Inf
```

```
csc(pi)
```

```
ans =

   8.1656e+15
```

Cosecant of vector of complex angles, specified in degrees

```
z = [35+i 15+2i 10+3i];
y = cscd(z)
```

```
y =
```

```
   1.7421 - 0.0434i   3.7970 - 0.4944i   5.2857 - 1.5681i
```

Input Arguments

X — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cscd` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosecant of angle

scalar value | vector | matrix | N-D array

Cosecant of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

See Also

`acsc` | `acscd` | `csc`

Introduced before R2006a

csch

Hyperbolic cosecant

Syntax

```
Y = csch(x)
```

Description

The `csch` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

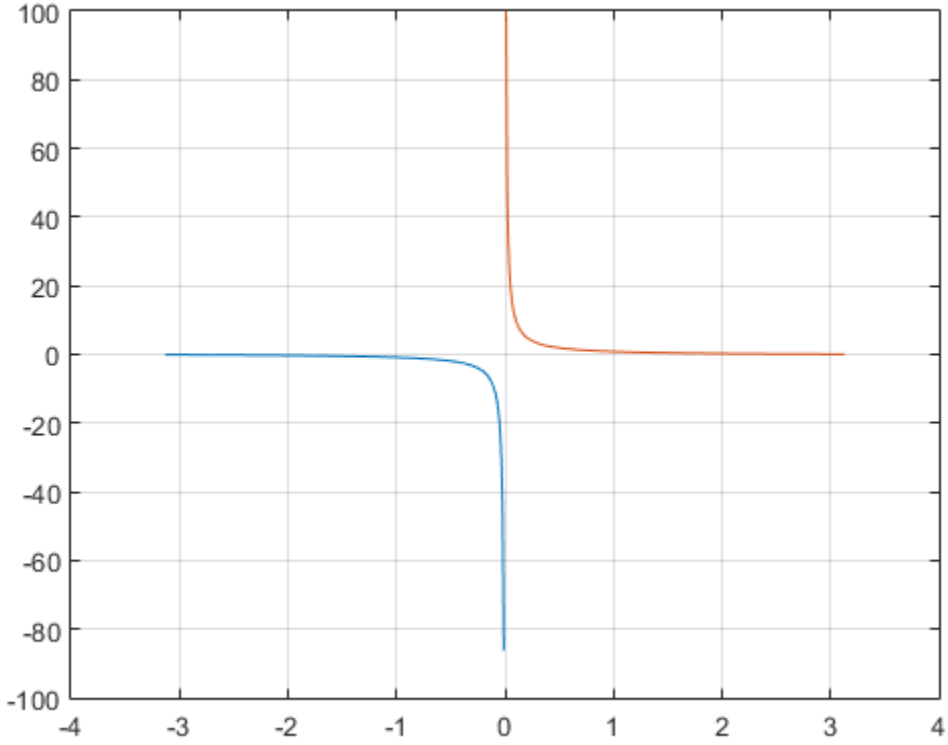
`Y = csch(x)` returns the hyperbolic cosecant for each element of `x`.

Examples

Graph of Hyperbolic Cosecant

Plot the hyperbolic cosecant over the domain $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csch(x1),x2,csch(x2)), grid on
```



More About

Hyperbolic Cosecant

The hyperbolic cosecant of z is

$$\operatorname{csch}(z) = \frac{1}{\sinh(z)}.$$

See Also

`acsch` | `csc` | `sinh` | `cosh`

Introduced before R2006a

csvread

Read comma-separated value (CSV) file

Syntax

```
M = csvread(filename)
M = csvread(filename,R1,C1)
M = csvread(filename,R1,C1,[R1 C1 R2 C2])
```

Description

`M = csvread(filename)` reads a comma-separated value (CSV) formatted file into array `M`. The file must contain only numeric values.

`M = csvread(filename,R1,C1)` reads data from the file starting at row offset `R1` and column offset `C1`. For example, the offsets `R1=0`, `C1=0` specify the first value in the file.

`M = csvread(filename,R1,C1,[R1 C1 R2 C2])` reads only the range bounded by row offsets `R1` and `R2` and column offsets `C1` and `C2`. Another way to define the range is to use spreadsheet notation, such as `'A1..B7'` instead of `[0 0 6 1]`.

Examples

Read Entire CSV File

Create a file named `csvlist.dat` that contains comma-separated values.

```
02, 04, 06, 08
03, 06, 09, 12
05, 10, 15, 20
07, 14, 21, 28
```

Read the numeric values in the file.

```
filename = 'csvlist.dat';
M = csvread(filename)
```

```
M =

     2     4     6     8
     3     6     9    12
     5    10    15    20
     7    14    21    28
```

Read CSV File Starting at Specific Row and Column Offset

Read the matrix starting two rows below the first row from the file described in the previous example.

```
M = csvread('csvlist.dat',2,0)

M =

     5    10    15    20
     7    14    21    28
```

Read Specific Range from CSV File

Read the matrix bounded by row offsets 1 and 2 and column offsets 0 and 2 from the file described in the first example.

```
M = csvread('csvlist.dat',1,0,[1,0,2,2])

M =

     3     6     9
     5    10    15
```

Input Arguments

`filename` — File name

string

File name, specified as a string.

Example: `'myFile.dat'`

Data Types: `char`

`R1` — Starting row offset

0 (default) | nonnegative integer

Starting row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`C1` — Starting column offset

0 (default) | nonnegative integer

Starting column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`R2` — Ending row offset

nonnegative integer

Ending row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`C2` — Ending column offset

nonnegative integer

Ending column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Tips

- Skip header rows or columns by specifying row and column offsets. All values in the file other than headers must be numeric.

Algorithms

`csvread` fills empty delimited fields with zero. When the `csvread` function reads data files with lines that end with a nonspace delimiter, such as a semicolon, it returns a matrix, `M`, that has an additional last column of zeros.

`csvread` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. The table shows valid forms for a complex number.

Form	Example
`±<real>±<imag>i|j`	`5.7-3.1i`
`±<imag>i|j`	`-7j`

Embedded white space in a complex number is invalid and is regarded as a field delimiter.

- "Ways to Import Text Files"

See Also

`csvwrite` | `dlmread` | `readtable` | `textscan` | `uiimport`

Introduced before R2006a

ctranspose, '

Complex conjugate transpose

Syntax

```
B = A'
B = ctranspose(A)
```

Description

`B = A'` computes the complex conjugate transpose of `A`.

`B = ctranspose(A)` is an alternate way to execute `A'`, but is rarely used. It enables operator overloading for classes.

Examples

Conjugate Transpose of Real Matrix

Create a 4-by-2 matrix.

```
A = [2 1; 9 7; 2 8; 3 5]
```

```
A =

     2     1
     9     7
     2     8
     3     5
```

Find the conjugate transpose of `A`.

```
B = A'
```

```
B =
```

```
     2     9     2     3
     1     7     8     5
```

The result is a 2-by-4 matrix. `B` has the same elements as `A`, but the row and column index for each element are interchanged. When no complex elements are present, `A'` produces the same result as `A.'`.

Conjugate Transpose of Complex Matrix

Create a 2-by-2 matrix with complex elements.

```
A = [0-1i 2+1i;4+2i 0-2i]
```

```
A =

   0.0000 - 1.0000i   2.0000 + 1.0000i
   4.0000 + 2.0000i   0.0000 - 2.0000i
```

Find the conjugate transpose of `A`.

```
B = A'
```

```
B =

   0.0000 + 1.0000i   4.0000 - 2.0000i
   2.0000 - 1.0000i   0.0000 + 2.0000i
```

The result, `B`, contains the elements of `A` with the row and column indices interchanged. The sign of the imaginary part of each number is also switched.

Input Arguments

`A` — Input array

vector | matrix | cell array | categorical array | datetime array | duration array | calendarDuration array | structure field

Input array, specified as a vector or matrix of any numeric, logical, or `char` data type, or as a cell array, categorical array, datetime array, duration array, calendarDuration array, or structure field.

Complex Number Support: Yes

More About

Complex Conjugate Transpose

The complex conjugate transpose of a matrix interchanges the row and column index for each element, reflecting the elements across the main diagonal. The operation also negates the imaginary part of any complex numbers.

For example, if `B = A'` and `A(1,2)` is `1+1i`, then the element `B(2,1)` is `1-1i`.

Tips

- The nonconjugate transpose operator, `A.'`, performs a transpose without conjugation. That is, it does not change the sign of the imaginary parts of the elements.
- For logical or non-numeric inputs, `ctranspose` and `transpose` produce the same result.

- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`conj` | `permute` | `transpose`

Introduced before R2006a

csvwrite

Write comma-separated value file

Syntax

```
csvwrite(filename,M)
csvwrite(filename,M,row,col)
```

Description

`csvwrite(filename,M)` writes matrix `M` into `filename` as comma-separated values. The `filename` input is a string enclosed in single quotes.

`csvwrite(filename,M,row,col)` writes matrix `M` into `filename` starting at the specified row and column offset. The row and column arguments are zero based, so that `row=0` and `C=0` specify the first value in the file.

Examples

The following example creates a comma-separated value file from the matrix `m`.

```
m = [3 6 9 12 15; 5 10 15 20 25; ...
     7 14 21 28 35; 11 22 33 44 55];

csvwrite('csvlist.dat',m)
type csvlist.dat

3,6,9,12,15
5,10,15,20,25
7,14,21,28,35
11,22,33,44,55
```

The next example writes the matrix to the file, starting at a column offset of `2`.

```
csvwrite('csvlist.dat',m,0,2)
type csvlist.dat
```

```
,,3,6,9,12,15
,,5,10,15,20,25
,,7,14,21,28,35
,,11,22,33,44,55
```

More About

Tips

- `csvwrite` terminates each line with a line feed character and no carriage return.
- `csvwrite` writes a maximum of five significant digits. If you need greater precision, use `dlmwrite` with a precision argument.
- `csvwrite` does not accept cell arrays for the input matrix `M`. To export a cell array that contains only numeric data, use `cell2mat` to convert the cell array to a numeric matrix before calling `csvwrite`. To export cell arrays with mixed alphabetic and numeric data, where each cell contains a single element, you can create an Excel spreadsheet (if your system has Excel installed) using `xlswrite`. For all other cases, you must use low-level export functions to write your data. For more information, see “Export Cell Array to Text File” in the MATLAB Data Import and Export documentation.

See Also

`csvread` | `dlmwrite` | `xlswrite` | `writetable` | `uiimport`

Introduced before R2006a

cummax

Cumulative maximum

Syntax

```
M = cummax(A)
M = cummax(A,dim)
M = cummax( ___ ,direction)
```

Description

`M = cummax(A)` returns the cumulative maximum elements of `A`. By default, `cummax(A)` operates along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `cummax(A)` returns a vector of the same size containing the cumulative maxima of `A`.
- If `A` is a matrix, then `cummax(A)` returns a matrix of the same size containing the cumulative maxima in each column of `A`.
- If `A` is a multidimensional array, then `cummax(A)` returns an array of the same size containing the cumulative maxima along the first array dimension of `A` whose size does not equal 1.

`M = cummax(A,dim)` returns the cumulative maxima along the dimension `dim`. For example, if `A` is a matrix, then `cummax(A,2)` returns the cumulative maxima along the rows of `A`.

`M = cummax( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A` and, optionally, can specify `dim`. For instance, `cummax(A,2,'reverse')` returns the cumulative maxima of `A` by working from end to beginning of the second dimension of `A`.

Examples

Cumulative Maximum Values in Vector

Find the cumulative maxima of a 1-by-10 vector of random integers.

```
v = randi(10,1,10)
```

```
v =

     8     3     6     7     9    10     6     2     2     3
```

```
M = cummax(v)
```

```
M =

     8     8     8     8     9    10    10    10    10    10
```

Cumulative Maximum Values in Matrix Columns

Find the cumulative maxima of the columns of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A =

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummax(A)
```

```
M =

     3     5     2
     3     6     3
     7     8     3
```

Cumulative Maximum Values in Matrix Rows

Find the cumulative maxima of the rows of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A =

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummax(A,2)
```

```
M =
```

```
     3     5     5
     1     6     6
     7     8     8
```

Cumulative Maximum Array Values in Reverse Direction

Calculate the cumulative maxima in the third dimension of a 2-by-2-by-3 array. Specify `direction` as `'reverse'` to work from the end of the third dimension to the beginning.

```
A = cat(3,[1 2; 3 4],[9 10; 11 12],[5 6; 7 8])
```

```
A(:,:,1) =

     1     2
     3     4


A(:,:,2) =

     9    10
    11    12


A(:,:,3) =

     5     6
     7     8
```

```
M = cummax(A,3,'reverse')
```

```
M(:,:,1) =

     9    10
    11    12


M(:,:,2) =

     9    10
    11    12


M(:,:,3) =

     5     6
```

```
     7     8
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. For complex elements, `cummax` compares the magnitude of the elements. If magnitudes are equal, `cummax` also compares the phase angles.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

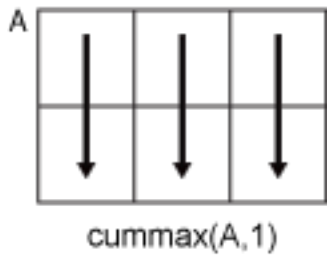
dim — Dimension to operate along

positive integer scalar

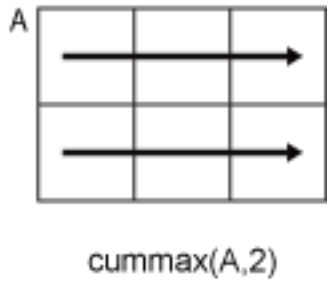
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `cummax(A,1)` works on successive elements in the columns of `A` and returns an array of the same size as `A` with the cumulative maxima in each column.



- `cummax(A,2)` works on successive elements in the rows of `A` and returns an array of the same size as `A` with the cumulative maxima in each row.



`cummax` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as the string `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Data Types: `char`

Output Arguments

`M` — Cumulative maxima
vector | matrix | multidimensional array

Cumulative maxima, returned as a vector, matrix, or multidimensional array. The size and data type of `M` are the same as those of `A`.

More About

Tips

- The `cummax` function does not return `NaN` values unless the first element in the active dimension is a `NaN`. In that case, the corresponding first element in the output is a `NaN`. If the input has further consecutive `NaN` values, so that `cummax` must compare `NaN`s to determine the cumulative maximum, the output also has consecutive `NaN`s.

- The `'reverse'` option in many cumulative functions allows quick directional calculations without requiring a flip or reflection of the input array.

See Also

`cummin` | `cumprod` | `cumsum` | `max` | `min`

Introduced in R2014b

cummin

Cumulative minimum

Syntax

```
M = cummin(A)
M = cummin(A,dim)
M = cummin( ___ ,direction)
```

Description

`M = cummin(A)` returns the cumulative minimum elements of `A`. By default, `cummin(A)` operates along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `cummin(A)` returns a vector of the same size containing the cumulative minima of `A`.
- If `A` is a matrix, then `cummin(A)` returns a matrix of the same size containing the cumulative minima in each column of `A`.
- If `A` is a multidimensional array, then `cummin(A)` returns an array of the same size containing the cumulative minima along the first array dimension of `A` whose size does not equal 1.

`M = cummin(A,dim)` returns the cumulative minima along the dimension `dim`. For example, if `A` is a matrix, then `cummin(A,2)` returns the cumulative minima along the rows of `A`.

`M = cummin( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A` and, optionally, can specify `dim`. For instance, `cummin(A,2,'reverse')` returns the cumulative minima of `A` by working from end to beginning of the second dimension of `A`.

Examples

Cumulative Minimum Values in Vector

Find the cumulative minima of a 1-by-10 vector of random integers.

```
v = randi([0,10],1,10)
```

```
v =

     8     9     1    10     6     1     3     6    10    10
```

```
M = cummin(v)
```

```
M =

     8     8     1     1     1     1     1     1     1     1
```

Cumulative Minimum Values in Matrix Columns

Find the cumulative minima of the columns of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A =

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummin(A)
```

```
M =

     3     5     2
     1     5     2
     1     5     1
```

Cumulative Minimum Values in Matrix Rows

Find the cumulative minima of the rows of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A =

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummin(A,2)
```

```
M =
```

```
     3     3     2
     1     1     1
     7     7     1
```

Cumulative Minimum Array Values in Reverse Direction

Calculate the cumulative minima in the third dimension of a 2-by-2-by-3 array. Specify `direction` as `'reverse'` to work from the end of the third dimension to the beginning.

```
A = cat(3,[1 2; 3 4],[9 10; 11 12],[5 6; 7 8])
```

```
A(:,:,1) =

     1     2
     3     4


A(:,:,2) =

     9    10
    11    12


A(:,:,3) =

     5     6
     7     8
```

```
M = cummin(A,3,'reverse')
```

```
M(:,:,1) =

     1     2
     3     4


M(:,:,2) =

     5     6
     7     8


M(:,:,3) =

     5     6
```

```
     7     8
```

Input Arguments

`A` — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. For complex elements, `cummin` compares the magnitude of the elements. If magnitudes are equal, `cummin` also compares the phase angles.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

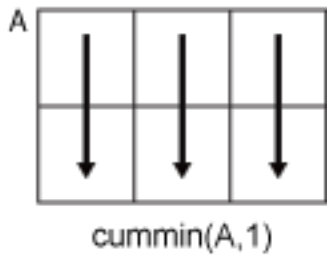
`dim` — Dimension to operate along

positive integer scalar

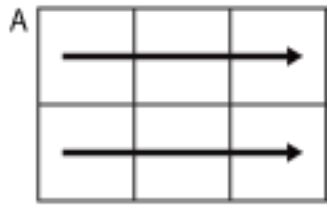
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `cummin(A,1)` works on successive elements in the columns of `A` and returns an array of the same size as `A` with the cumulative minima in each column.



- `cummin(A,2)` works on successive elements in the rows of `A` and returns an array of the same size as `A` with the cumulative minima in each row.



`cummin` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as the string `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Data Types: `char`

Output Arguments

`M` — Cumulative minima
vector | matrix | multidimensional array

Cumulative minima, returned as a vector, matrix, or multidimensional array. The size and data type of `M` are the same as those of `A`.

More About

Tips

- The `cummin` function does not return `NaN` values unless the first element in the active dimension is a `NaN`. In that case, the corresponding first element in the output is a `NaN`. If the input has further consecutive `NaN` values, so that `cummin` must compare `NaN`s to determine the cumulative minimum, the output also has consecutive `NaN`s.

- The `'reverse'` option in many cumulative functions allows quick directional calculations without requiring a flip or reflection of the input array.

See Also

`cummax` | `cumprod` | `cumsum` | `max` | `min`

Introduced in R2014b

cumprod

Cumulative product

Syntax

```
B = cumprod(A)
B = cumprod(A,dim)
B = cumprod( ___ ,direction)
```

Description

`B = cumprod(A)` returns the cumulative product of `A` starting at the beginning of the first array dimension in `A` whose size does not equal 1.

- If `A` is a vector, then `cumprod(A)` returns a vector containing the cumulative product of the elements of `A`.
- If `A` is a matrix, then `cumprod(A)` returns a matrix containing the cumulative products for each column of `A`.
- If `A` is a multidimensional array, then `cumprod(A)` acts along the first nonsingleton dimension.

`B = cumprod(A,dim)` returns the cumulative product along dimension `dim`. For example, if `A` is a matrix, then `cumprod(A,2)` returns the cumulative product of each row.

`B = cumprod( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A`, and optionally can specify `dim`. For instance, `cumprod(A,2,'reverse')` returns the cumulative product within the rows of `A` by working from end to beginning of the second dimension.

Examples

Cumulative Product of Vector

Find the cumulative product of the integers from `1` to `5`.

```
A = 1:5;
B = cumprod(A)

B =

     1     2     6    24   120
```

`B(2)` is the product of `A(1)` and `A(2)`, while `B(5)` is the product of elements `A(1)` through `A(5)`.

Cumulative Product of Each Column in Matrix

Define a 3-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 4 7; 2 5 8; 3 6 9]

A =

     1     4     7
     2     5     8
     3     6     9
```

Find the cumulative product of the columns of `A`.

```
B = cumprod(A)

B =

     1     4     7
     2    20    56
     6   120   504
```

`B(5)` is the product of `A(4)` and `A(5)`, while `B(9)` is the product of `A(7)` , `A(8)`, and `A(9)`.

Cumulative Product of Each Row in Matrix

Define a 2-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 3 5; 2 4 6]

A =

     1     3     5
     2     4     6
```

Find the cumulative product of the rows of A.

```
B = cumprod(A,2)
```

```
B =

     1     3    15
     2     8    48
```

B(3) is the product of A(1) and A(3), while B(5) is the product of A(1), A(3), and A(5) .

Logical Input with Double Output

Create an array of logical values.

```
A = [true false true; true true false]
```

```
A =

     1     0     1
     1     1     0
```

Find the cumulative product of the rows of A.

```
B = cumprod(A,2)
```

```
B =

     1     0     0
     1     1     0
```

The output is double.

```
class(B)
```

```
ans =

double
```

Reverse Cumulative Product

Create a 3-by-3 matrix of random integers between 1 and 10.

```
rng default;
A = randi([1,10],3)
```

```
A =

     9    10     3
    10     7     6
     2     1    10
```

Calculate the cumulative product along the columns. Specify the `'reverse'` option to work from bottom to top in each column.

```
B = cumprod(A,'reverse')

B =

   180    70   180
    20     7    60
     2     1    10
```

The result is the same size as `A`.

Input Arguments

`A` — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

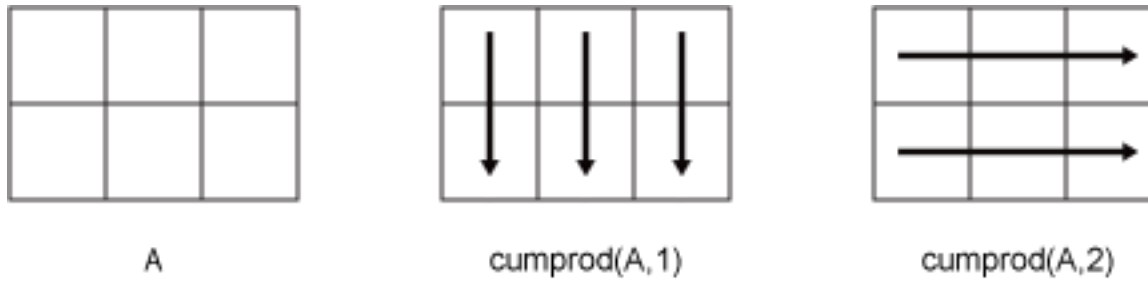
`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`.

- `cumprod(A,1)` works on successive elements in the columns of `A` and returns the cumulative products of each column.
- `cumprod(A,2)` works on successive elements in the rows of `A` and returns the cumulative products of each row.



`cumprod` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as the string `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Data Types: `char`

Output Arguments

`B` — Cumulative product array
vector | matrix | multidimensional array

Cumulative product array, returned as a vector, matrix, or multidimensional array of the same size as the input array `A`.

The class of `B` is the same as the class of `A` except if `A` is `logical`, in which case `B` is `double`.

More About

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of `X`.

Tips

- Many cumulative functions in MATLAB support the `'reverse'` option. This option allows quick directional calculations without needing a flip or reflection of the input array.

See Also

`cummax` | `cummin` | `cumsum` | `prod` | `sum`

Introduced before R2006a

cumsum

Cumulative sum

Syntax

```
B = cumsum(A)
B = cumsum(A,dim)
B = cumsum( ___ ,direction)
```

Description

`B = cumsum(A)` returns the cumulative sum of `A` starting at the beginning of the first array dimension in `A` whose size does not equal 1.

- If `A` is a vector, then `cumsum(A)` returns a vector containing the cumulative sum of the elements of `A`.
- If `A` is a matrix, then `cumsum(A)` returns a matrix containing the cumulative sums for each column of `A`.
- If `A` is a multidimensional array, then `cumsum(A)` acts along the first nonsingleton dimension.

`B = cumsum(A,dim)` returns the cumulative sum of the elements along dimension `dim`. For example, if `A` is a matrix, then `cumsum(A,2)` returns the cumulative sum of each row.

`B = cumsum( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A`, and optionally can specify `dim`. For instance, `cumsum(A,2,'reverse')` returns the cumulative sum within the rows of `A` by working from end to beginning of the second dimension.

Examples

Cumulative Sum of Vector

Find the cumulative sum of the integers from `1` to `5`.

```
A = 1:5;
B = cumsum(A)

B =

     1     3     6    10    15
```

`B(2)` is the sum of `A(1)` and `A(2)`, while `B(5)` is the sum of elements `A(1)` through `A(5)`.

Cumulative Sum of Each Column in Matrix

Define a 3-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 4 7; 2 5 8; 3 6 9]

A =

     1     4     7
     2     5     8
     3     6     9
```

Find the cumulative sum of the columns of `A`.

```
B = cumsum(A)

B =

     1     4     7
     3     9    15
     6    15    24
```

`B(5)` is the sum of `A(4)` and `A(5)`, while `B(9)` is the sum of `A(7)` , `A(8)`, and `A(9)`.

Cumulative Sum of Each Row in Matrix

Define a 2-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 3 5; 2 4 6]

A =

     1     3     5
     2     4     6
```

Find the cumulative sum of the rows of `A`.

```
B = cumsum(A,2)

B =

     1     4     9
     2     6    12
```

`B(3)` is the sum of `A(1)` and `A(3)`, while `B(5)` is the sum of `A(1)`, `A(3)`, and `A(5)` .

Logical Input with Double Output

Create an array of logical values.

```
A = [true false true; true true false]

A =

     1     0     1
     1     1     0
```

Find the cumulative sum of the rows of `A`.

```
B = cumsum(A,2)

B =

     1     1     2
     1     2     2
```

The output is `double`.

```
class(B)

ans =

double
```

Reverse Cumulative Sum

Create a 3-by-3 matrix of random integers between 1 and 10.

```
rng default;
A = randi([1,10],3)

A =
```

```
     9    10     3
    10     7     6
     2     1    10
```

Calculate the cumulative sum along the rows. Specify the `'reverse'` option to work from right to left in each row.

```
B = cumsum(A,2,'reverse')
```

```
B =

    22    13     3
    23    13     6
    13    11    10
```

The result is the same size as `A`.

Input Arguments

`A` — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

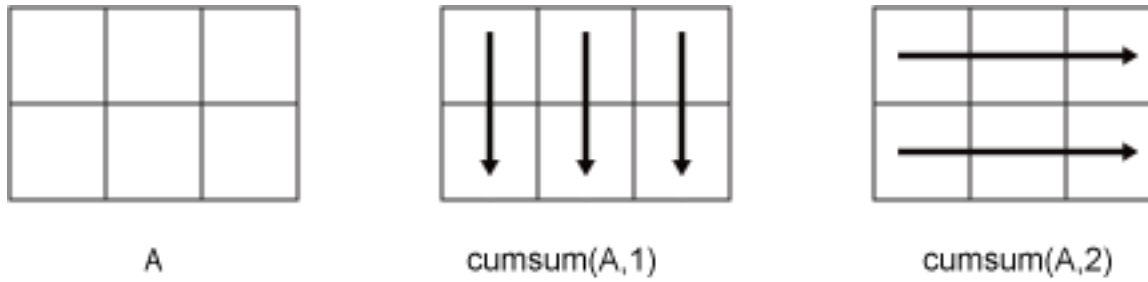
`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `cumsum(A,1)` works on successive elements in the columns of `A` and returns the cumulative sums of each column.
- `cumsum(A,2)` works on successive elements in the rows of `A` and returns the cumulative sums of each row.



`cumsum` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as the string `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Data Types: `char`

Output Arguments

`B` — Cumulative sum array
vector | matrix | multidimensional array

Cumulative sum array, returned as a vector, matrix, or multidimensional array of the same size as the input array `A`.

The class of `B` is the same as the class of `A` except if `A` is `logical`, in which case `B` is `double`.

More About

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of `X`.

Tips

- Many cumulative functions in MATLAB support the `'reverse'` option. This option allows quick directional calculations without needing a flip or reflection of the input array.

See Also

`cummax` | `cummin` | `cumprod` | `diff` | `prod` | `sum`

Introduced before R2006a

cumtrapz

Cumulative trapezoidal numerical integration

Syntax

```
Z = cumtrapz(Y)
Z = cumtrapz(X,Y)
Z = cumtrapz( ___ ,dim)
```

Description

`Z = cumtrapz(Y)` computes an approximation of the cumulative integral of `Y` via the trapezoidal method with unit spacing. To compute the integral with other than unit spacing, multiply `Z` by the spacing increment. Input `Y` can be complex.

For vectors, `cumtrapz(Y)` is a vector containing the cumulative integral of `Y`.

For matrices, `cumtrapz(Y)` is a matrix the same size as `Y` with the cumulative integral over each column.

For multidimensional arrays, `cumtrapz(Y)` works across the first nonsingleton dimension.

`Z = cumtrapz(X,Y)` computes the cumulative integral of `Y` with respect to `X` using trapezoidal integration. `X` and `Y` must be vectors of the same length, or `X` must be a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`. `cumtrapz` operates across this dimension. Inputs `X` and `Y` can be complex.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `cumtrapz(X,Y)` operates across this dimension.

`Z = cumtrapz( ___ ,dim)` integrates across the dimension of `Y` specified by scalar `dim`, using any of the input arguments in the previous syntaxes. The length of `X` must be the same as `size(Y,dim)`.

Examples

Example 1

```
Y = [0 1 2; 3 4 5];

cumtrapz(Y,1)
ans =
0         0         0
    1.5000   2.5000    3.5000

cumtrapz(Y,2)
ans =
0    0.5000    2.0000
        0    3.5000    8.0000
```

Example 2

This example uses two complex inputs:

```
z = exp(1i*pi*(0:100)/100);

ct = cumtrapz(z,1./z);
ct(end)
ans =
   0.0000 + 3.1411i
```

See Also

`cumsum` | `trapz`

Introduced before R2006a

curl

Compute curl and angular velocity of vector field

Syntax

```
[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)
[curlx,curly,curlz,cav] = curl(U,V,W)
[curlz,cav]= curl(X,Y,U,V)
[curlz,cav]= curl(U,V)
[curlx,curly,curlz] = curl(...)
[curlx,curly] = curl(...)
cav = curl(...)
```

Description

`[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)` computes the curl (`curlx`, `curly`, `curlz`) and angular velocity (`cav`) perpendicular to the flow (in radians per time unit) of a 3-D vector field `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`[curlx,curly,curlz,cav] = curl(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`[curlz,cav]= curl(X,Y,U,V)` computes the curl `z`-component and the angular velocity perpendicular to `z` (in radians per time unit) of a 2-D vector field `U`, and `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`[curlz,cav]= curl(U,V)` assumes `X` and `Y` are determined by the expression

`[X Y] = meshgrid(1:n,1:m)`

where `[m,n] = size(U)`.

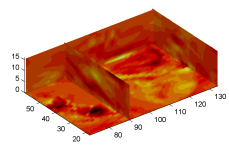
`[curlx,curly,curlz] = curl(...)`, `[curlx,curly] = curl(...)` returns only the curl.

`cav = curl(...)` returns only the curl angular velocity.

Examples

This example uses colored slice planes to display the curl angular velocity at specified locations in the vector field.

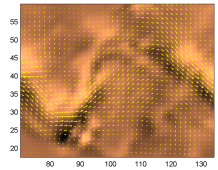
```
figure
load wind
cav = curl(x,y,z,u,v,w);
h = slice(x,y,z,cav,[90 134],59,0);
shading interp
daspect([1 1 1]);
axis tight
colormap hot(16)
camlight
set([h(1),h(2)],'ambientstrength',.6)
```



This example views the curl angular velocity in one plane of the volume and plots the velocity vectors (`quiver`) in the same plane.

```
load wind
k = 4;
x = x(:,:,k); y = y(:,:,k); u = u(:,:,k); v = v(:,:,k);
cav = curl(x,y,u,v);
pcolor(x,y,cav); shading interp
hold on;
quiver(x,y,u,v,'y')
```

```
hold off
colormap copper
```



More About

- “Displaying Curl with Stream Ribbons”

See Also

`streamribbon` | `divergence`

Introduced before R2006a

currentDirectory

Class: Tiff

Index of current IFD

Syntax

```
dirNum = currentDirectory(tiffobj)
```

Description

`dirNum = currentDirectory(tiffobj)` returns the index of the current image file directory (IFD). Index values are one-based. Use this index value with the `setDirectory` member function.

Examples

Determine Current IFD

Open a `Tiff` object and determine which IFD is the current IFD.

```
t = Tiff('example.tif','r');
dnum = currentDirectory(t)
```

```
dnum =

     1
```

Close the `Tiff` object.

```
close(t)
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFCurrentDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTiff - TIFF Library and Utilities.

See Also

`Tiff.setDirectory`

customverctrl

(To be removed) Allow custom source control system (UNIX platforms)

Note: `customverctrl` will be removed in a future release.

Syntax

```
customverctrl
```

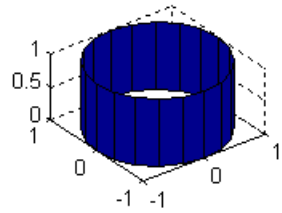
Description

`customverctrl` function is for customers who want to integrate a source control system that is not supported for use with MATLAB software. When using this function, conform to the structure of one of the supported version control systems, for example, RCS. For examples, see the files `clearcase.m`, `cvs.m`, `pvcs.m`, and `rcs.m` in *`matlabroot`*`\toolbox\matlab\verctrl`.

Introduced before R2006a

cylinder

Generate cylinder



Syntax

```
[X,Y,Z] = cylinder
[X,Y,Z] = cylinder(r)
[X,Y,Z] = cylinder(r,n)
cylinder(axes_handle,...)
cylinder(...)
```

Description

`cylinder` generates *x*-, *y*-, and *z*-coordinates of a unit cylinder. You can draw the cylindrical object using `surf` or `mesh`, or draw it immediately by not providing output arguments.

`[X,Y,Z] = cylinder` returns the *x*-, *y*-, and *z*-coordinates of a cylinder with a radius equal to `1`. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder using `r` to define a profile curve. `cylinder` treats each element in `r` as a radius at equally spaced heights along the unit height of the cylinder. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r,n)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder based on the profile curve defined by vector `r`. The cylinder has `n` equally spaced points around its circumference.

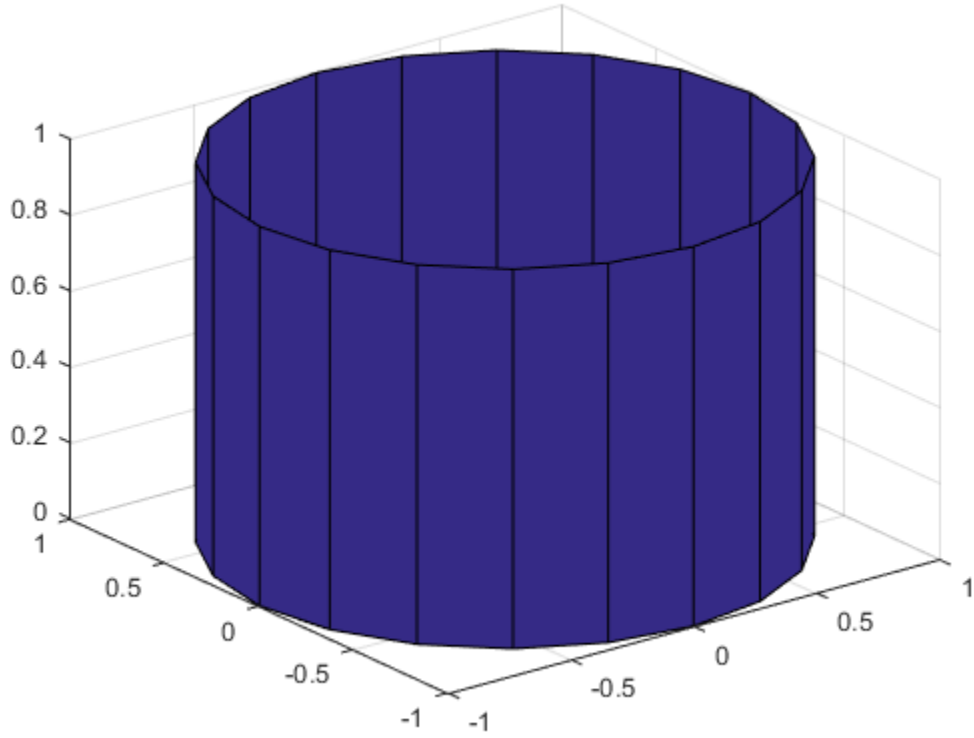
`cylinder(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`cylinder(...)`, with no output arguments, plots the cylinder using `surf`.

Examples

Display Unit Cylinder

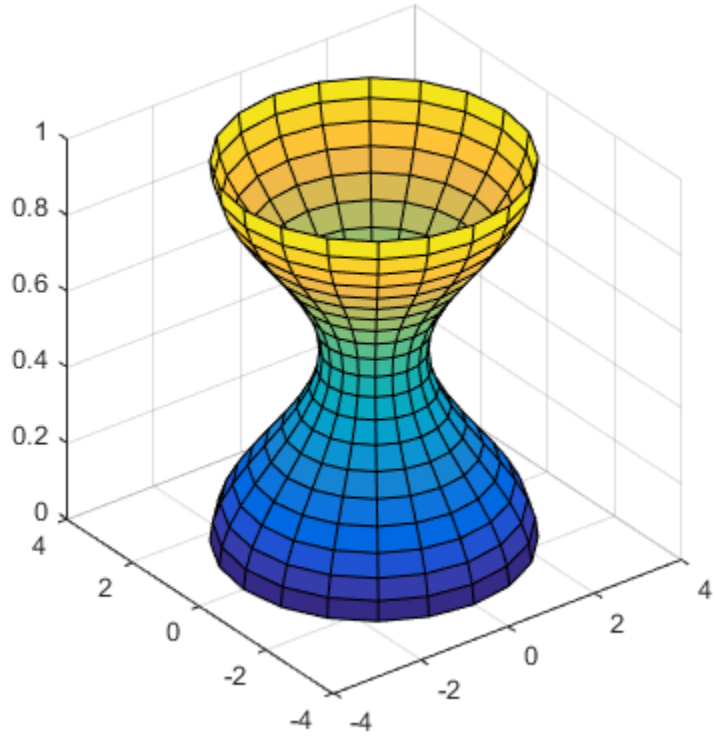
```
figure
cylinder
```



Generate Coordinates of Cylinder and Display Surface

Generate a cylinder defined by the profile function `2+sin(t)`.

```
t = 0:pi/10:2*pi;
figure
[X,Y,Z] = cylinder(2+cos(t));
surf(X,Y,Z)
axis square
```



More About

Tips

`cylinder` treats its first argument as a profile curve. The resulting surface graphics object is generated by rotating the curve about the *x*-axis, and then aligning it with the *z*-axis.

See Also

`sphere` | `surf`

Introduced before R2006a

daqread

Read Data Acquisition Toolbox (.daq) file

Syntax

```
data = daqread('filename')
[data,time] = daqread(...)
[data,time,abstime] = daqread(...)
[data,time,abstime,events] = daqread(...)
[data,time,abstime,events,daqinfo] = daqread(...)
data = daqread(...,'Param1', Val1,...)
daqinfo = daqread('filename','info')
```

Description

`data = daqread('filename')` reads all the data from the Data Acquisition Toolbox™ (`.daq`) file specified by `filename`. `daqread` returns `data`, an *m*-by-*n* data matrix, where *m* is the number of samples and *n* is the number of channels. If `data` includes data from multiple triggers, the data from each trigger is separated by a `NaN`. If you set the `OutputFormat` property to `tscollection`, `daqread` returns a time series collection object. See below for more information.

`[data,time] = daqread(...)` returns time/value pairs. `time` is an *m*-by-1 vector, the same length as `data`, that contains the relative time for each sample. Relative time is measured with respect to the first trigger that occurs.

`[data,time,abstime] = daqread(...)` returns the absolute time of the first trigger. `abstime` is returned as a `clock` vector.

`[data,time,abstime,events] = daqread(...)` returns a log of events. `events` is a structure containing event information. If you specify either the`Samples`, `Time`, or `Triggers` parameters (see below), the events structure contains only the specified events.

`[data,time,abstime,events,daqinfo] = daqread(...)` returns a structure, `daqinfo`, that contains two fields: `ObjInfo` and `HwInfo`. `ObjInfo` is a structure containing property name/property value pairs and `HwInfo` is a

structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

`data = daqread(...,'Param1', Val1,...)` specifies the amount of data returned and the format of the data, using the following parameters.

Parameter	Description
`Samples`	Specify the sample range.
`Time`	Specify the relative time range.
`Triggers`	Specify the trigger range.
`Channels`	Specify the channel range. Channel names can be specified as a cell array.
`DataFormat`	Specify the data format as `doubles` (default) or `native`.
`TimeFormat`	Specify the time format as `vector` (default) or `matrix`.
`OutputFormat`	Specify the output format as `matrix` (the default) or `tscollection`. When you specify `tscollection`, `daqread` only returns `data`.

The `Samples`, `Time`, and `Triggers` properties are mutually exclusive; that is, either `Samples`, `Triggers` or `Time` can be defined at once.

`daqinfo = daqread('filename','info')` returns metadata from the file in the `daqinfo` structure, without incurring the overhead of reading the data from the file as well. The `daqinfo` structure contains two fields:

`daqinfo.ObjInfo`

a structure containing parameter/value pairs for the data acquisition object used to create the file, `filename`. Note: The `UserData` property value is not restored.

`daqinfo.HwInfo`

a structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

Examples

Use Data Acquisition Toolbox to acquire data. The analog input object, `ai`, acquires one second of data for four channels, and saves the data to the output file `data.daq`.

```
ai = analoginput('nidaq','Dev1');
chans = addchannel(ai,0:3);
set(ai,'SampleRate',1000)
ActualRate = get(ai,'SampleRate');
set(ai,'SamplesPerTrigger, ActualRate)
set(ai,'LoggingMode','Disk&Memory')
set(ai,'LogFileName','data.daq')
start(ai)
```

After the data has been collected and saved to a disk file, you can retrieve the data and other acquisition-related information using `daqread`. To read all the sample-time pairs from `data.daq`:

```
[data,time] = daqread('data.daq');
```

To read samples 500 to 1000 for all channels from `data.daq`:

```
data = daqread('data.daq','Samples',[500 1000]);
```

To read only samples 1000 to 2000 of channel indices 2, 4 and 7 in native format from the file, `data.daq`:

```
data = daqread('data.daq', 'Samples', [1000 2000],...
               'Channels', [2 4 7], 'DataFormat', 'native');
```

To read only the data which represents the first and second triggers on all channels from the file, `data.daq`:

```
[data,time] = daqread('data.daq', 'Triggers', [1 2]);
```

To obtain the channel property information from `data.daq`:

```
daqinfo = daqread('data.daq','info');
chaninfo = daqinfo.ObjInfo.Channel;
```

To obtain a list of event types and event data contained by `data.daq`:

```
daqinfo = daqread('data.daq','info');
events = daqinfo.ObjInfo.EventLog;
event_type = {events.Type};
event_data = {events.Data};
```

To read all the data from the file `data.daq` and return it as a time series collection object:

```
data = daqread('data.daq','OutputFormat','tscollection');
```

More About

Tips

More About .daq Files

- The format used by `daqread` to return data, relative time, absolute time, and event information is identical to the format used by the `getdata` function that is part of Data Acquisition Toolbox. For more information, see the Data Acquisition Toolbox documentation.
- If data from multiple triggers is read, then the size of the resulting data array is increased by the number of triggers issued because each trigger is separated by a `NaN`.
- `ObjInfo.EventLog` always contains the entire event log regardless of the value specified by `Samples`, `Time`, or `Triggers`.
- The `UserData` property value is not restored when you return device object (`ObjInfo`) information.
- When reading a `.daq` file, the `daqread` function does not return property values that were specified as a cell array.
- Data Acquisition Toolbox (`.daq`) files are created by specifying a value for the `LogFileName` property (or accepting the default value), and configuring the `LoggingMode` property to `Disk` or `Disk&Memory`.

More About Time Series Collection Object Returned

When `OutputFormat` is set to `tscollection`, `daqread` returns a time series collection object. This times series collection object contains an absolute time series object for each channel in the file. The following describes how `daqread` sets some of the properties of the times series collection object and the time series objects.

- The `time` property of the time series collection object is set to the value of the `InitialTriggerTime` property specified in the file.
- The `name` property of each time series object is set to the value of the `Name` property of a channel in the file. If this name cannot be used as a time series object name, `daqread` sets the name to `'Channel'` with the `HwChannel` property of the channel appended.
- The value of the `Units` property of the time series object depends on the value of the `DataFormat` parameter. If the `DataFormat` parameter is set to `'double'`, `daqread`

sets the `DataInfo` property of each time series object in the collection to the value of the `Units` property of the corresponding channel in the file. If the `DataFormat` parameter is set to `'native'`, `daqread` sets the `Units` property to `'native'`. See the Data Acquisition Toolbox documentation for more information on these properties.

- Each time series object will have `tsdata.event` objects attached corresponding to the log of events associated with the channel.

If `daqread` returns data from multiple triggers, the data from each trigger is separated by a `NaN` in the time series data. This increases the length of data and time vectors in the time series object by the number of triggers.

See Also

`timeseries` | `tscollection`

daspect

Set or query axes data aspect ratio

Syntax

```
daspect
daspect([aspect_ratio])
daspect('mode')
daspect('auto')
daspect('manual')
daspect(axes_handle,...)
```

Description

The data aspect ratio determines the relative scaling of the data units along the *x*-, *y*-, and *z*-axes.

`daspect` with no arguments returns the data aspect ratio of the current axes.

`daspect([aspect_ratio])` sets the data aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the *x*-, *y*-, and *z*-axis scaling (e.g., `[1 1 3]` means one unit in *x* is equal in length to one unit in *y* and three units in *z*).

`daspect('mode')` returns the current value of the data aspect ratio mode, which can be either `auto` (the default) or `manual`. See Tips.

`daspect('auto')` sets the data aspect ratio mode to `auto`.

`daspect('manual')` sets the data aspect ratio mode to `manual`.

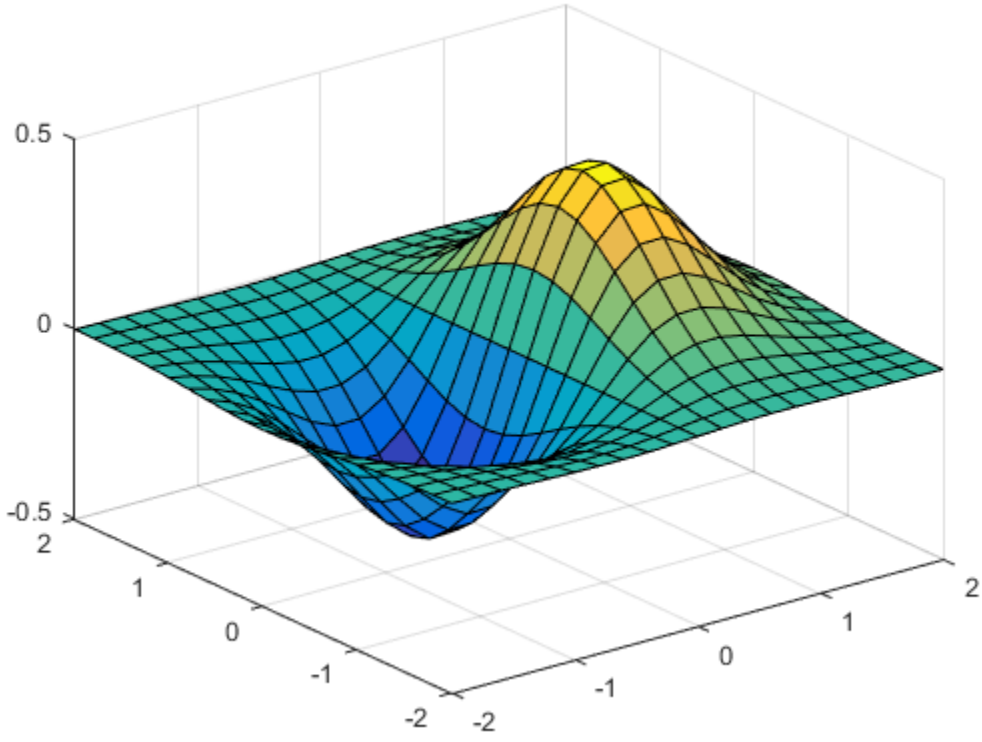
`daspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `daspect` operates on the current axes.

Examples

Equal Scaling Along Each Axis

Plot the function $z = xe^{(-x^2-y^2)}$ over the range $-2 \le x \le 2$ and $-2 \le y \le 2$.

```
[x,y] = meshgrid(-2:.2:2);
z = x.*exp(-x.^2 - y.^2);

figure
surf(x,y,z)
```

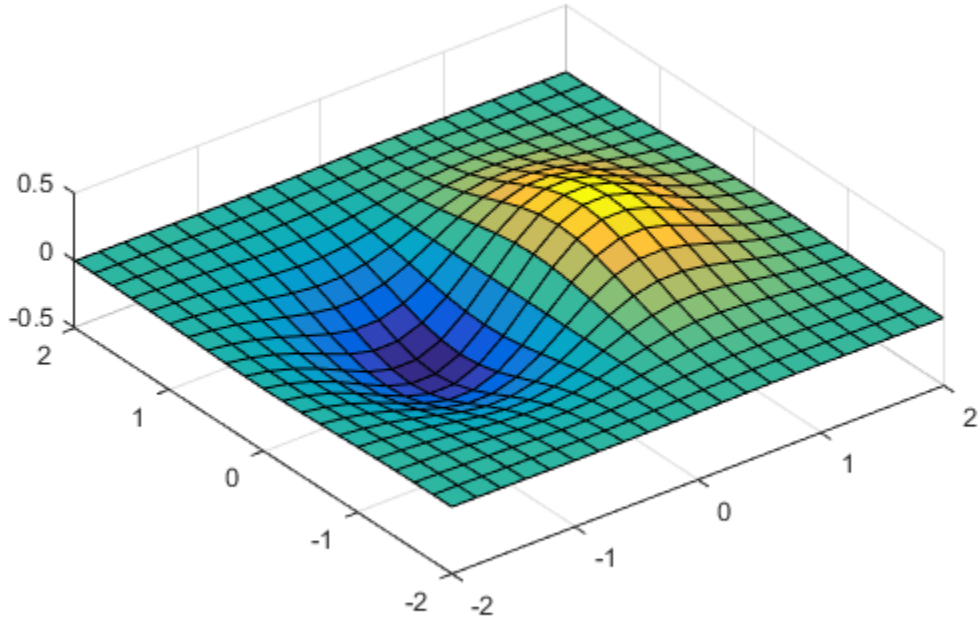


Query the data aspect ratio to show the axis scaling.

```
daspect

ans =

     4     4     1
```

Use equal scaling along each axis by setting the data aspect ratio to `[1,1,1]`.

```
daspect([1,1,1])
```



More About

Tips

`daspect` sets or queries values of the axes object `DataAspectRatio` and `DataAspectRatioMode` properties.

When the data aspect ratio mode is `auto`, the data aspect ratio adjusts so that each axis spans the space available in the figure window. If you are displaying a representation of a real-life object, you should set the data aspect ratio to `[1 1 1]` to produce the correct proportions.

Setting a value for data aspect ratio or setting the data aspect ratio mode to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the data aspect ratio to a value, including its current value,

```
daspect(daspect)
```

can cause a change in the way the graphs look. See the Stretch-to-Fill section of the `axes` description for more information.

See Also

`axis` | `pbaspect` | `xlim` | `ylim` | `zlim`

Introduced before R2006a

datacursormode

Enable, disable, and manage interactive data cursor mode

Syntax

```
datacursormode on
datacursormode off
datacursormode
datacursormode toggle
datacursormode(figure_handle)
dcm_obj = datacursormode(figure_handle)
```

Description

`datacursormode on` enables data cursor mode on the current figure.

`datacursormode off` disables data cursor mode on the current figure.

`datacursormode` or `datacursormode toggle` toggles data cursor mode in the current figure.

`datacursormode(figure_handle)` enables or disables data cursor mode on the specified figure.

`dcm_obj = datacursormode(figure_handle)` returns the data cursor mode object for the figure. The object enables you to customize the data cursor. For more information on data cursor mode objects, see “Output Arguments” on page 1-1848. You cannot change the state of data cursor mode in a call to `datacursormode` that returns a mode object.

A *data cursor* is a small black square with a white border that you interactively position on a graph in data cursor mode. When you click a graphic object such as a line on a graph, a *data tip* appears. Data tips are small text boxes or windows that float within an axes that display data values at data cursor locations. The default style is a text box. Data tips list *x*-, *y*- and (where appropriate) *z*-values for one data point at a time. See “Examples” on page 1-1851 for an illustration of these two styles.

Input Arguments

`figure_handle`

Optional handle of figure window

Default: The current figure

`state`

`''`, `'toggle'`, `'on'`, or `'off'`

Default: `'toggle'`

Output Arguments

`dcm_obj`

Use the object returned by `datacursormode` to control aspects of data cursor behavior. You can use the `set` and `get` commands to set and query object property values. You can customize how data cursor mode presents information by coding callback functions for these objects.

Parameter Name/Value Pairs for Data Cursor Mode Objects

The following parameters apply to objects returned by calls to `datacursormode`, not to the function itself.

`'DisplayStyle'`

`datatip | window`

Determines how the data cursor displays.

- `datatip` displays data cursor information in a small yellow text box attached to a black square marker at a data point you interactively select.
- `window` displays data cursor information for the data point you interactively select in a floating window within the figure.

Default: `datatip`

`'Enable'`

`on | off`

Specifies whether data cursor mode is currently enabled for the figure.

Default: `off`

`'Figure'`

handle

Handle of the figure associated with the data cursor mode object.

`'SnapToDataVertex'`

`on | off`

Specifies whether the data cursor snaps to the nearest data value or is located at the actual pointer position.

Default: `on`

`'UpdateFcn'`

function handle

Reference to a function that formats the text appearing in the data cursor. You can supply your own function to customize data tip display. Your function must include at least two arguments. The first argument is unused, and can be a variable name or tilde (~). The second argument passes the data cursor event object to your update function. The event object encapsulates the state of the data cursor. The following function definition illustrates the update function:

```
function output_txt = myfunction(~,event_obj)
% ~            Currently not used (empty)
% event_obj    Object containing event data structure
% output_txt   Data cursor text (string or cell array
%              of strings)
```

`event_obj` is an object that has the following properties.

`Target`	Handle of the object the data cursor is referencing (the object which you click, for example, a line or a bar from a series)
`Position`	An array specifying the *x*, *y* (and *z* for 3-D graphs) coordinates of the cursor

You can query these properties within your function. For example,

```
pos = get(event_obj,'Position');
```

returns the coordinates of the cursor. Another way of accessing that data is to obtain the struct and query its `Position` field:

```
eventdata = get(event_obj);
pos = eventdata.Position;
```

You can also obtain the position directly from the object:

```
pos = event_obj.Position;
```

You can redefine the data cursor `Updatefcn` at run time. For example:

```
set(dcm_obj,'UpdateFcn',@myupdatefcn)
```

applies the function `myupdatefcn` to the current data tip or tips. When you set an update function in this way, the function must be on the MATLAB path. If instead you select the data cursor mode context menu item **Select text update function**, you can interactively select a function that is not on the path.

Do not redefine figure window callbacks, such as `ButtonDownFcn`, `KeyPpressFcn`, or `CloseRequestFcn` *while in data cursor mode*. If you attempt to change any *figure* callbacks when you are in an interactive mode, you receive a warning and the attempt fails. MATLAB interactive modes are:

- `brush`
- `datacursormode`
- `pan`
- `rotate3d`
- `zoom`

This restriction does not apply to changing the figure `WindowButtonMotionFcn` callback or `uicontrol` callbacks.

Querying Data Cursor Mode

Use the `getCursorInfo` function to query the data cursor mode object (`dcm_obj` in the update function syntax) to obtain information about the data cursor. For example,

```
info_struct = getCursorInfo(dcm_obj);
```

returns a vector of structures, one for each data cursor on the graph. Each structure has the following fields.

`Target`	The handle of the graphics object containing the data point
`Position`	An array specifying the *x*, *y*, (and *z*) coordinates of the cursor

Line and lineseries objects have an additional field.

`DataIndex`	A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

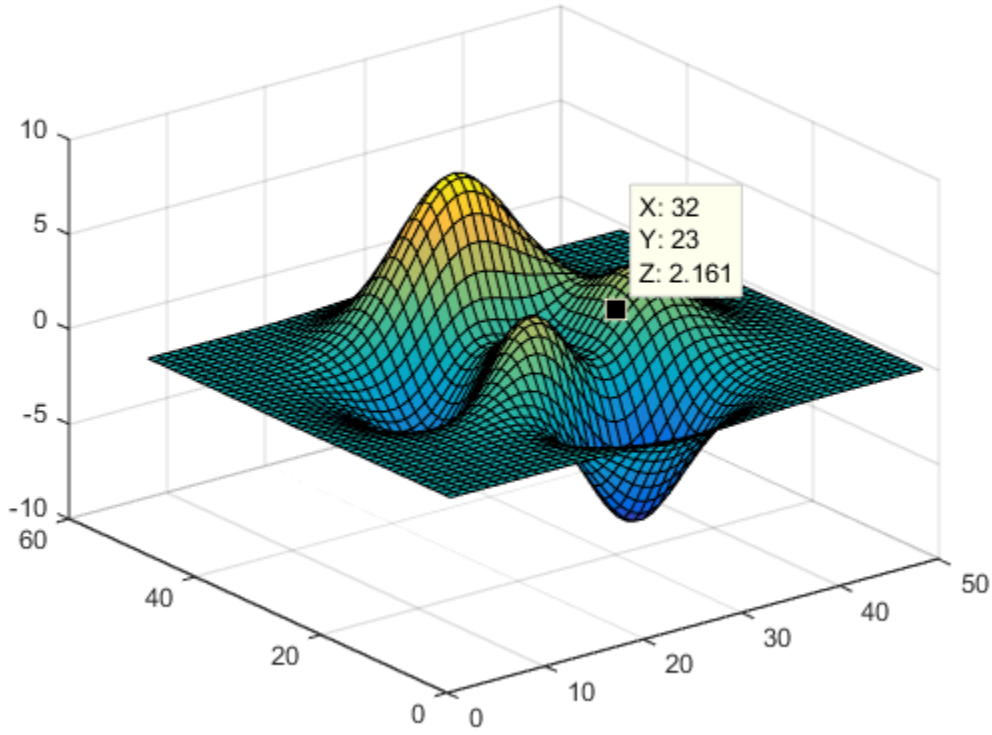
See “Output Arguments” on page 1-1848 for more details on data cursor mode objects.

Examples

This example creates a plot and enables data cursor mode from the command line.

```
surf(peaks)
datacursormode on
% Click mouse on surface to display data cursor
```

Selecting a point on the surface opens a data tip displaying its *x*-, *y*-, and z-coordinates.



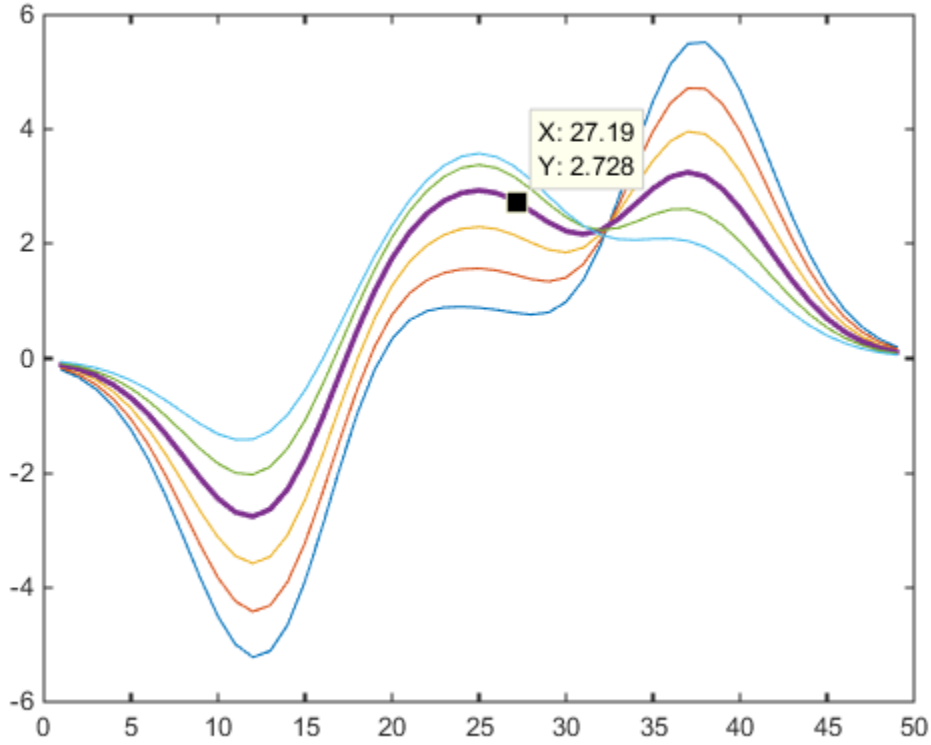
You change the data tip display style to be a window instead of a text box using the **Tools > Options > Display cursor in window** , or use the context menu **Display Style > Window inside figure** to view the data tip in a floating window that you can move around inside the axes.

You can position multiple text box data tips on the same graph, the window style of data tip displays only one value at a time. For more information on interacting with data cursors, including point selection options and exporting data tips to the workspace, see “Display Data Values Interactively”.

This example enables data cursor mode on the current figure and sets data cursor mode options. The following statements

- Create a graph
- Toggle data cursor mode to on
- Obtain the data cursor mode object, specify data tip options, and get the handle of the line the data tip occupies:

```
fig = figure;
z = peaks;
plot(z(:,30:35))
dcm_obj = datacursormode(fig);
set(dcm_obj,'DisplayStyle','datatip',...
    'SnapToDataVertex','off','Enable','on')

disp('Click line to display a data tip, then press Return.')
% Wait while the user does this.
pause

c_info = getCursorInfo(dcm_obj);
% Make selected line wider
set(c_info.Target,'LineWidth',2)
```



This example shows you how to customize the text that the data cursor displays. For example, you can replace the text displayed in the data tip and data window (`x:` and `y:`) with `Time:` and `Amplitude:` by creating a simple update function.

Save the following functions in your current directory or any writable directory on the MATLAB path before running them. As they are functions, you cannot highlight them and then evaluate the selection to make them work.

Save this code as `doc_datacursormode.m`:

```
function doc_datacursormode
% Plots graph and sets up a custom data tip update function
fig = figure;
```

```
a = -16; t = 0:60;
plot(t,sin(a*t))
dcm_obj = datacursormode(fig);
set(dcm_obj,'UpdateFcn',@myupdatefcn)
```

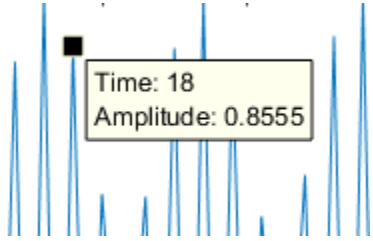
Save the following code as `myupdatefcn.m` on the MATLAB path:

```
function txt = myupdatefcn(empt,event_obj)
% Customizes text of data tips

pos = get(event_obj,'Position');
txt = {['Time: ',num2str(pos(1))],...
       ['Amplitude: ',num2str(pos(2))]};
```

To set up and use the update function, type:

```
doc_datacursormode
```

When you place a data tip using this update function, it looks like the one in the following figure.



Alternatives

Use the Data Cursor tool  to label *x*, *y*, and *z* values on graphs and surfaces. You can control how data tips display by right-clicking and selecting items from the context menu.

More About

Tips

- Most types of graphs and 3-D plots support data cursor mode, but several do not (`pareto`, for example).

- Polar plots support data tips, but display Cartesian rather than polar coordinates on them.
- Histograms created with `histogram` display specialized data tips that itemize the observation counts and bin edges.
- You place data tips only by clicking data objects on graphs. You cannot place them programmatically (by executing code to position a data cursor).
- When `DisplayStyle` is `datatip`, you can place multiple data tips on a graph. When `DisplayStyle` is `window`, it reports only the most recent data tip.
- `datacursormode off` exits data cursor mode but does not remove displayed data tips. However, if the `DisplayStyle` is `window`, the data tip window goes away.

- "Example — Visually Exploring Demographic Statistics"
- "Data Cursors with Histograms"

See Also

`brush` | `pan` | `rotate3d` | `zoom`

Introduced before R2006a

datastore

Create datastore to access collection of data

Syntax

```
ds = datastore(location)
ds = datastore(location,Name,Value)
```

Description

`ds = datastore(location)` creates a datastore from the collection of data specified by `location`. A datastore is a repository for collections of data that are too large to fit in memory. After creating `ds`, you can read the data.

`ds = datastore(location,Name,Value)` specifies the properties of `ds` using one or more name-value pair arguments. For example, you can specify the type of datastore to create as a `TabularTextDatastore` or a `KeyValueDatastore`.

Examples

Create Datastore for Text Data

Create a datastore associated with the sample file, `airlinesmall.csv`. This file contains airline data from the years 1987 through 2008.

Use the `'TreatAsMissing'` name-value pair argument to specify the strings in the file to treat as missing data.

```
ds = datastore('airlinesmall.csv',...
    'TreatAsMissing','NA')
```

```
ds =

  TabularTextDatastore with properties:
```

```
                    Files: {
                           ' .../matlab/toolbox/matlab/demos/airlinesmall.csv'
                           }
            ReadVariableNames: true
                VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

  Text Format Properties:
               NumHeaderLines: 0
                    Delimiter: ','
                  RowDelimiter: '\r\n'
               TreatAsMissing: 'NA'
                 MissingValue: NaN

  Advanced Text Format Properties:
             TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
           ExponentCharacters: 'eEdD'
                 CommentStyle: ''
                   Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
```

`datastore` creates a `TabularTextDatastore`.

Input Arguments

`location` — Files or folders to include in the datastore

string | cell array of strings

Files or folders to include in the datastore, specified as a string or cell array of strings. If the files are not in the current folder, then `files` must include full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You must indicate the paths to these subfolders in `location`.

If the files are not available locally, then the full path of the files or folders must be an internationalized resource identifier (IRI) of the form `hdfs://`*`hostname`*`:`*`portnumber`*`/`*`path_to_file`*.

Before reading from HDFS™, you must set the `HADOOP_HOME`, `HADOOP_PREFIX`, or `MATLAB_HADOOP_INSTALL` environment variable to the folder where Hadoop® is installed. For more information, see "Read from HDFS".

Note: When reading from HDFS or when reading Sequence files locally, the `datastore` function calls the `javaaddpath` command. This command clears the definitions of all Java classes defined by files on the dynamic class path, removes all global variables and variables from the base workspace, and removes all compiled scripts, functions, and MEX-functions from memory. To prevent persistent variables, code files or MEX-files from being cleared, use the `mlock` function.

For `TabularTextDatastore` and `KeyValueDatastore`, you can use the wildcard character (*) to indicate that all matching files or all files in the matching folders are included in the datastore.

For `KeyValueDatastore`, the files must be MAT-files or Sequence files generated by the `mapreduce` function, and they must be in a local file system or in a network file system.

Example: `'file1.csv'`

Example: `'../dir/data/file1.csv'`

Example: `{'C:\dir\data\file1.csv','C:\dir\data\file2.csv'}`

Example: `'C:\dir\data\*.mat'`

Example: `'hdfs://myserver:7867/data/file1.txt'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Delimiter',','` specifies a comma delimiter for a `TabularTextDatastore` object.

All Datastore Types

`'DatastoreType'` — Type of datastore

`'tabulartext' | 'keyvalue'`

Type of datastore, specified as the comma-separated pair consisting of `'DatastoreType'` and one of the following strings.

Value of dsType	Description
`'tabulartext'`	Interpret the data set specified by `location` as text files containing tabular data. The encoding of the data must be ASCII or UTF-8.
`'keyvalue'`	Interpret the data set specified by `location` as key-value pair data. The files can be MAT-files or sequence files with key-value pair data generated by `mapreduce`.

Other datastore types might be supported if you have other MathWorks products installed. In this case, other values for `'DatastoreType'` are valid.

Use the `'DatastoreType'` argument to specify a datastore type when there are multiple types that support the format of the files.

If you do not specify a value for `DatastoreType`, then `datastore` automatically determines the appropriate type of datastore to create, based on the extension of the files specified by the `location` argument.

`TabularTextDatastore` Only

In addition to the following, you can specify other `TabularTextDatastore` properties using name-value pair arguments. See TabularTextDatastore Properties.

`'VariableNames'` — Names of variables
cell array of strings

Names of variables in the datastore, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of strings. Specify the variable names in the order in which they appear in the files. If you do not specify the variable names, they are detected from the first nonheader line in the first file in the datastore. You can rename the variables after creating the datastore, by modifying its `VariableNames` property.

When `ReadVariableNames` is `false`, then `VariableNames` defaults to `{'Var1','Var2', ...}`.

Example: `'VariableNames',{'Time','Name','Quantity'}`

`'TextscanFormats'` — Format of the data fields

string | cell array of strings

Format of the data fields, specified as the comma-separated pair consisting of `'TextscanFormats'` and a string or a cell array of strings. If the value of `TextscanFormats` is a string, then it can contain one or more conversion specifiers. If the value of `TextscanFormats` is a cell array of strings, then each string must contain only one conversion specifier. You can use the same conversion specifiers that the `textscan` function accepts.

If the value of `TextscanFormats` includes conversion specifiers that skip fields using asterisk characters (*), then the value of the `SelectedVariableNames` property automatically updates. For more information, see the `TextscanFormats` property.

If you do not specify a value for `TextscanFormats`, then `datastore` determines the format of the data fields by scanning text from first nonheader, nonvariable name line in the first file.

Example: `'TextscanFormats','%s%s%f'`

Example: `'TextscanFormats',{'%s','%s','%f'}`

`'SelectedVariableNames'` — Variables to read

cell array of strings

Variables to read, specified as the comma-separated pair consisting of `'SelectedVariableNames'` and a cell array of strings, where each string contains the name of one variable. You can specify the variable names in any order.

Example: `'SelectedVariableNames',{'Var3','Var7','Var4'}`

`'SelectedFormats'` — Formats for selected variables

cell array of strings

Formats for the selected variables, specified as the comma-separated pair consisting of `'SelectedFormats'` and a cell array of strings, where each string contains one conversion specifier. Use the `SelectedVariableNames` property to indicate the variables to read. The number of strings in `SelectedFormats` must match the number of variables to read.

You can use the same conversion specifiers that the `textscan` function accepts. However, you cannot use a conversion specifier that skips a field. For example, the conversion specifier cannot include an asterisk character (*).

Example: `'SelectedFormats',{'%d','%d'}`

`KeyValueDatastore` Only

You can specify `KeyValueDatastore` properties using name-value pair arguments. See `KeyValueDatastore`.

Output Arguments

`ds` — Datastore for collection of files

`TabularTextDatastore | KeyValueDatastore`

Datastore for a collection of files, returned as a `TabularTextDatastore` or a `KeyValueDatastore`. The type of the datastore depends on the type of files in the datastore. For more information, click the output datastore name.

Type of Files	Output
Text files	`TabularTextDatastore`
MAT-files or Sequence files	`KeyValueDatastore`

For each of the above datastore types, the `Files` property is a cell array of strings, where each string is an absolute path to a file resolved from the `location` argument.

More About

- "Getting Started with Datastore"
- "Read from HDFS"
- Using DatabaseDatastore Objects

See Also

`javaaddpath | mapreduce`

Introduced in R2014b

hasdata

Determine if data available to read

Syntax

```
tf = hasdata(ds)
```

Description

`tf = hasdata(ds)` returns logical `1` (`true`) if there is data available to read from the datastore specified by `ds`. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Data is Available to Read

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

While there is data available in the datastore, read the data.

```
while hasdata(ds)
    T = read(ds);
end
```

Input Arguments

`ds` — Input datastore

datastore

Input datastore. Use the `datastore` function to create a datastore object from your data.

See Also

`datastore`

Introduced in R2014b

Using KeyValueDatastore Objects

Datastore for key-value pair data

`KeyValueDatastore` objects are associated with files containing key-value pair data that are outputs of or inputs to `mapreduce`. Use the `KeyValueDatastore` properties to specify how you want to access the data. Use dot notation to view or modify a particular property of a `KeyValueDatastore` object:

```
ds = datastore('mapredout.mat');
ds.ReadSize = 20;
```

You also can specify the value of `KeyValueDatastore` properties using name-value pair arguments when you create a datastore using the `datastore` function:

```
ds = datastore('mapredout.mat','ReadSize',20);
```

Examples

Set Number of Key-Value Pairs to Read

Create a datastore from the sample file, `mapredout.mat`, which is an output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat')
```

```
ds =
  KeyValueDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\demos\mapredout.mat'
              }
    ReadSize: 1 key-value pairs
    FileType: 'mat'
```

Set the `ReadSize` property to 8 so that each call to `read` reads at most 8 key-value pairs.

```
ds.ReadSize = 8
```

```
ds =
  KeyValueDatastore with properties:

       Files: {
```

```
            ' ...\matlab\toolbox\matlab\demos\mapredout.mat'
            }
    ReadSize: 8 key-value pairs
    FileType: 'mat'
```

Read 8 key-value pairs at a time using the `read` function in a `while` loop. The loop executes until there is no more data available to read and `hasdata(ds)` returns `false`.

```
while hasdata(ds)
    T = read(ds);
end
```

Show the last set of key-value pairs read.

```
T
```

```
T =

    Key     Value
    ____    ______

    '0O'    [3090]
    'TZ'    [ 216]
    'XE'    [2357]
    '9E'    [ 521]
    'YV'    [ 849]
```

- "Read and Analyze MAT-File with Key-Value Data"
- "Read and Analyze Hadoop Sequence File"

Properties

`Files` — Files included in datastore

cell array of strings

Files included in the datastore, specified as a cell array of strings, where each string is a full path to a file. These are the files defined by the `location` argument to the `datastore` function.

The files must be either MAT-files or Sequence files generated by the `mapreduce` function. The files must be in a local file system or in a network file system. Sequence files also can be in HDFS.

Example: `{'C:\dir\data\file1.mat','C:\dir\data\file2.mat'}`

`FileType` — File type

`'mat'` (default) | `'seq'`

File type, specified as either `'mat'` for MAT-files or `'seq'` for sequence files. By default, the output of `mapreduce` running against Hadoop is a datastore containing sequence files. By default, the output of all other `mapreduce` operations is a datastore containing MAT-files.

`ReadSize` — Maximum number of key-value pairs to read

1 (default) | positive integer

Maximum number of key-value pairs to read in a call to the `read` or `preview` functions, specified as a positive integer.

`KeyValueLimit` — Maximum number of key-value pairs to read

1 (default) | positive integer

Note: `KeyValueLimit` will be removed in a future release. Use `ReadSize` instead.

Maximum number of key-value pairs to read in a call to the `read` or `preview` functions, specified as a positive integer.

Object Functions

`hasdatanumpartitions` `partition` `preview` `read` `readall` `reset`

Create Object

Create `KeyValueDatastore` objects using the `datastore` function.

See Also

`datastore` | `mapreduce`

numpartitions

Number of partitions

Syntax

```
N = numpartitions(ds)
N = numpartitions(ds,pool)
```

Description

`N = numpartitions(ds)` returns the default number of partitions for datastore `ds`.

`N = numpartitions(ds,pool)` returns a number of partitions to parallelize datastore access over the parallel pool specified by `pool`. To parallelize datastore access, you must have Parallel Computing Toolbox™ installed.

Examples

Number of Partitions

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Get the default number of partitions.

```
N = numpartitions(ds)
```

```
N =
     1
```

By default, there is only one partition in `ds` because it contains only one small file.

Partition the datastore and return the datastore corresponding to the first part.

```
subds = partition(ds,N,1);
```

Read the data in `subds`.

```
while hasdata(subds)
 data = read(subds);
end
```

Number of Partitions for Parallel Datastore Access

Get a number of partitions to parallelize datastore access over the current parallel pool. You must have Parallel Computing Toolbox installed.

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Get a number of partitions to parallelize datastore access over the current parallel pool.

```
N = numpartitions(ds, gcp);
```

Partition the datastore and read the data in each part.

```
parfor ii=1:N
    subds = partition(ds,N,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

- "Partition a Datastore in Parallel"

Input Arguments

`ds` — Input datastore

datastore

Input datastore. Use the `datastore` function to create a datastore object from your data.

`pool` — Parallel pool

parallel pool object

Parallel pool object.

Example: `gcp`

See Also

`datastore` | `partition`

Introduced in R2015a

partition

Partition a datastore

Syntax

```
subds = partition(ds,N,index)

subds = partition(ds,'Files',index)
subds = partition(ds,'Files',filename)
```

Description

`subds = partition(ds,N,index)` partitions datastore `ds` into the number of parts specified by `N` and returns the partition corresponding to the index `index`.

`subds = partition(ds,'Files',index)` partitions the datastore by files and returns the partition corresponding to the file of index `index` in the `Files` property.

`subds = partition(ds,'Files',filename)` partitions the datastore by files and returns the partition corresponding to the file specified by `filename`.

Examples

Partition Datastore into Specific Number of Parts

Create a datastore from the sample file, `airlinesmall.csv`, which contains tabular data.

```
ds = datastore('airlinesmall.csv');
```

Partition the datastore into 3 parts.

```
subds = partition(ds,3,1)
```

```
subds =
  TabularTextDatastore with properties:
```

```
                    Files: {
                           ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                           }
            ReadVariableNames: true
                VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

  Text Format Properties:
               NumHeaderLines: 0
                    Delimiter: ','
                RowDelimiter: '\r\n'
               TreatAsMissing: ''
                 MissingValue: NaN

  Advanced Text Format Properties:
             TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
           ExponentCharacters: 'eEdD'
                 CommentStyle: ''
                   Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
```

Partition Datastore into Default Number of Parts

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Get the default number of partitions for `ds`.

```
N = numpartitions(ds);
```

Partition the datastore into the default number of partitions and return the datastore corresponding to the first partition.

```
subds = partition(ds,N,1);
```

Read the data in `subds`.

```
while hasdata(subds)
 data = read(subds);
```

```
end
```

Partition Datastore by Files

Create a datastore that contains ten instances of the sample file, `airlinesmall.csv`.

```
ds = datastore(repmat({'airlinesmall.csv'}, 1, 10))
```

```
ds =
  TabularTextDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv',
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv',
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                              ... and 7 more
                             }
          ReadVariableNames: true
              VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

  Text Format Properties:
             NumHeaderLines: 0
                  Delimiter: ','
               RowDelimiter: '\r\n'
             TreatAsMissing: ''
               MissingValue: NaN

  Advanced Text Format Properties:
            TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
        ExponentCharacters: 'eEdD'
              CommentStyle: ''
                Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
```

Partition the datastore by files and return the part corresponding to the first file.

```
subds = partition(ds,'Files',1)
```

```
subds =
  TabularTextDatastore with properties:
```

```
                    Files: {
                           ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                           }
          ReadVariableNames: true
              VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

  Text Format Properties:
             NumHeaderLines: 0
                  Delimiter: ','
               RowDelimiter: '\r\n'
             TreatAsMissing: ''
               MissingValue: NaN

  Advanced Text Format Properties:
            TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
         ExponentCharacters: 'eEdD'
               CommentStyle: ''
                 Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
```

`subds` contains one file.

Partition Data in Parallel

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Partition the datastore into three parts on three workers in a parallel pool.

```
N = 3;
p = parpool('local',N);

parfor ii=1:N
    subds = partition(ds,N,ii);
    while hasdata(subds)
        data = read(subds);
    end
```

```
end
```

- “Partition a Datastore in Parallel”

Input Arguments

`ds` — Input datastore
datastore

Input datastore. Use the `datastore` function to create a datastore object from your data.

`N` — Number of partitions
positive integer

Number of partitions, specified as a positive integer.

Example: `3`

Data Types: `double`

`index` — Index
positive integer

Index, specified as a positive integer.

Example: `1`

Data Types: `double`

`filename` — file name
string

File name, specified as a string.

Example: `'file1.csv'`

Example: `'../dir/data/file1.csv'`

Example: `'hdfs://myserver:7867/data/file1.txt'`

Output Arguments

`subds` — Output datastore
datastore

Output datastore. The output datastore is of the same type as the input datastore `ds`.

See Also

`datastore` | `numpartitions`

Introduced in R2015a

preview

Return subset of data from datastore

Syntax

```
data = preview(ds)
```

Description

`data = preview(ds)` returns a subset of data from datastore `ds` without changing its current position.

Examples

Preview Data in TabularTextDatastore

Create a datastore from the sample file, `airlinesmall.csv`, which contains tabular data.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA');
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'DepTime','ArrTime','ActualElapsedTime'};
```

Preview the data for the selected variables.

```
data = preview(ds)
```

```
data =

    DepTime    ArrTime    ActualElapsedTime
    _______    _______    _________________

     642        735        53
    1021       1124        63
```

```
    2055        2218          83
    1332        1431          59
     629         746          77
    1446        1547          61
     928        1052          84
     859        1134         155
```

Preview Data in KeyValueDatastore

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Preview the data in the datastore.

```
data = preview(ds)
```

```
data =

    Key       Value
    ____    ________

    'AA'    [14930]
```

Input Arguments

`ds` — Input datastore

datastore

Input datastore. Use the `datastore` function to create a datastore object from your data.

Output Arguments

`data` — Subset of data

table

Subset of data, returned as a table.

Type of data store	Description
`TabularTextDatastore`	Table of the variables specified by the `SelectedVariableNames` property of the datastore. The table contains at most 8 rows.
`KeyValueDatastore`	Table with the variables `Key` and `Value`.

See Also

`datastore` | `hasdata`

Introduced in R2014b

read

Read data in datastore

Syntax

```
data = read(ds)
[data,info] = read(ds)
```

Description

`data = read(ds)` returns data from a datastore. Subsequent calls to `read` continue reading from the endpoint of the previous call.

`[data,info] = read(ds)` additionally returns information about the extracted data in `info`, including metadata.

Examples

Read Data in TabularTextDatastore

Create a datastore from the sample file, `airlinesmall.csv`, which contains tabular data.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA','MissingValue',0);
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'DepTime','ArrTime','ActualElapsedTime'};
```

While there is data available to be read from the datastore, read one block of data at a time and analyze the data. In this example, sum the actual elapsed time.

```
sumElapsedTime = 0;
while hasdata(ds)
    T = read(ds);
    sumElapsedTime = sumElapsedTime + sum(T.ActualElapsedTime);
end
```

View the sum of the actual elapsed time.

```
sumElapsedTime
```

```
sumElapsedTime =

    14531797
```

Read Data in KeyValueDatastore

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Read a subset of data in the datastore.

```
T = read(ds)
```

```
T =

    Key       Value
    ____    _______

    'AA'    [14930]
```

Change the number of key-value pairs to read at a time, by changing the `ReadSize` property of the datastore.

```
ds.ReadSize = 5;
```

Read the next five key-value pairs in the datastore.

```
T = read(ds)
```

```
T =

    Key       Value
    ____    _______

    'AS'    [ 2910]
```

```
'CO'    [ 8138]
'DL'    [16578]
'EA'    [  920]
'HP'    [ 3660]
```

Input Arguments

`ds` — Input datastore

datastore

Input datastore. Use the `datastore` function to create a datastore object from your data.

Output Arguments

`data` — Output data

table

Output data, returned as a table.

For `TabularTextDatastore` objects, the table variables are determined by the `SelectedVariableNames` property.

For `KeyValueDatastore` objects, the variable names are `Key` and `Value`.

`info` — Information about datastore

structure array

Information about the datastore, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
`File`	File name.
`FileType`	The type of file from which data is read, either `'mat'` for MAT-files or `'seq'` for sequence files. Applies to `KeyValueDatastore` only.

Field Name	Description
`Offset`	Starting position of the read operation, in bytes. For MAT-files, `Offset` is the index of the first key and value read.
`Size`	Total file size, in bytes. For MAT-files, the total number of key-value pairs in the file.
`NumCharactersRead`	Number of characters read. Applies to `TabularTextDatastore` only.

See Also

`datastore` | `hasdata` | `readall`

Introduced in R2014b

readall

Read all data in datastore

Syntax

```
data = readall(ds)
```

Description

`data = readall(ds)` returns all the data in the datastore specified by `ds`, and resets the datastore to the point where no data has been read from it.

If all of the data in the datastore does not fit in memory, then `readall` returns an error.

Examples

Read All Data in TabularTextDatastore

Create a datastore from the sample file `airlinesmall.csv`, which contains tabular data.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA');
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'DepTime','ArrTime','ActualElapsedTime'};
```

Read all the data in the datastore.

```
T = readall(ds);
```

`readall` returns all the data in a table.

View information about the table.

```
T.Properties
```

```
ans =
```

```
          Description: ''
    VariableDescriptions: {}
           VariableUnits: {}
         DimensionNames: {'Row'  'Variable'}
               UserData: []
               RowNames: {}
          VariableNames: {'DepTime'  'ArrTime'  'ActualElapsedTime'}
```

View a summary of the output table.

```
summary(T)
```

```
Variables:

    DepTime: 123523x1 double
        Values:

            min          1
            median    1335
            max       2505
            NaNs      2351

    ArrTime: 123523x1 double
        Values:

            min          1
            median    1522
            max       2608
            NaNs      2656

    ActualElapsedTime: 123523x1 double
        Values:

            min         11
            median     102
            max       1650
            NaNs      2657
```

Read All Data in KeyValueDatastore

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Read all the data in the datastore.

```
T = readall(ds);
```

View a summary of the output table.

```
summary(T)
```

```
Variables:

    Key: 29x1 cell string

    Value: 29x1 cell
```

Input Arguments

`ds` — Input datastore

datastore

Input datastore. Use the `datastore` function to create a datastore object from your data.

Output Arguments

`data` — All data in the datastore

table

All data in the datastore, returned as a table.

For `TabularTextDatastore` objects, the table variables are determined by the `SelectedVariableNames` property.

For `KeyValueDatastore` objects. the variable names are `Key` and `Value`.

See Also

`datastore` | `hasdata` | `readall`

Introduced in R2014b

reset

Reset datastore to initial state

Syntax

```
reset(ds)
```

Description

`reset(ds)` resets the datastore specified by `ds` to the state where no data has been read from it. This allows re-reading from the same datastore.

Examples

Reset Datastore to Initial State

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Read the first key-value pair.

```
T = read(ds);
```

Reset the datastore to the state where no data has been read from it.

```
reset(ds)
```

Input Arguments

`ds` — Input datastore

datastore

Input datastore. Use the `datastore` function to create a datastore object from your data.

See Also

`datastore`

Introduced in R2014b

Using TabularTextDatastore Objects

Datastore for collections of tabular text files

`TabularTextDatastore` objects are associated with collections of text files containing column-oriented or tabular data. Tabular data is data arranged in rows and columns of equal length. The encoding of the data must be ASCII or UTF-8. `TabularTextDatastore` properties specify how data is formatted in the files, and how you want to access the data.

Examples

Select Variables to Read

Create a datastore from the sample file `airlinesmall.csv`, which contains tabular data.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA');
```

View the variables in the datastore.

```
ds.VariableNames
```

```
ans =

  Columns 1 through 5

    'Year'    'Month'    'DayofMonth'    'DayOfWeek'    'DepTime'

  Columns 6 through 10

    'CRSDepTime'    'ArrTime'    'CRSArrTime'    'UniqueCarrier'    'FlightNum'

  Columns 11 through 14

    'TailNum'    'ActualElapsedTime'    'CRSElapsedTime'    'AirTime'

  Columns 15 through 20

    'ArrDelay'    'DepDelay'    'Origin'    'Dest'    'Distance'    'TaxiIn'
```

```
  Columns 21 through 24

    'TaxiOut'    'Cancelled'    'CancellationCode'    'Diverted'

  Columns 25 through 28

    'CarrierDelay'    'WeatherDelay'    'NASDelay'    'SecurityDelay'

  Column 29

    'LateAircraftDelay'
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'Year','Month','Cancelled'};
```

Alternatively, you can specify the variables of interest when you create the datastore.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA','SelectedVariableNames',{'Year
```

Specify Format of Data to Read

Create a datastore from the sample file `airlinesmall.csv`, which contains tabular data.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA');
```

Specify the variables of interest.

```
ds.SelectedVariableNames = {'Year','Month','UniqueCarrier'};
```

View the `SelectedFormats` property.

```
ds.SelectedFormats
```

```
ans =

    '%f'    '%f'    '%q'
```

The `SelectedFormats` property indicates that the `Year` and `Month` variables will be interpreted as columns of floating-point values, and the `UniqueCarrier` variable will be interpreted as a column of text strings.

Specify that the first two variables should be read as signed integers, and the third variable should be read as a categorical value by modifying the `SelectedFormats` property.

```
ds.SelectedFormats = {'%d','%d','%C'};
```

Preview the data.

```
T = preview(ds)
```

```
T =

    Year    Month    UniqueCarrier
    ____    _____    _____________

    1987    10       PS
    1987    10       PS
    1987    10       PS
    1987    10       PS
    1987    10       PS
    1987    10       PS
    1987    10       PS
    1987    10       PS
```

- "Read and Analyze Large Tabular Text File"

Properties

TabularTextDatastore Properties

Object Functions

`hasdatanumpartitions` `partition` `preview` `read` `readall` `reset`

Create Object

Create `TabularTextDatastore` objects using the `datastore` function.

See Also

`mapreduce` | `textscan`

More About

- “Getting Started with Datastore”

datatipinfo

Produce short description of input variable

Syntax

```
datatipinfo(var)
```

Description

`datatipinfo(var)` displays a short description of a variable, similar to what is displayed in a datatip in the MATLAB debugger.

Examples

Get datatip information for a 5-by-5 matrix:

```
A = rand(5);

datatipinfo(A)
A: 5x5 double =
    0.4445    0.3567    0.7458    0.0767    0.4400
    0.7962    0.6575    0.3918    0.8289    0.9746
    0.5641    0.9808    0.0265    0.4838    0.6722
    0.9099    0.9653    0.2508    0.4859    0.4054
    0.2857    0.5198    0.7383    0.9301    0.9604
```

Get datatip information for a 50-by-50 matrix. For this larger matrix, `datatipinfo` displays just the size and data type:

```
A = rand(50);

datatipinfo(A)
A: 50x50 double
```

Also for multidimensional matrices, `datatipinfo` displays just the size and data type:

```
A = rand(5);
```

```
A(:,:,2) = A(:,:,1);

datatipinfo(A)
A: 5x5x2 double
```

See Also

inputname | nargin | narginchk | varargin | inputParser

Introduced before R2006a

date

Current date string

Syntax

```
str = date
```

Description

`str = date` returns a string containing the date in the format, day-month-year, for example, `01-Jan-2014`.

More About

Tips

- To return a datetime scalar representing the current date, type:

  ```
  d = datetime('today')
  ```

See Also

`clock` | `datestr` | `datetime` | `datenum` | `now`

Introduced before R2006a

datenum

Convert date and time to serial date number

The `datenum` function creates a numeric array that represents each point in time as the number of days from January 0, 0000. The numeric values also can represent elapsed time in units of days. However, the best way to represent points in time is by using the `datetime` data type. The best way to represent elapsed time is by using the `duration` or `calendarDuration` data types.

Syntax

```
DateNumber = datenum(t)

DateNumber = datenum(DateString)
DateNumber = datenum(DateString,formatIn)
DateNumber = datenum(DateString,PivotYear)
DateNumber = datenum(DateString,formatIn,PivotYear)

DateNumber = datenum(DateVector)
DateNumber = datenum(Y,M,D)
DateNumber = datenum(Y,M,D,H,MN,S)
```

Description

`DateNumber = datenum(t)` converts the datetime values in `datetime` array `t` to serial date numbers.

A serial date number represents the whole and fractional number of days from a fixed, preset date (January 0, 0000).

`DateNumber = datenum(DateString)` converts date strings to serial date numbers. If the date string format is known, use `formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

`DateNumber = datenum(DateString,formatIn)` uses `formatIn` to interpret each date string.

`DateNumber = datenum(DateString,PivotYear)` uses `PivotYear` to interpret date strings that specify the year as two characters. If the date string format is known, use

`formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

`DateNumber = datenum(DateString,formatIn,PivotYear)` uses `formatIn` to interpret each date string, and `PivotYear` to interpret date strings that specify the year as two characters. You can specify `formatIn` and `PivotYear` in either order.

`DateNumber = datenum(DateVector)` converts date vectors to serial date numbers, and returns a column vector of `m` date numbers, where `m` is the total number of date vectors in `DateVector`.

`DateNumber = datenum(Y,M,D)` returns the serial date numbers for corresponding elements of the `Y`, `M`, and `D` (year, month, day) arrays. The arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y,M,D]`.

`DateNumber = datenum(Y,M,D,H,MN,S)` additionally returns the serial date numbers for corresponding elements of the `H`, `MN`, and `S` (hour, minute, and second) arrays. The arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y,M,D,H,MN,S]`.

Examples

Convert datetime Array to Date Numbers

```
format long

t = [datetime('now');datetime('tomorrow')]
DateNumber = datenum(t)
```

```
t =

   23-Feb-2015 09:59:02
   24-Feb-2015 00:00:00


DateNumber =

   1.0e+05 *

   7.360184160032523
```

```
   7.360190000000000
```

Convert Date String to Date Number

```
DateString = '19-May-2001';
formatIn = 'dd-mmm-yyyy';
datenum(DateString,formatIn)
```

```
ans =

      730990
```

`datenum` returns a date number for the date string with the format `'dd-mmm-yyyy'` .

Convert Multiple Date Strings to Date Numbers

Pass multiple date strings in a cell array. All input date strings must use the same format.

```
DateString = {'09/16/2007';'05/14/1996';'11/29/2010'};
formatIn = 'mm/dd/yyyy';
datenum(DateString,formatIn)
```

```
ans =

      733301
      729159
      734471
```

Convert Date String to Date Number Using Pivot Year

Convert a date string to a serial date number using the default pivot year.

```
n = datenum('12-jun-17','dd-mmm-yy')
```

```
n =

      736858
```

The corresponding date string to this date number is 12-Jun-2017.

Convert the same date string to a serial date number using 1400 as the pivot year.

```
n = datenum('12-jun-17','dd-mmm-yy',1400)
```

```
n =

      517712
```

The corresponding date string to this date number is 12-Jun-1417.

Convert Date Vector to Date Number

```
datenum([2009,4,2,11,7,18])
```

```
ans =

   7.3387e+05
```

Convert Year, Month, and Day to Date Number

Convert a date specified by year, month and day values to a serial date number.

```
n = datenum(2001,12,19)
```

```
n =

      731204
```

Input Arguments

`t` — `datetime` values

scalar | vector | matrix | multidimensional array

`datetime` values, specified as a scalar, vector, matrix, or multidimensional `datetime` array. `datenum` does not account for time zone information in `t` and does not adjust datetime values that occur during Daylight Saving Time. That is, `datenum` treats the `TimeZone` property of `t` as empty and converts the remaining date and time information to a serial date number.

`DateVector` — Date vectors

matrix

Date vectors, specified as an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. A full date vector has six elements, specifying year, month, day, hour, minute, and second, in that order. A partial date vector has three elements, specifying year, month, and day, in that order. Each element of `DateVector` should be a positive or negative integer value with the exception of the seconds element, which can be fractional. If an element falls outside the conventional range, `datenum` adjusts both that date vector element and the previous element. For example, if the minutes element is `70`, `datenum` adjusts the hours element by `1` and sets the minutes element to `10`. If the minutes element is `-15`, then `datevec` decreases the hours element by `1` and sets the minutes element to `45`. Month values are an exception. `datenum` sets month values less than `1` to `1`.

Example: `[2003,10,24,12,45,07]`

Data Types: `double`

`DateString` — Date strings

string | character array | 1–D cell array of strings

Date strings, specified as a character array where each row corresponds to one date string, or as a one dimensional cell array of strings. All of the date strings must have the same format.

Example: `'24–Oct-2003 12:45:07'`

Example: `['19-Sep-2013';'20-Sep-2013';'21-Sep-2013']`

Example: `{'15-Oct-2010' '20-Nov-2012'}`

If the date string format is known, you should also specify `formatIn`. If you do not specify `formatIn`, `DateString` must be in one of the following formats.

Date String Format	Example
`'dd-mmm-yyyy HH:MM:SS'`	`01-Mar-2000 15:45:17`
`'dd-mmm-yyyy'`	`01-Mar-2000`
`'mm/dd/yyyy'`	`03/01/2000`
`'mm/dd/yy'`	`03/01/00`

Date String Format	Example
'mm/dd'	03/01
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
'mmm.dd,yyyy'	Mar.01,2000
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17
'yyyy-mm-dd'	2000-03-01
'yyyy/mm/dd'	2000/03/01
'HH:MM:SS'	15:45:17
'HH:MM:SS PM'	3:45:17 PM
'HH:MM'	15:45
'HH:MM PM'	3:45 PM

Note: The symbolic identifiers describing date string formats are different from those that describe the display formats of `datetime` arrays.

Certain date string formats might not contain enough information to convert the date string. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. `datevec` and `datenum` consider two-character date string years (e.g., `'79'`) to fall within the 100-year range centered around the current year.

When you do not specify `formatIn`, note the following:

- For the formats that specify the month as two digits (`mm`), the month value must not be greater than 12.
- However, for the format `'mm/dd/yy'`, if the first entry in the date string is greater than 12 and the second entry is less than or equal to 12, then `datenum` considers the date string to be in `'yy/mm/dd'` format.

`formatIn` — Format of the input date string

string

Format of the input date string, specified as a string of symbolic identifiers.

Example: `'dddd, mmm dd, yyyy'`

The following table shows symbolic identifiers you can use to construct the `formatIn` string. You can include characters such as a hyphen, space, or colon to separate the fields.

Note: The symbolic identifiers describing date string formats are different from those that describe the display formats of `datetime` arrays.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	`03`, `12`
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`
`dd`	Day in two digits	`05`, `20`
`d`	Day using capitalized first letter	`M`, `T`
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	`05`, `5 AM`
`MM`	Minute in two digits	`12`, `02`
`SS`	Second in two digits	`07`, `59`
`FFF`	Millisecond in three digits	`057`
`AM or PM`	`AM` or `PM` inserted in date string	`3:45:02 PM`

The `formatIn` string must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- You only can use `QQ` alone or with a year specifier.

`PivotYear` — Start year of 100-year date range

present minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret date strings that specify the year as two characters.

If `formatIn` contains the time of day, the pivot year is computed from the current time of the current day, month, and year. Otherwise it is computed from midnight of the current day, month, and year.

Example: `2000`

Data Types: `double`

`Y,M,D` — Year, month, and day arrays

scalar | vector | matrix | array

Year, month, and day arrays specified as a scalar, vector, matrix or array. These must be the same size, or any one can be a scalar. `Y,M,D` should be integer values.

If `Y,M,D` are all scalars or all column vectors, you can specify the input arguments as a date vector, `[Y,M,D]`.

Example: `2003,10,24`

Data Types: `double`

`Y,M,D,H,MN,S` — Year, month, day, hour, minute and second arrays

scalar | vector | matrix | array

Year, month, day, hour, minute and second arrays specified as a scalar, vector, matrix or array. These must be the same size, or any one can be a scalar. `datenum` does not accept milliseconds as a separate input, but as a fractional part of the seconds input, `S`. `Y,M,D,H,MN` should be integer values.

If `Y,M,D,H,MN,S` are all scalars or all column vectors, you can specify the input arguments as a date vector `[Y,M,D,H,MN,S]`.

Example: `2003,10,24,12,45,07.451`

Data Types: `double`

Output Arguments

`DateNumber` — Serial date numbers

scalar | vector

Serial date numbers, returned as a column vector of length `m`, where `m` is the total number of input date vectors or date strings.

More About

Tips

- To create arbitrarily shaped output, use the `datenum(Y,M,D)` and `datenum(Y,M,D,H,MN,S)` syntaxes. The `datenum(DateVector)` syntax creates only a column vector of date numbers.

  ```
  datenum(2013,[1 3; 2 4],ones(2,2))

  ans =

        735235       735294
        735266       735325
  ```

- "Represent Dates and Times in MATLAB"
- "Carryover in Date Vectors and Strings"

See Also

`datestr` | `datetime` | `datevec`

Introduced before R2006a

dateshift

Shift date or generate sequence of dates and time

Syntax

```
t2 = dateshift(t,'start',unit)
t2 = dateshift(t,'end',unit)
t2 = dateshift(t,'dayofweek',dow)

t2 = dateshift(___,rule)
```

Description

`t2 = dateshift(t,'start',unit)` shifts each value in the `datetime` array `t` back to the beginning of the unit of time specified by `unit`. The output `t2` is the same size as `t`.

`t2 = dateshift(t,'end',unit)` shifts the values ahead to the end of the unit of time specified by `unit`. The end of a day, hour, minute, or second is also the beginning of the next one. For example, the end of a day occurs at midnight at the beginning of the next day. The end of a year, quarter, month, or week is midnight at the beginning of the last day of that time period.

`t2 = dateshift(t,'dayofweek',dow)` returns the next occurrence of the specified day of the week on or after each datetime in array `t`. If the date in `t` falls on the specified day of the week, then `dateshift` returns the same date.

`t2 = dateshift(___,rule)` shifts each value in array `t` according to the pattern specified by `rule`. You can use this syntax with any of the arguments in the previous syntaxes.

Examples

Shift Current Date to End of Current Month

Define the current date.

```
t = datetime('today')
```

```
t =

   23-Feb-2015
```

Shift the date to the end of the same month.

```
t2 = dateshift(t,'end','month')
```

```
t2 =

   28-Feb-2015
```

Shift Current Date to Next Month

Define the current date.

```
t = datetime('today')
```

```
t =

   23-Feb-2015
```

Shift the date to the start of the next month.

```
t2 = dateshift(t,'start','month','next')
```

```
t2 =

   01-Mar-2015
```

Shift the date to the end of the next month.

```
t2 = dateshift(t,'end','month','next')
```

```
t2 =
```

```
   31-Mar-2015
```

Shift Dates to Specific Day of Week

Shift an array of dates forward to the next Friday.

```
t = datetime([2014,08,03;2014,04,15])
```

```
t =

   03-Aug-2014
   15-Apr-2014
```

```
t2 = dateshift(t,'dayofweek','Friday')
```

```
t2 =

   08-Aug-2014
   18-Apr-2014
```

Shift the array of dates backward to the previous Monday.

```
t2 = dateshift(t,'dayofweek','Monday','previous')
```

```
t2 =

   28-Jul-2014
   14-Apr-2014
```

Determine Future Date

Find the date that falls at the end of the fifth week from today.

```
t = datetime('today')
```

```
t =
```

```
   23-Feb-2015

t2 = dateshift(t,'end','week',5)

t2 =

   04-Apr-2015
```

Create Sequence of Dates Falling on Specific Day of Week

Generate a sequence of dates consisting of the next three occurrences of Friday.

```
t = datetime('today')

t =

   23-Feb-2015

t2 = dateshift(t,'dayofweek','Friday',1:3)

t2 =

   27-Feb-2015   06-Mar-2015   13-Mar-2015
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`unit` — Unit of time

`'year' | 'quarter' | 'month' | 'week' | 'day' | 'hour' | 'minute' | 'second'`

Unit of time, specified as one of the following strings:

- `'year'`

- 'quarter'
- 'month'
- 'week'
- 'day'
- 'hour'
- 'minute'
- 'second'

dow — Day of Week

scalar integer | string

Day of the week, specified as a scalar integer indicating the day of week number, or a string containing a localized day name.

Example: 'Sunday'

Example: 1

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char

rule — Rule for shifting datetime values

string | scalar integer | array of integer values

Rule for shifting datetime values, specified as a string, scalar integer, or an array of integer values. If rule is a string, it must be one of the following.

Value of rule	Description
'next'	Shift datetime to next unit of time or specified day.
'previous'	Shift datetime to previous unit of time or specified day.
'nearest'	Shift datetime to nearest occurrence of unit of time or specified day.
'current'	Shift datetime within the current unit of time, or to the specified day in the current week.

If rule is an integer or an array of integers, then:

- When used with `unit`, `0` corresponds to the start or end of the current unit for each datetime, `1` corresponds to the next unit, and `-1` corresponds to the previous unit, and so on.
- When used with `dow`, `0` corresponds to the specified day in the current week for each datetime, `1` corresponds to the next occurrence of the specified day, and `-1` corresponds to the previous occurrence, and so on.
- `t` and `rule` must be the same size, or one must be a scalar.

See Also

`between` | `colon (:)`

Introduced in R2014b

datestr

Convert date and time to string format

The `datestr` function creates a string or character array that display one or more points in time. However, the best way to represent points in time is by using the `datetime` data type.

Syntax

```
DateString = datestr(t)
DateString = datestr(DateVector)
DateString = datestr(DateNumber)

DateString = datestr( ___ ,formatOut)

DateString = datestr(DateStringIn)
DateString = datestr(DateStringIn,formatOut,PivotYear)

DateString = datestr( ___ ,'local')
```

Description

`DateString = datestr(t)` converts the datetime values in `datetime` array `t` to date strings. `datestr` returns a column vector of `m` date strings, where `m` is the total number of datetime values in `t`.

By default, `datestr` returns date strings in the format, day-month-year hour:minute:second. If hour:minute:second `= 00:00:00`, then the date string returned has the format, day-month-year.

`DateString = datestr(DateVector)` converts date vectors to date strings. `datestr` returns a column vector of `m` date strings, where `m` is the total number of date vectors in `DateVector`.

`DateString = datestr(DateNumber)` converts serial date numbers to date strings. `datestr` returns a column vector of `m` date strings, where `m` is the total number of date numbers in `DateNumber`.

`DateString = datestr( ___ ,formatOut)` specifies the format of the output date strings using `formatOut`. You can use formatOut with any of the input arguments in the above syntaxes.

`DateString = datestr(DateStringIn)` converts `DateStringIn` to date strings in the format, day-month-year hour:minute:second. All date strings in `DateStringIn` must have the same format.

`DateString = datestr(DateStringIn,formatOut,PivotYear)` converts `DateStringIn` to `DateString`, in the format specified by `formatOut`, and using optional `PivotYear` to interpret date strings that specify the year as two characters.

`DateString = datestr( ___ ,'local')` returns the date string in the language of the current locale. This is the language you select by means of your computer's operating system. If you leave `local` out of the argument list, `datestr` returns the date string in the default language, which is US English. Use `local` with any of the previous syntaxes. The `local` argument must be last in the argument sequence.

Examples

Convert datetime Array to Date Strings

```
t = [datetime('now');datetime('tomorrow')]
DateString = datestr(t)
```

```
t =

   23-Feb-2015 09:59:08
   24-Feb-2015 00:00:00


DateString =

23-Feb-2015 09:59:08
24-Feb-2015 00:00:00
```

`datestr` returns date strings in the format, day-month-year hour:minute:second.

Convert Date Vector to Date String

```
DateVector = [2009,4,2,11,7,18];
```

```
datestr(DateVector)
```

```
ans =

02-Apr-2009 11:07:18
```

`datestr` returns a date string in the default date string format.

Convert Date and Time to Specific Format

Format the current date in the `mm/dd/yy` format.

You can specify this format using a string of symbolic identifiers.

```
formatOut = 'mm/dd/yy';
datestr(now,formatOut)
```

```
ans =

02/23/15
```

Alternatively, you can specify this format using a numeric identifier.

```
formatOut = 2;
datestr(now,formatOut)
```

```
ans =

02/23/15
```

You can reformat the date and time, and also show milliseconds.

```
dt = datestr(now,'mmmm dd, yyyy HH:MM:SS.FFF AM')
```

```
dt =

February 23, 2015  9:59:03.358 AM
```

Convert 12-Hour Time String to 24-Hour Equivalent

Convert the 12-hour time 05:32 p.m. to its 24-hour equivalent.

```
datestr('05:32 PM','HH:MM')
```

```
ans =

17:32
```

Convert the 24-hour time 05:32 to its 12-hour equivalent.

```
datestr('05:32','HH:MM PM')
```

```
ans =

 5:32 AM
```

The use of `AM` or `PM` in the `formatOut` string does not influence which characters actually become part of the date string; they only determine whether or not to include them in the date string. MATLAB® selects `AM` or `PM` based on the time entered.

Convert Date String from Custom Format

Call `datenum` inside of `datestr` to specify the format of the input date string.

```
formatOut = 'dd mmm yyyy';
datestr(datenum('16-04-55','dd-mm-yy',1900),formatOut)
```

```
ans =

16 Apr 1955
```

Convert Multiple Date Strings

Convert more than one date string input by passing the multiple date strings in a cell array.

All input date strings must use the same format. For example, the following command passes three dates that all use the `mm/dd/yyyy` format.

```
datestr(datenum({'09/16/2007';'05/14/1996';'11/29/2010'}, ...
    'mm/dd/yyyy'))
```

```
ans =

16-Sep-2007
14-May-1996
29-Nov-2010
```

`datestr` returns a character array of converted date strings in the format, day-month-year.

Convert Date String with Values Outside Normal Range

Call `datenum` inside of `datestr` to return the expected value, because the date below uses a value outside its normal range (month=13).

```
datestr(datenum('13/24/88','mm/dd/yy'))
```

```
ans =

24-Jan-1989
```

Use a Pivot Year

Change the pivot year to change the year range.

Use a pivot year of 1900.

```
DateStringIn = '4/16/55';
formatOut = 1;
PivotYear = 1900;
datestr(DateStringIn,formatOut,PivotYear)
```

```
ans =

16-Apr-1955
```

For the same date string, use a pivot year of 2000.

```
PivotYear = 2000;
datestr(DateStringIn,formatOut,PivotYear)
```

```
ans =

16-Apr-2055
```

Return Date String in Local Language

Convert a date number to a date string in the language of the current locale.

Use the `local` argument in a French locale.

```
DateNumber = 725935;
formatOut = 'mmmm-dd-yyyy';
str = datestr(DateNumber,formatOut,'local')
```

```
str =
Juillet-17-1987
```

You can make the same call without specifying `'local'`.

```
str = datestr(DateNumber,formatOut)
```

```
str =
July-17-1987
```

In this case, the output defaults to the English language.

Input Arguments

`t` — `datetime` values

scalar | vector | matrix | multidimensional array

`datetime` values, specified as a scalar, vector, matrix, or multidimensional `datetime` array.

`DateVector` — Date vectors

matrix

Date vectors, specified as an `m`-by-6 matrix, where `m` is the number of full (six-element) date vectors. Each element of `DateVector` should be a positive or negative integer value with the exception of the seconds element, which can be fractional. If an element falls outside the conventional range, `datestr` adjusts both that date vector element and the previous element. For example, if the minutes element is `70`, `datestr` adjusts the hours element by `1` and sets the minutes element to `10`. If the minutes element is `-15`, then `datestr` decreases the hours element by `1` and sets the minutes element to `45`. Month values are an exception. `datestr` sets month values less than `1` to `1`.

Example: `[2003,10,24,12,45,07]`

Data Types: `double`

`DateNumber` — Serial date numbers

scalar | vector | matrix | array

Serial date numbers, specified as a scalar, vector, matrix, or array of positive double-precision numbers.

Example: `731878`

Data Types: `double`

`formatOut` — Format of the date string output

`-1` (default) | string | integer

Format of the date string output, specified as a string of symbolic identifiers or an integer that corresponds to a predefined format. If you do not specify `formatOut`, then `datestr` returns a date string in the default date string format `dd-mmm-yyyy HH:MM:SS` (day-month-year hour:minute:second). By default, if `HH:MM:SS = 00:00:00` then the date string returned has the format `dd-mmm-yyyy`.

The following table shows symbolic identifiers you can use to construct the `formatOut` string. You can include characters such as a hyphen, space, or colon to separate the fields.

Note: The symbolic identifiers describing date string formats are different from those that describe the display formats of `datetime` arrays.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	1990, 2002

Symbolic Identifier	Description	Example
`yy`	Year in two digits	`90`, `02`
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	`03`, `12`
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`
`dd`	Day in two digits	`05`, `20`
`d`	Day using capitalized first letter	`M`, `T`
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	`05`, `5 AM`
`MM`	Minute in two digits	`12`, `02`
`SS`	Second in two digits	`07`, `59`
`FFF`	Millisecond in three digits	`057`
`AM or PM`	`AM` or `PM` inserted in date string	`3:45:02 PM`

The `formatOut` string must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- You only can use `QQ` alone or with a year specifier.

This table lists predefined date formats that you can use with `datestr`.

Numeric Identifier	Date String Format	Example
-1 (default)	'dd-mmm-yyyy HH:MM:SS' or 'dd-mmm-yyyy' if 'HH:MM:SS'= 00:00:00	01-Mar-2000 15:45:17 or 01-Mar-2000
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1-01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd,yyyy'	Mar.01,2000
23	'mm/dd/yyyy'	03/01/2000
24	'dd/mm/yyyy'	01/03/2000

Numeric Identifier	Date String Format	Example
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01
27	'QQ-YYYY'	Q1-2001
28	'mmmyyyy'	Mar2000
29	'yyyy-mm-dd' (ISO 8601)	2000-03-01
30	'yyyymmddTHHMMSS' (ISO 8601)	20000301T154517
31	'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

`DateStringIn` — Date strings to convert
string | cell array

Date strings to convert, specified as a single string or a cell array of strings, where each row corresponds to one date string.

`datestr` considers two-character date string years (for example, `'79'`) to fall within the 100-year range centered around the current year.

All date strings must have the same date format, and they must be in one of the following date formats.

Date String Format	Example
'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
'dd-mmm-yyyy'	01-Mar-2000
'mm/dd/yyyy'	03/01/2000
'mm/dd/yy'	03/01/00
'mm/dd'	03/01
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
'mmm.dd,yyyy'	Mar.01,2000
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17
'yyyy-mm-dd'	2000-03-01

Date String Format	Example
'yyyy/mm/dd'	2000/03/01
'HH:MM:SS'	15:45:17
'HH:MM:SS PM'	3:45:17 PM
'HH:MM'	15:45
'HH:MM PM'	3:45 PM

Note: When converting from one date string format to another, you should first pass the strings to the `datenum` function, so that you can specify the format of the input date strings. This ensures that the format of the input date strings is correctly interpreted. For example, see "Convert Date String from Custom Format" on page 1-1915.

`PivotYear` — Start year of 100-year date range

present minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret date strings that specify the year as two characters.

If `formatIn` contains the time of day, the pivot year is computed from the current time of the current day, month, and year. Otherwise it is computed from midnight of the current day, month, and year.

Example: `2000`

Data Types: `double`

Output Arguments

`DateString` — Date strings

string | two-dimensional character array

Date strings, returned as a character array with `m` rows, where `m` is the total number of input date vectors, serial date numbers, or date strings. The default output date string format is `dd-mmm-yyyy HH:MM:SS` (day-month-year hour:minute:second) unless the hours, minutes and seconds are all 0 in which case `HH:MM:SS` is suppressed.

More About

Tips

- To convert a date string not in a predefined MATLAB date format, first convert the date string to a date number, using either `datenum` or `datevec`.

- "Represent Dates and Times in MATLAB"
- "Converting Date Vector Returns Unexpected Output"

See Also

`cellstr` | `datenum` | `datetime` | `datevec`

Introduced before R2006a

datetick

Date formatted tick labels

The `datetick` function labels the tick lines of an axis using dates, replacing the default labels. `datetick` is useful when plotting numeric values that are serial date numbers.

For 2-D line and scatter plots, it is more convenient to plot datetime values using the `plot` function. Specify the format of the tick labels using the `DatetimeTickFormat` name-value pair argument. You do not need to use `datetick` to label such plots of datetime values.

Syntax

```
datetick(tickaxis)
datetick(tickaxis,dateFormat)

datetick(___,'keeplimits')

datetick(___,'keepticks')

datetick(axes_handle,___)
```

Description

`datetick(tickaxis)` labels the tick lines of the axis specified by `tickaxis` using dates, replacing the default numeric labels. `datetick` selects a label format based on the minimum and maximum limits of the specified axis. The axis data values should be serial date numbers, as returned by the `datenum` function.

`datetick(tickaxis,dateFormat)` formats the labels according to the string `dateFormat`.

`datetick(___,'keeplimits')` changes the tick labels to date-based labels while preserving the axis limits. Append `'keeplimits'` to any of the previous syntaxes.

`datetick(___,'keepticks')` changes the tick labels to date-based labels while preserving their locations. Append `'keepticks'` to any of the previous syntaxes.

`datetick(axes_handle, ___ )` labels the tick lines of an axis on the axes specified by `axes_handle`. The `axes_handle` argument can precede any of the input argument combinations in the previous syntaxes.

Examples

Label x-Axis Ticks with 2-digit Years

Graph population data for the 20th Century taken from the 1990 US census and label *x*-axis ticks with 2-digit years.

Create time data by decade.

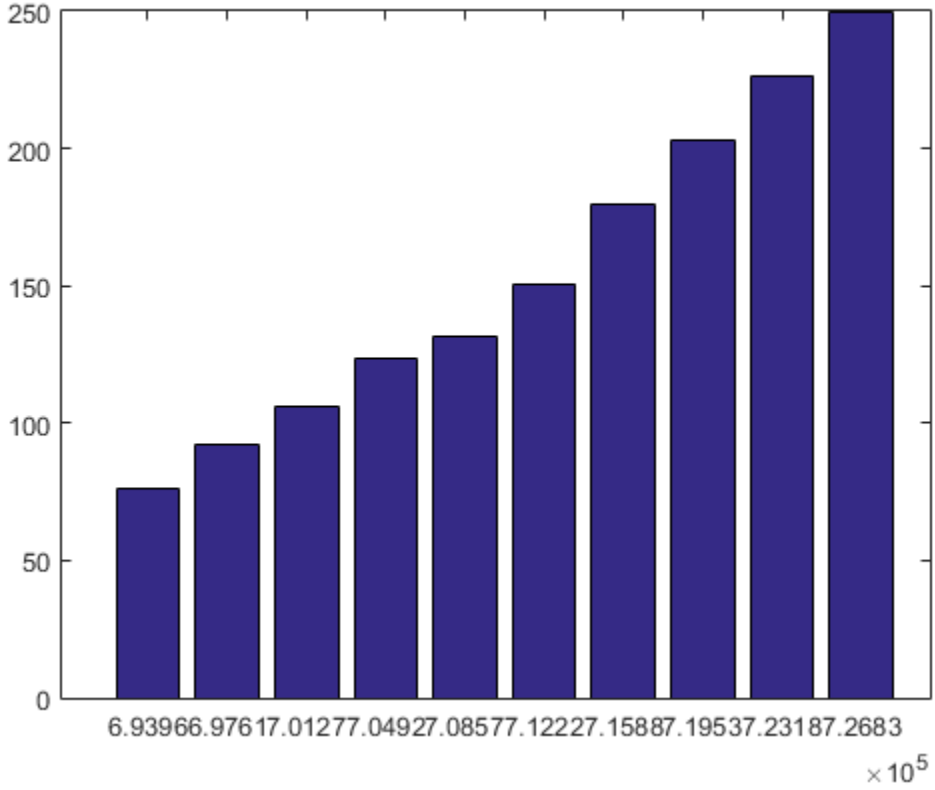
```
t = (1900:10:1990)';
```

Enter total population counts for the USA.

```
p = [75.995 91.972 105.711 123.203 131.669 ...
 150.697 179.323 203.212 226.505 249.633]';
```

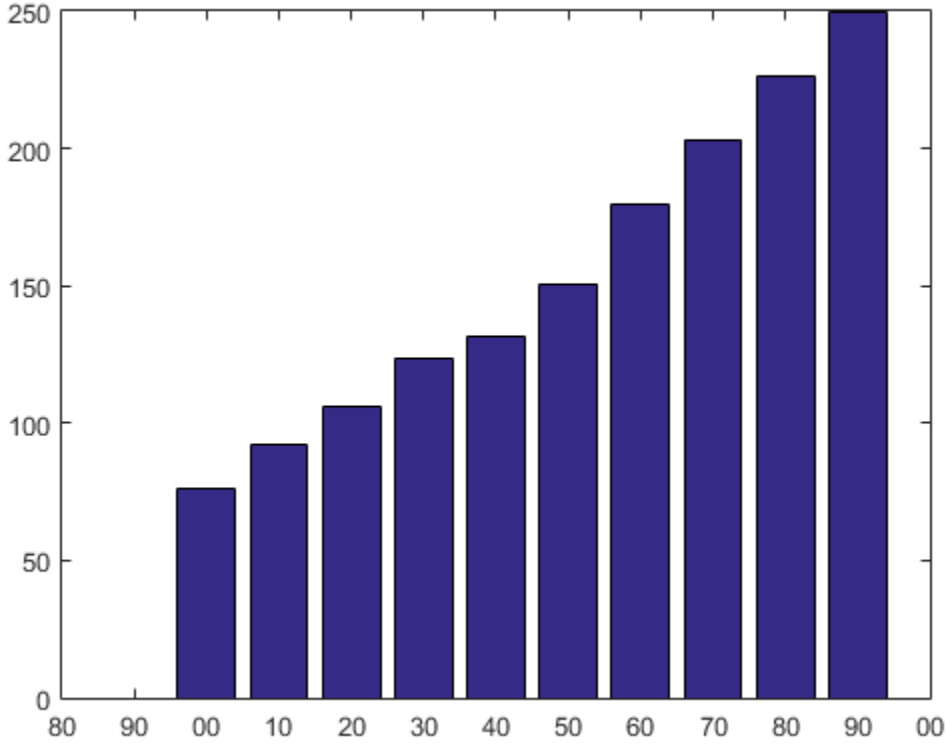
Convert years to serial date numbers using the `datenum` function, and then create a bar graph of the data.

```
figure
bar(datenum(t,1,1),p)
```



Replace *x*-axis ticks with 2-digit years. The numeric identifier `11` corresponds to the predefined MATLAB® date format `'yy'`.

```
dateFormat = 11;
datetick('x',dateFormat)
```



Label x-Axis Ticks with Hours of the Day

Plot traffic count data against date ticks for hours of the day showing AM and PM.

Get traffic count data.

```
load count.dat
```

Create arrays for an arbitrary date, for example, April 18, 1995.

```
n = length(count);
year = repmat(1995,1,n);
month = repmat(4,1,n);
day = repmat(18,1,n);
```

Create arrays for each of 24 hours.

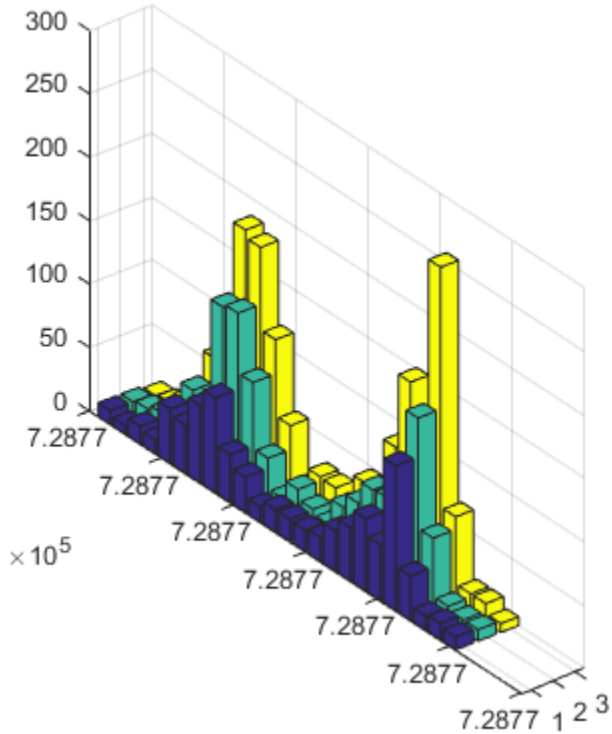
```
hour = 1:n;
minutes = zeros(1,n);
```

Get the serial date numbers for the date arrays.

```
xdate = datenum(year,month,day,hour,minutes,minutes);
```

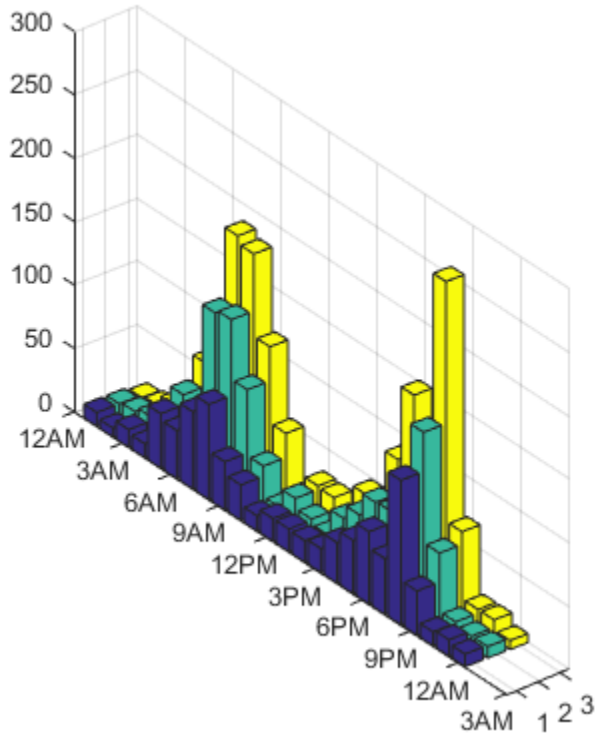
Plot a 3-D bar graph of the traffic data against the serial date numbers.

```
figure
bar3(xdate,count)
```



Label the tick lines of the graph's *x*-axis with the hours of the day.

```
datetick('y','HHPM')
```



Label x-Axis and Preserve Axis Limits

Select a starting date.

```
startDate = datenum('02-01-1962');
```

Select an ending date.

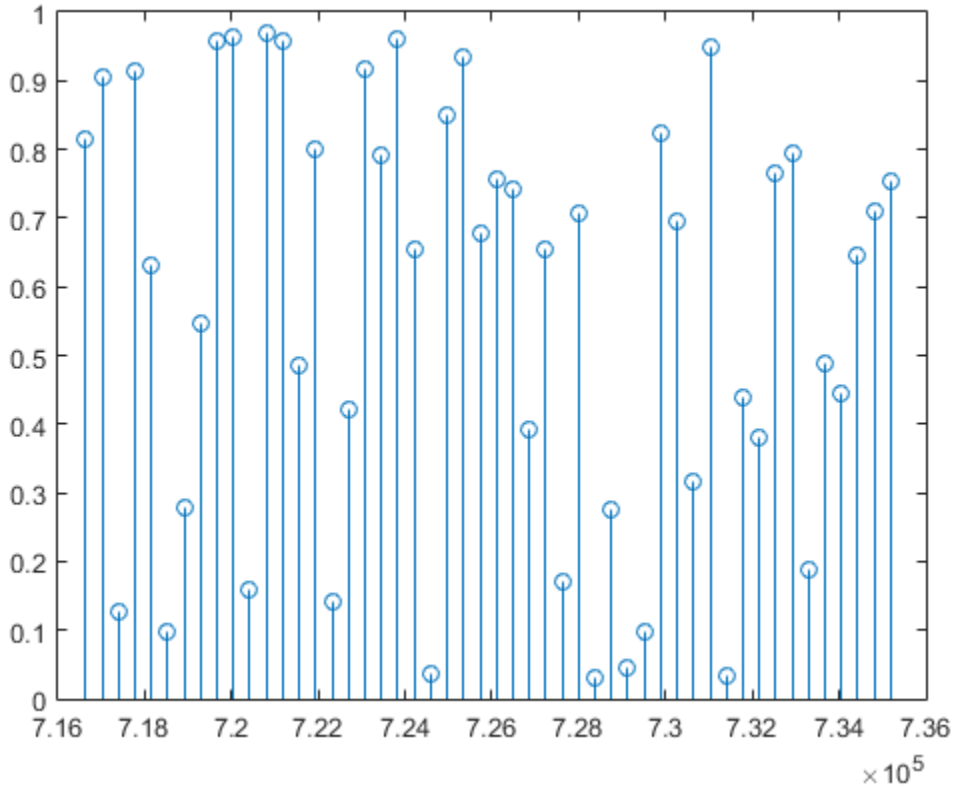
```
endDate = datenum('11-15-2012');
```

Create a variable, `xdata`, that corresponds to the number of years between the start and end dates.

```
xData = linspace(startDate,endDate,50);
```

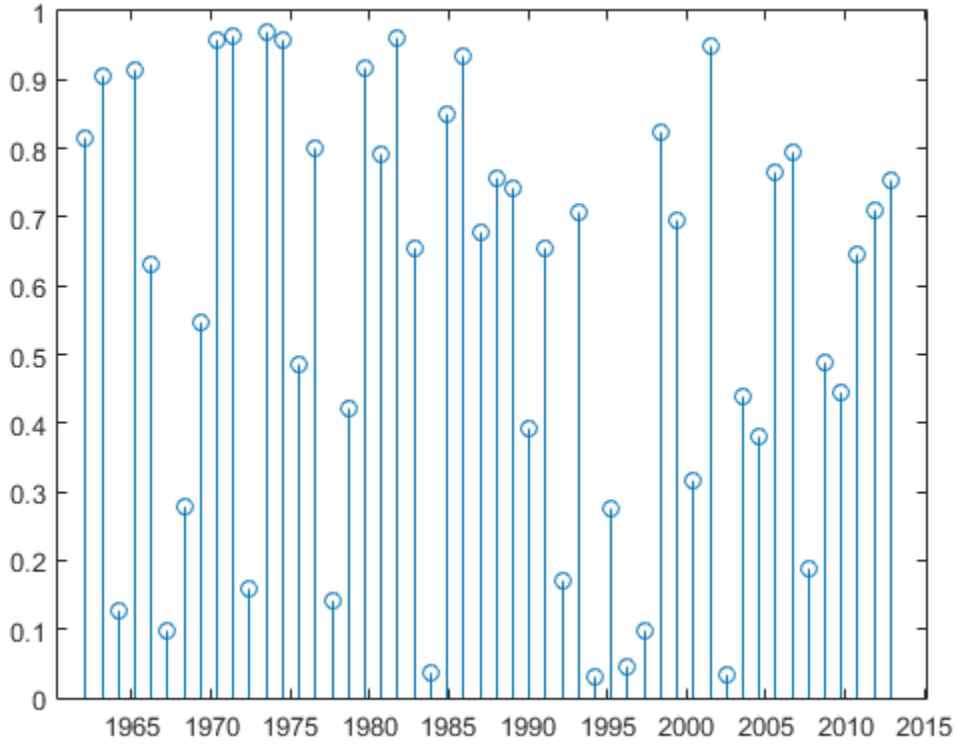
Plot random data.

```
figure
stem(xData,rand(1,50))
```



Label the *x*-axis with 4-digit years, preserving the *x*-axis limits by using the `'keeplimits'` option.

```
datetick('x','yyyy','keeplimits')
```



Add Month Labels to Plot and Preserve Number of Ticks

Select a starting date.

```
startDate = datenum('01-01-2009');
```

Select an ending date.

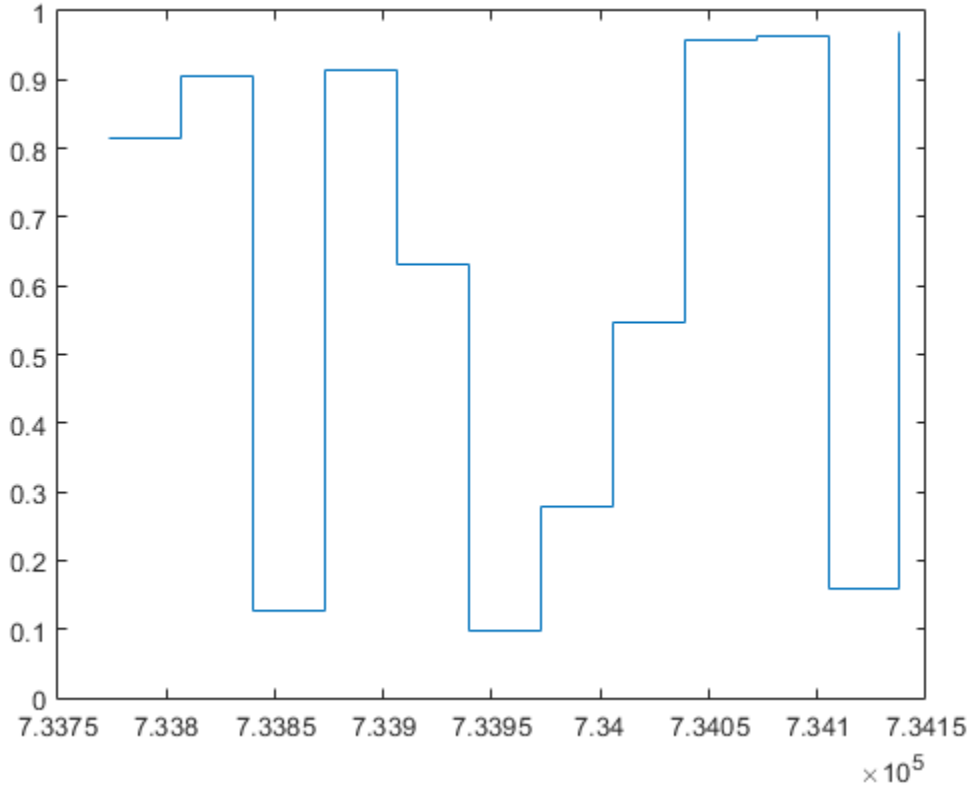
```
endDate = datenum('12-31-2009');
```

Create a variable, `xdata`, that corresponds to the number of months between the start and end dates.

```
xData = linspace(startDate,endDate,12);
```

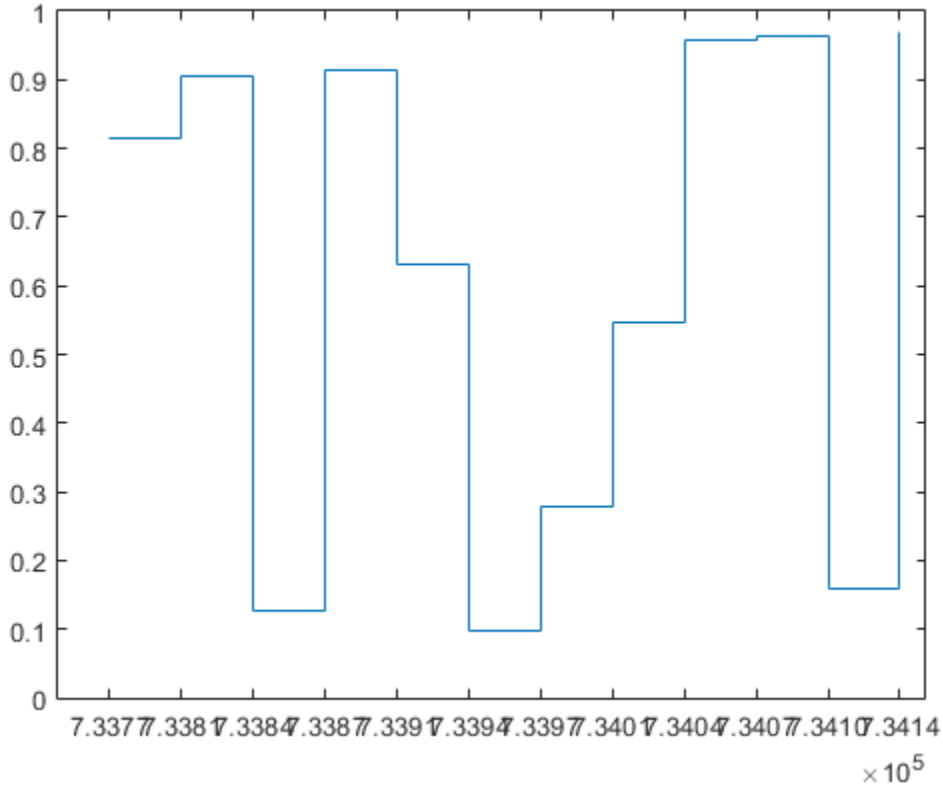
Plot random data.

```
figure
stairs(xData,rand(1,12))
```



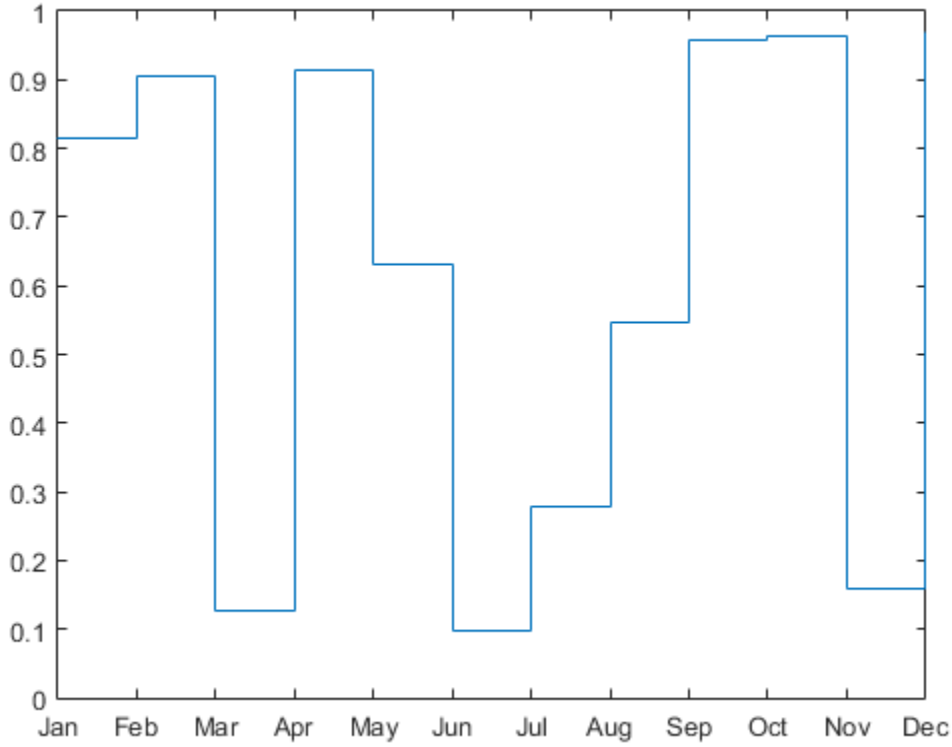
Set the number of `XTicks` to the number of points in `xData`.

```
ax = gca;
ax.XTick = xData;
```



Label the *x*-axis with month names, preserving the total number of ticks by using the `'keepticks'` option.

```
datetick('x','mmm','keepticks')
```



Create Multiple Plots Within Figure and Label Axis with Month Names

Select a starting date and an ending date.

```
startDate = datenum('01-01-2009');
endDate = datenum('12-31-2009');
```

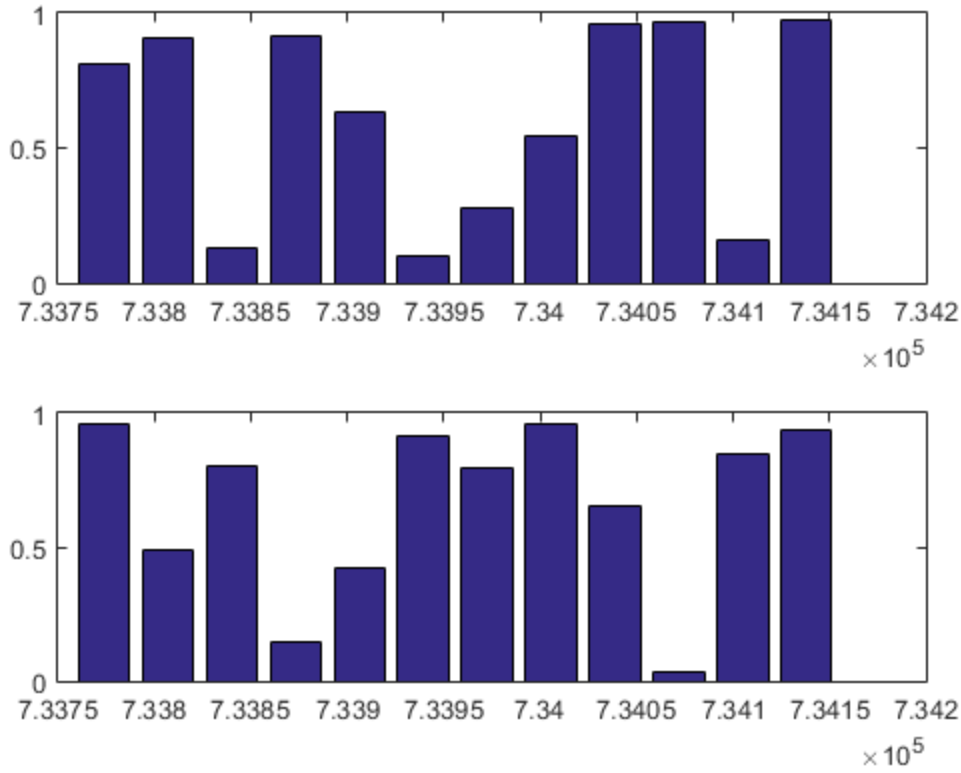
Create a variable, `xdata`, that corresponds to the number of months between the start and end dates.

```
xData = linspace(startDate,endDate,12);
```

Plot random data.

```
ax1 = subplot(2,1,1);
```

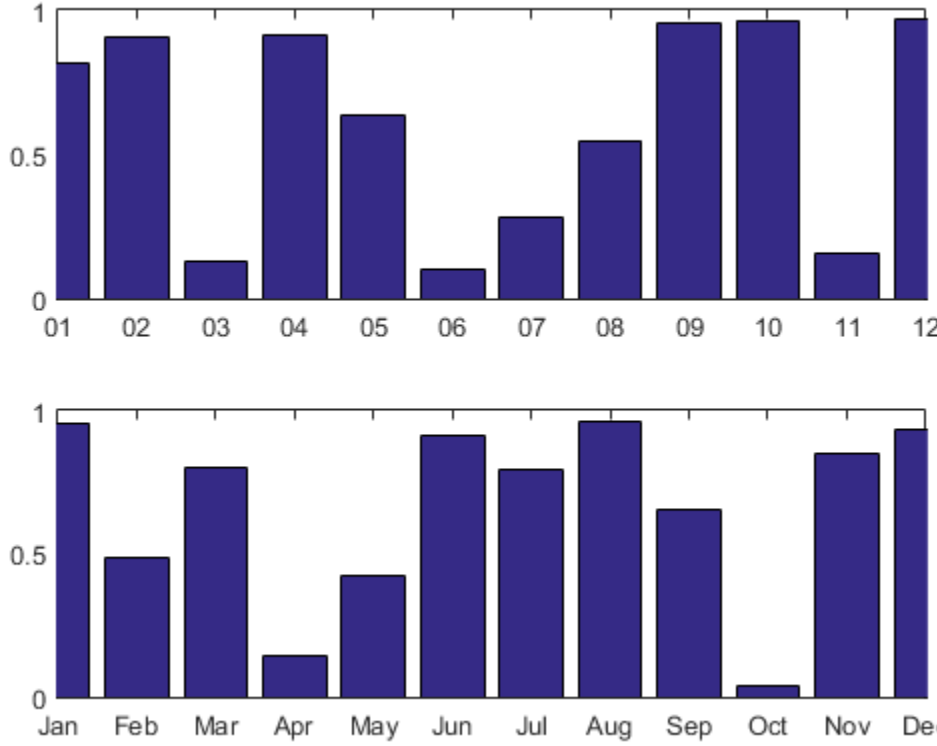
```
bar(xData,rand(1,12))
ax2 = subplot(2,1,2);
bar(xData,rand(1,12))
```



Set the number of `XTicks` to the number of points in `xData`. Label the *x*-axis of each subplot with month names, referring to each subplot using its axes handle. Preserve the total number of ticks by using the `'keepticks'` option. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
ax1.XTick = xData;
datetick(ax1,'x','mm','keepticks')

ax2.XTick = xData;
```

```
datetick(ax2,'x','mmm','keepticks')
```



Input Arguments

`tickaxis` — Axis to label

`'x'` (default) | `'y'` | `'z'`

Axis to label with dates, specified as `'x'`, `'y'`, or `'z'`.

`dateFormat` — Format of tick line labels

string | integer

Format of the tick line labels, specified as a string of symbolic identifiers or an integer that corresponds to a predefined format.

The following table shows symbolic identifiers that you can use to construct the `dateFormat` string. You can include characters such as a hyphen, space, or colon to separate the fields. For example, to display the day of the month followed by the three-letter abbreviation of the day of the week in parentheses, use `dateFormat = 'dd (ddd)'`.

Note: The letter identifiers that `datetick` accepts are different from the identifiers used by the `datetime` function.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	`03`, `12`
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`
`dd`	Day in two digits	`05`, `20`
`d`	Day using capitalized first letter	`M`, `T`
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	`05`, `5 AM`
`MM`	Minute in two digits	`12`, `02`
`SS`	Second in two digits	`07`, `59`
`FFF`	Millisecond in three digits	`057`

Symbolic Identifier	Description	Example
AM or PM	AM or PM inserted in date string	3:45:02 PM

The following table lists predefined MATLAB date formats.

Numeric Identifier	Date String Format	Example
-1 (default)	'dd-mmm-yyyy HH:MM:SS' or 'dd-mmm-yyyy' if 'HH:MM:SS'= 00:00:00	01-Mar-2000 15:45:17 or 01-Mar-2000
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1-01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00

Numeric Identifier	Date String Format	Example
21	`'mmm.dd,yyyy HH:MM:SS'`	`Mar.01,2000 15:45:17`
22	`'mmm.dd,yyyy'`	`Mar.01,2000`
23	`'mm/dd/yyyy'`	`03/01/2000`
24	`'dd/mm/yyyy'`	`01/03/2000`
25	`'yy/mm/dd'`	`00/03/01`
26	`'yyyy/mm/dd'`	`2000/03/01`
27	`'QQ-YYYY'`	`Q1-2001`
28	`'mmmyyyy'`	`Mar2000`
29	`'yyyy-mm-dd'` (ISO 8601)	`2000-03-01`
30	`'yyyymmddTHHMMSS'` (ISO 8601)	`20000301T154517`
31	`'yyyy-mm-dd HH:MM:SS'`	`2000-03-01 15:45:17`

More About

Tips

- To change the tick spacing and locations, set the appropriate axes property (that is, `XTick`, `YTick`, or `ZTick`) before calling `datetick`.
- Calling `datetick` sets the `TickMode` of the specified axis to `'manual'`. This means that after zooming, panning or otherwise changing axis limits, you should call `datetick` again to update the ticks and labels.
- The best way to work with dates and times in MATLAB is to use `datetime` values, which offer more features than serial date numbers. Plot `datetime` values using the `plot` function. Use the `DatetimeTickFormat` name-value pair argument to modify the format of the axis tick labels.

Algorithms

`datetick` calls the `datestr` function to convert date numbers to date strings.

- “Plot Dates and Durations”

See Also

`datenum` | `datestr` | `datetime` | `plot`

Introduced before R2006a

datetime

Create array based on current date, or convert from date strings or numbers

The `datetime` function creates an array that represents points in time using the Gregorian calendar. `datetime` values have flexible display formats up to nanosecond precision and can account for time zones, daylight saving time, and leap seconds.

Syntax

```
t = datetime
t = datetime(relativeDay)

t = datetime(DateStrings)
t = datetime(DateStrings,'InputFormat',infmt)

t = datetime(DateVectors)
t = datetime(Y,M,D)
t = datetime(Y,M,D,H,MI,S)

t = datetime(X,'ConvertFrom',dateType)

t = datetime( ___ ,Name,Value)
```

Description

`t = datetime` returns a scalar `datetime` array corresponding to the current date and time.

`t = datetime(relativeDay)` uses the date specified by `relativeDay`. The `relativeDay` input can be `'today'`, `'tomorrow'`, `'yesterday'`, or `'now'`.

`t = datetime(DateStrings)` creates an array of datetime values representing points in time from the strings in `DateStrings`.

`t = datetime(DateStrings,'InputFormat',infmt)` interprets the strings in `DateStrings` using the format specified by `infmt`. All strings in `DateStrings` must have the same format.

To avoid ambiguities between similar formats, specify `'InputFormat'` and its corresponding value, `infmt`.

`t = datetime(DateVectors)` creates a column vector of datetime values from the date vectors in `DateVectors`.

`t = datetime(Y,M,D)` creates an array of datetime values for corresponding elements of the `Y`, `M`, and `D` (year, month, day) arrays. The arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y,M,D]`.

`t = datetime(Y,M,D,H,MI,S)` creates an array of `datetime` values for corresponding elements of the `Y`, `M`,`D`,`H`, `MI`, and `S` (year, month, day, hour, minute, and second) arrays. The arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y,M,D,H,MI,S]`.

`t = datetime(X,'ConvertFrom',dateType)` converts the numeric values in `X` to a `datetime` array, `t`. The `dateType` argument specifies the type of values in `X`.

`t = datetime(___,Name,Value)` specifies additional options using one or more name-value pair arguments, in addition to any of the input arguments in the previous syntaxes. For example, you can specify the display format of `t` using the `'Format'` name-value pair argument.

For best performance when creating datetime values from strings, specify either `'Format'` or `'InputFormat'` and its corresponding value, `infmt`.

Examples

Current Date and Time in Specific Time Zone

Specify the current date and time in the local system time zone.

```
t = datetime('now','TimeZone','local','Format','d-MMM-y HH:mm:ss Z')
```

```
t =

   23-Feb-2015 09:59:16 -0500
```

Specify the current date and time in the time zone represented by Seoul, Korea

```
t = datetime('now','TimeZone','Asia/Seoul','Format','d-MMM-y HH:mm:ss Z')
```

```
t =

   23-Feb-2015 23:59:16 +0900
```

Date and Time from Strings

Create a `datetime` array from a cell array of two strings.

```
DateStrings = {'2014-05-26';'2014-08-03'};
t = datetime(DateStrings,'InputFormat','yyyy-MM-dd')
```

```
t =

   26-May-2014
   03-Aug-2014
```

The datetime values in `t` display using the default format, and not the format of the input date strings.

Date and Time from Strings with Literal Characters

Convert date strings in ISO 8601 format to datetime values.

Create a cell array of strings containing dates in ISO 8601 format. In this format, the letter T is used as a delimiter that separates a date and a time. Each string includes a time zone offset. The letter Z indicates no offset from UTC.

```
DateStrings = {'2014-05-26T13:30-05:00';'2014-08-26T13:30-04:00';'2014-09-26T13:30Z'}
```

```
DateStrings =

    '2014-05-26T13:30-05:00'
    '2014-08-26T13:30-04:00'
    '2014-09-26T13:30Z'
```

Convert the strings to datetime values. When specifying the input format, enclose the letter T in single quotes to indicate that it is a literal character. Specify the time zone of the output datetime array using the `TimeZone` name-value pair argument.

```
t = datetime(DateStrings,'InputFormat','uuuu-MM-dd''T''HH:mmXXX','TimeZone','UTC')
```

```
t =

   26-May-2014 18:30:00
   26-Aug-2014 17:30:00
   26-Sep-2014 13:30:00
```

The datetime values in `t` display in the default format.

Date and Time from Strings in Foreign Language

Create a cell array of strings containing dates in French.

```
C = {'8 avril 2013','9 mai 2013';'10 juin 2014','11 juillet 2014'}
```

```
C =

    '8 avril 2013'    '9 mai 2013'
    '10 juin 2014'    '11 juillet 2014'
```

Convert the strings in `C` to datetime values. If your computer is set to a locale that uses English, you must specify the `'Locale'` name-value pair argument to indicate that the strings are in French.

```
t = datetime(C,'InputFormat','d MMMM yyyy','Locale','fr_FR')
```

```
t =

   08-Apr-2013   09-May-2013
   10-Jun-2014   11-Jul-2014
```

The datetime values in `t` display in the default format, and in the language MATLAB uses depending on your system locale.

Date and Time from Vectors

Create a `datetime` array from individual arrays of year, month, and day values.

Create sample numeric arrays of year values Y and day values D. In this case, the month value M is a scalar.

```
Y = [2014;2013;2012];
M = 01;
D = [31;30;31];
```

Create the `datetime` array.

```
t = datetime(Y,M,D)
```

```
t =

   31-Jan-2014
   30-Jan-2013
   31-Jan-2012
```

Specify a custom display format for the output, using the `Format` name-value pair argument.

```
t = datetime(Y,M,D,'Format','eeee, MMMM d, y')
```

```
t =

   Friday, January 31, 2014
   Wednesday, January 30, 2013
   Tuesday, January 31, 2012
```

Convert Excel Date Number to Datetime

Create a sample array of Excel® date numbers that represent a number of days since January 0, 1900.

```
X = [39558, 39600; 39700, 39800]
```

```
X =

       39558       39600
       39700       39800
```

Convert the values in X to datetime values.

```
t = datetime(X,'ConvertFrom','excel')
```

```
t =

   20-Apr-2008 00:00:00   01-Jun-2008 00:00:00
   09-Sep-2008 00:00:00   18-Dec-2008 00:00:00
```

Input Arguments

`relativeDay` — Day relative to current date

`'yesterday' | 'today' | 'tomorrow' | 'now'`

Day relative to the current date, specified as one of the following strings.

Value of `relativeDay`	Description
`'yesterday'`	Date of the previous day, at midnight
`'today'`	Current date, at midnight
`'tomorrow'`	Date of the following day, at midnight
`'now'`	Current date and time

`DateStrings` — Date strings

string | character array | cell array of strings

Date strings, specified as a string, character array, or a cell array of strings. The `datetime` function first attempts to match the format of the strings to some common formats. If you know the format of the date strings, you should specify `'InputFormat'` and its corresponding value, `infmt`, or the `'Format'` name-value pair argument.

Example: `'24-Oct-2014 12:45:07'`

Example: `{'15-Oct-2013' '20-Nov-2014'}`

Data Types: `char | cell`

`infmt` — Format of input date strings

string

Format of the input date strings, specified as a string of letter identifiers:

- If `infmt` does not include a date specifier, then `datetime` assumes that the values in `DateStrings` occur during the current day.
- If `infmt` does not include a time specifier, then `datetime` assumes that the values in `DateStrings` occur at midnight.
- If `infmt` includes a time-zone specifier and you do not specify a time zone, then `datetime` creates unzoned datetimes and ignores any time-zone information in `DateStrings`.

This table shows several common input formats and examples of the formatted input for the date, Saturday, April 19, 2014 at 9:41:06 PM in New York City.

Value of `Format`	Example
`'yyyy-MM-dd'`	`2014-04-19`
`'dd/MM/yyyy'`	`19/04/2014`
`'dd.MM.yyyy'`	`19.04.2014`
`'yyyy# MM# dd#'`	`2014# 04# 19#`
`'MMMM d, yyyy'`	`April 19, 2014`
`'eeee, MMMM d, yyyy h:mm a'`	`Saturday, April 19, 2014 9:41 PM`
`'MMMM d, yyyy HH:mm:ss Z'`	`April 19, 2014 21:41:06 -0400`
`'yyyy-MM-dd''T''HH:mmXXX'`	`2014-04-19T21:41-04:00`

For a complete list of valid letter identifiers, see the `Format` property for datetime arrays.

Note: The letter identifiers that `datetime` accepts are different from those used by the `datestr`, `datenum`, and `datevec` functions.

Data Types: `char`

`DateVectors` — Date vectors

matrix

Date vectors, specified as an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. A full date vector has six elements, specifying year, month, day,

hour, minute, and second, in that order. A partial date vector has three elements, specifying year, month, and day, in that order. Each element of `DateVector` should be a positive or negative integer value with the exception of the seconds element, which can be fractional. If an element falls outside the conventional range, `datetime` adjusts both that date vector element and the previous element. For example, if the minutes element is `70`, then `datetime` adjusts the hours element by `1` and sets the minutes element to `10`. If the minutes element is `-15`, then `datetime` decreases the hours element by `1` and sets the minutes element to `45`.

Example: `[2014,10,24,12,45,07]`

Example: `[2014,10,24]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y,M,D` — Year, month, and day arrays
scalar | vector | matrix | array

Year, month, and day arrays specified as a scalar, vector, matrix, or array. These must be the same size, or any one can be a scalar. `Y,M,D` should be integer values.

If `Y,M,D` are all scalars or all column vectors, you can specify the input arguments as a date vector, `[Y,M,D]`.

If an element of the `Y`, `M`, or `D` inputs falls outside the conventional range, then `datetime` adjusts both that element and the same element of the previous input. For details, see the description for the `DateVectors` input argument.

Example: `2003,10,24`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y,M,D,H,MI,S` — Year, month, day, hour, minute, and second arrays
scalar | vector | matrix | array

Year, month, day, hour, minute, and second arrays specified as a scalar, vector, matrix, or array. These must be the same size, or any one can be a scalar. Specify fractional seconds as part of the seconds input, `S`. The `Y,M,D,H,MI` arrays should contain integer values.

If `Y,M,D,H,MI,S` are all scalars or all column vectors, you can specify the input arguments as a date vector`[Y,M,D,H,MI,S]`.

If an element of the `Y`, `M`, `D`, `H`, `MI`, or `S` inputs falls outside the conventional range, then `datetime` adjusts both that element and the same element of the previous input. For details, see the description for the `DateVectors` input argument.

Example: `2003,10,24,12,45,07.451`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`X` — Numeric values

scalar | matrix | multidimensional array

Numeric values, specified as a scalar, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`dateType` — Type of values in `X`

`'datenum'` | `'excel'` | `'juliandate'` | `'posixtime'` | `'yyyymmdd'` | ...

Type of values in `X`, specified as one of the following strings.

Value of `dateType`	Type of values in `X`
`'datenum'`	Number of days since 0-Jan-0000 (Gregorian calendar)
`'excel'`	Number of days since 0-Jan-1900 Excel date numbers are rounded to the nearest microsecond.
`'excel1904'`	Number of days since 0-Jan-1904 Excel date numbers are rounded to the nearest microsecond.
`'juliandate'`	Number of days since noon UTC 24-Nov-4712 BCE (Gregorian calendar)
`'modifiedjuliandate'`	Number of days since midnight UTC 17-Nov-1858
`'posixtime'`	Number of seconds since 1-Jan-1970 00:00:00 UTC

Value of dateType	Type of values in X
'yyyymmdd'	Dates as YYYYMMDD numeric values. For example, 20140402 represents April 2, 2014.
'epochtime','Epoch',*epochValue*	Number of seconds since an epoch. You must additionally specify *epochValue*, which is a scalar datetime or a date string representing the epoch time. This example returns the number of days since January 1, 2000: `T = datetime(X,'ConvertFrom',... 'epochtime','Epoch','2000-01-01')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Format','eeee MMMM d, y','TimeZone','local' applies a display format to datetime values and specifies the local time zone.

'Format' — Display format

'default' | 'defaultdate' | 'preserveinput' | string

Display format of the values in the output array, y, specified as the comma-separated pair consisting of 'Format' and one of the following strings.

Value of Format	Description
'default'	Use the default display format.
'defaultdate'	Use the default display format for datetime values created without time components.
'preserveinput'	Use the format specified by the input format, infmt. If you do not specify infmt, then datetime determines the format automatically.

The factory default format depends on your system locale. To change the default display format, see “Default datetime Format”.

Alternatively, use the letters `A-Z` and `a-z` to construct a custom value for `Format`. These letters correspond to the Unicode® Locale Data Markup Language (LDML) standard for dates. You can include non-ASCII or nonletter characters such as a hyphen, space, or colon to separate the fields. To include the letters `A-Z` and `a-z` as literal characters in the format, enclose them with single quotes.

Example: `'Format','eeee, MMMM d, yyyy HH:mm:ss'` displays a date and time such as `Saturday, April 19, 2014 21:41:06`.

This table shows several common display formats and examples of the formatted output for the date, Saturday, April 19, 2014 at 9:41:06 PM in New York City. For a complete list of valid letter identifiers, see the `Format` property for datetime arrays.

Value of `Format`	Example
`'yyyy-MM-dd'`	`2014-04-19`
`'dd/MM/yyyy'`	`19/04/2014`
`'dd.MM.yyyy'`	`19.04.2014`
`'yyyy# MM# dd#'`	`2014# 04# 19#`
`'MMMM d, yyyy'`	`April 19, 2014`
`'eeee, MMMM d, yyyy h:mm a'`	`Saturday, April 19, 2014 9:41 PM`
`'MMMM d, yyyy HH:mm:ss Z'`	`April 19, 2014 21:41:06 -0400`
`'yyyy-MM-dd''T''HH:mmXXX'`	`2014-04-19T21:41-04:00`

Note: The letter identifiers that `datetime` accepts are different from those used by the `datestr`, `datenum`, and `datevec` functions.

If you specify a `DateStrings` input but do not specify the `'InputFormat'` parameter, then `datetime` tries to use the `Format` value to interpret the strings.

Data Types: `char`

`'Locale'` — Locale of `DateStrings`

string

Locale of the date strings in `DateStrings`, specified as the comma-separated pair consisting of `'Locale'` and a string. The `Locale` value determines how `datetime` interprets the strings in `DateStrings`. However, the output `datetime` values always display in the system locale.

The `Locale` value can be:

- `'system'`, to specify your system locale.
- a string in the form *`xx_YY`*, where *`xx`* is a lowercase ISO 639-1 two-letter code that specifies a language, and *`YY`* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

This table lists some common values for the locale.

Locale	Language	Country
`'de_DE'`	German	Germany
`'en_GB'`	English	United Kingdom
`'en_US'`	English	United States
`'es_ES'`	Spanish	Spain
`'fr_FR'`	French	France
`'it_IT'`	Italian	Italy
`'ja_JP'`	Japanese	Japan
`'ko_KR'`	Korean	Korea
`'nl_NL'`	Dutch	Netherlands
`'zh_CN'`	Chinese (simplified)	China

You can use the `'Locale'` name-value pair only when you use the `DateStrings` input argument.

Example: `'Locale','de_DE'`

Data Types: `char`

`'PivotYear'` — Start year of 100-year date range

`year(datetime('now'))-50` (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as the comma-separated pair consisting of `'PivotYear'` and an integer. Use a pivot year to

interpret date strings that specify the year as two characters. That is, the pivot year has an effect only when the `infmt` argument includes `y` or `yy`.

You can use the `'PivotYear'` name-value pair only when you use the `DateStrings` input argument.

Example: `'PivotYear',1900`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'TimeZone'` — Time zone region

`''` (default) | string

Time zone region, specified as the comma-separated pair consisting of `'TimeZone'` and a string. The value of `TimeZone` specifies the time zone that the `datetime` function uses to interpret the input data. `TimeZone` also specifies the time zone of the output array, `T`. If the input data are strings that include a time zone, then the `datetime` function converts all values to the specified time zone

The value of `TimeZone` can be:

- `''`, to create an "unzoned" `datetime` array that does not belong to a specific time zone.
- The name of a time zone region from the IANA Time Zone Database, for example, `'America/Los_Angeles'`. The name of a time zone region accounts for the current and historical rules for standard and daylight offsets from UTC that are observed in a geographic region.
- An ISO 8601 string of the form `+HH:mm` or `-HH:mm`, for example, `'+01:00'`, to specify a time zone that is a fixed offset from UTC.
- `'UTC'`, to create a `datetime` array in Universal Coordinated Time.
- `'UTCLeapSeconds'`, to create a `datetime` array in Universal Coordinated Time that accounts for leap seconds.
- `'local'`, to create a `datetime` array in the system time zone.

This table lists some common names of time zone regions from the IANA Time Zone Database.

Value of `TimeZone`	UTC Offset	UTC DST Offset
`'Africa/Johannesburg'`	+02:00	+02:00

Value of TimeZone	UTC Offset	UTC DST Offset
'America/Chicago'	−06:00	−05:00
'America/Denver'	−07:00	−06:00
'America/Los_Angeles'	−08:00	−07:00
'America/New_York'	−05:00	−04:00
'America/Sao_Paulo'	−03:00	−02:00
'Asia/Hong_Kong'	+08:00	+08:00
'Asia/Kolkata'	+05:30	+05:30
'Asia/Tokyo'	+09:00	+09:00
'Australia/Sydney'	+10:00	+11:00
'Europe/London'	+00:00	+01:00
'Europe/Zurich'	+01:00	+02:00

Data Types: char

Output Arguments

t — Dates and time

datetime array

Dates and time, returned as a datetime array. Each element includes a date and a time of day.

t has the following properties.

Property	Description
Format	Display format of the values in t, specified as a string
TimeZone	Time zone in which the values in t are interpreted, specified as a string
Year	Array containing the year number of each element in t
Month	Array containing the month number of each element in t
Day	Array containing the day of month number of each element in t

Property	Description
`Hour`	Array containing the hour of each element in `t`
`Minute`	Array containing the minute of each element in `t`
`Second`	Array containing the second, including a fractional part, of each element in `t`

After you create a `datetime` array, `t`, you can access or change the value of any of its properties using dot notation. For example, to display the values of `t` in the format, `'yyyy-MM-dd'`, type:

```
t.Format = 'yyyy-MM-dd';
```

More About

- "Represent Dates and Times in MATLAB"
- "Share Code and Data Across Locales"

See Also

datetime Properties

Introduced in R2014b

datetime Properties

Assign date and time components and specify datetime display format

The `Format` property controls the display of `datetime` values. The other properties control the values of components such as years and months in the `datetime` array. Use dot notation to refer to a particular array and property:

```
t = datetime(2014,07,01,06,0,0);
t.Format = 'MMMM d, y';
```

Properties

`Format` — Display format

`'default'` | `'defaultdate'` | string

Display format, specified as a string of the letters `A-Z` and `a-z`, that correspond to the Unicode Locale Data Markup Language (LDML) standard for dates.

Example: `'eeee, MMMM d, yyyy HH:mm:ss'` displays a date and time such as `Saturday, April 5, 2014 21:41:06`.

The following tables show the letters you can use to construct the value for `Format`. You can include nonletter characters such as a hyphen, space, colon, or any non-ASCII characters to separate the fields. To include the letters `A-Z` and `a-z` as literal characters in the format, enclose them with single quotes.

Example: `'uuuu-MM-dd''T''HH:mm:ss'` displays a date and time, such as `2014-04-05T09:41:06`.

The examples display the formatted output for the date, Saturday, April 5, 2014 at 9:41:06.12345 PM, in New York City.

Date and Time Formats

Use these identifiers to specify the display formats of date and time fields.

Letter Identifier	Description	Display
`G`	Era	`CE`

Letter Identifier	Description	Display
y	Year, with no leading zeros. See the Note that follows this table.	2014
yy	Year, using last two digits. See the Note that follows this table.	14
yyy, yyyy ...	Year, using at least the number of digits specified by the number of instances of 'y'	For the year 2014, 'yyy' displays 2014, while 'yyyyy' displays 02014.
u, uu, ...	ISO year. A single number designating the year. An ISO year value assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.	2014
Q	Quarter, using one digit	2
QQ	Quarter, using two digits	02
QQQ	Quarter, abbreviated	Q2
QQQQ	Quarter, full name	2nd quarter
M	Month, numerical using one or two digits	4
MM	Month, numerical using two digits	04
MMM	Month, abbreviated name	Apr
MMMM	Month, full name	April
MMMMM	Month, capitalized first letter	A
W	Week of the month	1
d	Day of the month, using one or two digits	5
dd	Day of the month using two digits	05
D	Day of the year, using one, two or three digits	95
DD	Day of the year using two digits	95
DDD	Day of the year using three digits	095

Letter Identifier	Description	Display
e	Day of the week, numerical using one or two digits.	7, where Sunday is the first day of the week.
ee	Day of the week, numerical using two digits	07
eee	Day, abbreviated name	Sat
eeee	Day, full name	Saturday
eeeee	Day, capitalized first letter	S
a	Day period (AM or PM)	PM
h	Hour, 12-hour clock notation using one or two digits	9
hh	Hour, 12-hour clock notation using two digits	09
H	Hour, 24-hour clock notation using one or two digits	21
HH	Hour, 24-hour clock notation using two digits	21
m	Minute, using one or two digits	41
mm	Minute, using two digits	41
s	Second, using one or two digits	6
ss	Second, using two digits	06
S, SS, ..., SSSSSSSSS	Fractional second, using the number of digits specified by the number of instances of 'S' (up to 9 digits).	'SSS' truncates 6.12345 seconds to 123.

Note: If you read a two-digit year number and specify the format as y or yy, then the pivot year determines the century to which the year belongs.

Use one or more u characters instead of y characters to represent the year when working with year numbers near zero.

Datetime values later than 144683 years CE or before 140743 BCE display only the year numbers, regardless of the specified `Format` value.

Time Zone Offset Formats

Use these identifiers to specify the display format of the time zone offset. A time zone offset is the amount of time that a specific datetime is offset from UTC. This is different from a time zone, which comprises rules that determine the offsets that are used at specific times of the year. Include a time zone offset identifier in the display format for a datetime array when you want to ensure that the time components are displayed unambiguously.

Letter Identifier	Description	Display
`z`	Abbreviated name of the time zone offset. If this value is not available, then the time zone offset uses the short UTC format, such as `UTC-4`.	`EDT`
`Z`	ISO 8601 basic format with hours, minutes, and optional seconds fields.	`-0400`
`ZZZZ`	Long UTC format.	`UTC-04:00`
`ZZZZZ`	ISO 8601 extended format with hours, minutes, and optional seconds fields. A time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04:00`
`x` or `X`	ISO 8601 basic format with hours field and optional minutes field. If you specify `X`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04`
`xx` or `XX`	ISO 8601 basic format with hours and minutes fields. If you specify `XX`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-0400`
`xxx` or `XXX`	ISO 8601 extended format with hours and minutes fields. If you	`-04:00`

Letter Identifier	Description	Display
	specify `XXX`, a time offset of zero is displayed as the ISO 8601 UTC indicator "Z".	
`xxxx` or `XXXX`	ISO 8601 basic format with hours, minutes, and optional seconds fields. If you specify `XXXX`, a time offset of zero is displayed as the ISO 8601 UTC indicator "Z".	`-0400`
`xxxxx` or `XXXXX`	ISO 8601 extended format with hours, minutes, and optional seconds fields. If you specify `XXXXX`, a time offset of zero is displayed as the ISO 8601 UTC indicator "Z".	`-04:00`

`TimeZone` — Time zone

`''` (default) | string

Time zone, specified as one of the following strings:

- `''`, to create an "unzoned" `datetime` array that does not belong to a specific time zone.
- The name of a time zone region from the IANA Time Zone Database, for example, `'America/Los_Angeles'`. The name of a time zone region accounts for the current and historical rules for standard and daylight offsets from UTC that are observed in a geographic region.
- An ISO 8601 string of the form `+HH:mm` or `-HH:mm`, for example, `'+01:00'`, to specify a time zone that is a fixed offset from UTC.
- `'UTC'`, to create a `datetime` array in Universal Coordinated Time.
- `'UTCLeapSeconds'`, to create a `datetime` array in Universal Coordinated Time that accounts for leap seconds.
- `'local'`, to create a `datetime` array in the system time zone. When you query the `TimeZone` property, the IANA string is returned.

This table lists some common names of time zone regions from the IANA Time Zone Database.

Value of TimeZone	UTC Offset	UTC DST Offset
'Africa/Johannesburg'	+02:00	+02:00
'America/Chicago'	−06:00	−05:00
'America/Denver'	−07:00	−06:00
'America/Los_Angeles'	−08:00	−07:00
'America/New_York'	−05:00	−04:00
'America/Sao_Paulo'	−03:00	−02:00
'Asia/Hong_Kong'	+08:00	+08:00
'Asia/Kolkata'	+05:30	+05:30
'Asia/Tokyo'	+09:00	+09:00
'Australia/Sydney'	+10:00	+11:00
'Europe/London'	+00:00	+01:00
'Europe/Zurich'	+01:00	+02:00

Data Types: char

Year — Year number

scalar | vector | matrix | multidimensional array

Year number of each value in the datetime array, specified as a scalar or a vector, matrix, or multidimensional array the same size and shape as the datetime array. Each year number is an integer value based on the ISO calendar. Years in the current era are positive and years in the previous era are zero or negative. For example, the year number of 1 BCE is 0.

If you set the Year property to a nonleap year for a datetime value that occurs on a leap day (February 29), then the Day and Month properties change to March 1.

Month — Month number

scalar | vector | matrix | multidimensional array

Month number of each value in the datetime array, specified as a scalar or a vector, matrix, or multidimensional array the same size and shape as the datetime array. Each month number is an integer value from 1 to 12. If you set a value outside that range, then the Year property adjusts accordingly, and the Month property stays within the

range 1 to 12. For example, month 0 corresponds to month 12 of the previous year. For historical dates, the month number is based on the proleptic Gregorian calendar.

`Day` — Day number

scalar | vector | matrix | multidimensional array

Day number of each value in the `datetime` array, specified as a scalar or a vector, matrix, or multidimensional array the same size and shape as the `datetime` array. Each day number is an integer value from 1 to 28, 29, 30, or 31, depending on the month and year. If you set a value outside that range, then the `Month` and `Year` properties adjust accordingly, and the `Day` property stays within the appropriate range. For example, day 0 corresponds to the last day of the previous month. For historical dates, the day number is based on the proleptic Gregorian calendar.

`Hour` — Hour number

scalar | vector | matrix | multidimensional array

Hour number of each value in the `datetime` array, specified as a scalar or a vector, matrix, or multidimensional array the same size and shape as the `datetime` array. Each hour number is an integer value from 0 to 23. If you set a value outside that range, then the `Day`, `Month`, and `Year` properties adjust accordingly, and the `Hour` property stays within the appropriate range. For example, hour -1 corresponds to hour 23 of the previous day.

The following apply to datetime arrays with a specific time zone that follows daylight saving time:

- If you specify a value for the `Hour` property that would create a nonexistent datetime in the hour gap when daylight saving time begins, the value of the `Hour` property adjusts to the next hour.
- If you specify a value for the `Hour` property that would create an ambiguous datetime in the hour overlap when daylight saving time ends, then the datetime adjusts to the second of the two times (in standard time) with that hour.

`Minute` — Minute number

scalar | vector | matrix | multidimensional array

Minute number of each value in the `datetime` array, specified as a scalar or a vector, matrix, or multidimensional array the same size and shape as the `datetime` array. Each minute number is an integer value from 0 to 59. If you specify a value outside that range, then the `Hour`, `Day`, `Month`, and `Year` properties adjust accordingly, and the `Minute`

property stays within the appropriate range. For example, minute -1 corresponds to minute 59 of the previous hour.

`Second` — Second

scalar | vector | matrix | multidimensional array

Second of each value in the `datetime` array, specified as a scalar or a vector, matrix, or multidimensional array the same size and shape as the `datetime` array. Each second value is a floating-point value ordinarily ranging from 0 to less than 60. If you set a value outside that range, then the `Minute`, `Hour`, `Day`, `Month`, and `Year` properties adjust accordingly, and the `Second` property stays within the appropriate range. For example, second -1 corresponds to second 59 of the previous minute.

A `datetime` array with a `TimeZone` value of `'UTCLeapSeconds'` has seconds ranging from 0 to less than 61. The values from 60 to 61 represent datetimes that occur during a leap second.

`SystemTimeZone` — System time zone setting

string

System time zone setting, specified as a string. This time zone setting is determined by the system on which MATLAB is running.

Example: `America/New_York`

See Also

`datetime`

datevec

Convert date and time to vector of components

The `datevec` function creates a numeric array whose values represent the date and time components of years, months, days, hours, minutes, and seconds. However, the best way to represent points in time is by using the `datetime` data type. The best way to represent elapsed time is by using the `duration` or `calendarDuration` data types.

Syntax

```
DateVector = datevec(t)
DateVector = datevec(DateNumber)

DateVector = datevec(DateString)
DateVector = datevec(DateString,formatIn)
DateVector = datevec(DateString,PivotYear)
DateVector = datevec(DateString,formatIn,PivotYear)

[Y,M,D,H,MN,S] = datevec( ___ )
```

Description

`DateVector = datevec(t)` converts the datetime values in `datetime` array `t` to date vectors. `datevec` returns an `m`-by-6 matrix where each row corresponds to a datetime value in `t`.

`DateVector = datevec(DateNumber)` converts one or more date numbers to date vectors. `datevec` returns an `m`-by-6 matrix containing `m` date vectors, where `m` is the total number of date numbers in `DateNumber`.

`DateVector = datevec(DateString)` converts date strings to date vectors. If the date string format is known, use `formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

`DateVector = datevec(DateString,formatIn)` uses `formatIn` to interpret each date string.

`DateVector = datevec(DateString,PivotYear)` uses `PivotYear` to interpret date strings that specify the year as two characters. If the date string format is known, use `formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

`DateVector = datevec(DateString,formatIn,PivotYear)` uses `formatIn` to interpret each date string, and `PivotYear` to interpret date strings that specify the year as two characters. You can specify `formatIn` and `PivotYear` in either order.

`[Y,M,D,H,MN,S] = datevec( ___ )` returns the components of the date vector as individual variables `Y`, `M`, `D`, `H`, `MN`, and `S` (year, month, day, hour, minutes and seconds). `datevec` returns milliseconds as a fractional part of the seconds (`S`) output.

Examples

Convert datetime Array to Date Vectors

```
format short g

t = [datetime('now');datetime('tomorrow')]
DateVector = datevec(t)
```

```
t =

   23-Feb-2015 09:59:20
   24-Feb-2015 00:00:00


DateVector =

         2015           2          23           9          59       20.352
         2015           2          24           0           0            0
```

Convert Date Number to Date Vector

```
format short g

n = 733779.651;
datevec(n)
```

```
ans =

        2009           1           6          15          37        26.4
```

Convert Date String to Date Vector

```
DateString = '28.03.2005';
formatIn = 'dd.mm.yyyy';
datevec(DateString,formatIn)
```

```
ans =

        2005           3          28           0           0           0
```

`datevec` returns a date vector for the date string with the format `'dd.mm.yyyy'`.

Convert Multiple Date Strings to Date Vectors

Pass multiple date strings in a cell array. All input date strings must use the same format.

```
DateString = {'09/16/2007';'05/14/1996';'11/29/2010'};
formatIn = 'mm/dd/yyyy';
datevec(DateString,formatIn)
```

```
ans =

        2007           9          16           0           0           0
        1996           5          14           0           0           0
        2010          11          29           0           0           0
```

Convert Date with Milliseconds to Date Vector

```
datevec('11:21:02.647','HH:MM:SS.FFF')
```

```
ans =

   1.0e+03 *

    2.0150    0.0010    0.0010    0.0110    0.0210    0.0026
```

In the output date vector, milliseconds are a fractional part of the seconds field. The date string `'11:21:02.647'` does not contain enough information to convert to a full date vector. The days default to 1, months default to January, and years default to the current year.

Convert Date String to Date Vector Using Pivot Year

Convert a date string to a date vector using the default pivot year.

```
DateString = '12-jun-17';
formatIn = 'dd-mmm-yy';
DateVector = datevec(DateString,formatIn)
```

```
DateVector =

        2017           6          12           0           0           0
```

Convert the same date string to a date vector using 1800 as the pivot year.

```
DateVector = datevec(DateString,formatIn,1800)
```

```
DateVector =

        1817           6          12           0           0           0
```

Assign Elements of Returned Date Vector

Convert a date string to a date vector and return the components of the date vector.

```
[y, m, d, h, mn, s] = datevec('01.02.12','dd.mm.yy')
```

```
y =

        2012
```

```
m =
```

```
     2

d =

     1

h =

     0

mn =

     0

s =

     0
```

Input Arguments

`t` — `datetime` values

scalar | vector | matrix | multidimensional array

`datetime` values, specified as a scalar, vector, matrix, or multidimensional `datetime` array.

`DateNumber` — Serial date number

scalar | vector | multidimensional array

Serial date number, specified as a scalar, vector, or multidimensional array of positive double-precision numbers.

Example: `731878`

Data Types: `double`

`DateString` — Date strings

string | character array | 1–D cell array of strings

Date strings, specified as a character array where each row corresponds to one date string, or as a one dimensional cell array of strings. All of the date strings must have the same format.

Example: `'24–Oct-2003 12:45:07'`

Example: `['19-Sep-2013';'20-Sep-2013';'21-Sep-2013']`

Example: `{'15-Oct-2010' '20-Nov-2012'}`

If the date string format is known, you should also specify `formatIn`. If you do not specify `formatIn`, `DateString` must be in one of the following formats.

Date String Format	Example
`'dd-mmm-yyyy HH:MM:SS'`	`01-Mar-2000 15:45:17`
`'dd-mmm-yyyy'`	`01-Mar-2000`
`'mm/dd/yyyy'`	`03/01/2000`
`'mm/dd/yy'`	`03/01/00`
`'mm/dd'`	`03/01`
`'mmm.dd,yyyy HH:MM:SS'`	`Mar.01,2000 15:45:17`
`'mmm.dd,yyyy'`	`Mar.01,2000`
`'yyyy-mm-dd HH:MM:SS'`	`2000-03-01 15:45:17`
`'yyyy-mm-dd'`	`2000-03-01`
`'yyyy/mm/dd'`	`2000/03/01`
`'HH:MM:SS'`	`15:45:17`
`'HH:MM:SS PM'`	`3:45:17 PM`
`'HH:MM'`	`15:45`
`'HH:MM PM'`	`3:45 PM`

Note: The symbolic identifiers describing date string formats are different from those that describe the display formats of `datetime` arrays.

Certain date string formats might not contain enough information to convert the date string. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. `datevec` and `datenum`

consider two-character date string years (e.g., `'79'`) to fall within the 100-year range centered around the current year.

When you do not specify `formatIn`, note the following:

- For the formats that specify the month as two digits (`mm`), the month value must not be greater than 12.
- However, for the format `'mm/dd/yy'`, if the first entry in the date string is greater than 12 and the second entry is less than or equal to 12, then `datevec` considers the date string to be in `'yy/mm/dd'` format.

`formatIn` — Format of the input date string

string

Format of the input date string, specified as a string of symbolic identifiers.

The following table shows symbolic identifiers you can use to construct the `formatIn` string. You can include characters such as a hyphen, space, or colon to separate the fields.

Note: The symbolic identifiers describing date string formats are different from those that describe the display formats of `datetime` arrays.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	`03`, `12`
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`

Symbolic Identifier	Description	Example
dd	Day in two digits	05, 20
d	Day using capitalized first letter	M, T
HH	Hour in two digits (no leading zeros when symbolic identifier AM or PM is used)	05, 5 AM
MM	Minute in two digits	12, 02
SS	Second in two digits	07, 59
FFF	Millisecond in three digits	057
AM or PM	AM or PM inserted in date string	3:45:02 PM

The `formatIn` string must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- `datevec` does not accept formats that include `'QQ'`

`PivotYear` — Start year of 100-year date range

present minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret date strings that specify the year as two characters.

If `formatIn` contains the time of day, the pivot year is computed from the current time of the current day, month, and year. Otherwise it is computed from midnight of the current day, month, and year.

Example: `2000`

Data Types: `double`

Output Arguments

`DateVector` — Date vectors
vector | matrix

Date vectors, returned as an `m`-by-6 matrix, where each row corresponds to one date vector, and `m` is the total number of input date numbers or date strings.

`[Y,M,D,H,MN,S]` — Components of the date vector
scalar

Components of the date vector (year, month, day, hour, minute, and second), returned as individual scalar variables. Each variable is a scalar or a vector. Milliseconds are a fractional part of the seconds output. When converting a `datetime` array `t`, these components are equal to the values of the `Year`, `Month`, `Day`, `Hour`, `Minute`, and `Second` properties. For example, `Y = t.Year`.

Limitations

- When computing date vectors, `datevec` sets month values less than 1 to 1. Day values, `D`, less than 1 are set to the last day of the previous month minus `|D|`. However, if $0 \le$ DateNumber < 1, then `datevec(DateNumber)` returns a date vector of the form `[0 0 0 H MN S]`, where `H`, `MN`, and `S` are hours, minutes, and seconds, respectively.

More About

Tips

- The vectorized calling syntax can offer significant performance improvement for large arrays.

- "Represent Dates and Times in MATLAB"
- "Carryover in Date Vectors and Strings"

See Also

`datenum` | `datestr` | `datetime`

Introduced before R2006a

day

Day number or name

Syntax

```
d = day(t)
d = day(t,dayType)
```

Description

`d = day(t)` returns the day-of-month numbers for the datetime values in `t`. The `d` output contains integer values from 1 to 31, depending on the month and year.

`d = day(t,dayType)` returns the type of day number or name specified by `dayType`.

The `day` function returns the day numbers or names of datetime values. To assign day values to datetime array `t`, use `t.Day` and modify the `Day` property.

Examples

Extract Day Number from Dates

Extract the day of month numbers from an array of dates.

```
t = [datetime('yesterday');datetime('today');datetime('tomorrow')]
d = day(t)
```

```
t =

   22-Feb-2015
   23-Feb-2015
   24-Feb-2015


d =
```

```
    22
    23
    24
```

Determine Day of Week

Determine the day of the week for an arbitrary date, by specifying `'name'` as the second input to the `day` function.

```
t = datetime(2014,05,16)
```

```
t =

   16-May-2014
```

```
d = day(t,'name')
```

```
d =

    'Friday'
```

Alternatively, specify `'dayofweek'` to return the day of the week as a number.

```
d = day(t,'dayofweek')
```

```
d =

     6
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

dayType — Type of day values

'dayofmonth' (default) | 'dayofweek' | 'dayofyear' | 'name' | 'shortname'

Type of day values, specified as one of the following strings.

Value of dayType	Description
'dayofmonth'	Day-of-month number, from 1 to 28, 29, 30, or 31. The range depends on the month.
'dayofweek'	Day-of-week number, from 1 to 7, where day 1 of the week is Sunday.
'dayofyear'	Day-of-year number, from 1 to 365 or 366, depending on the year.
'name'	Full day names, for example, Sunday.
'shortname'	Abbreviated day names, for example, Sun.

Output Arguments

d — Day number or name

double array | cell array of strings

Day number or name, returned as a numeric array of integers of type double, or a cell array of strings. d is the same size as t.

See Also

datetime Properties | isweekend | month | quarter | week | year | ymd

Introduced in R2014b

days

Duration in days

Syntax

```
D = days(X)
```

Description

`D = days(X)` returns an array of days equivalent to the values in `X`.

- If `X` is a numeric array, then `D` is a `duration` array in units of fixed-length days. A fixed-length day is equal to 24 hours.
- If `X` is a `duration` array, then `D` is a `double` array with each element equal to the number of fixed-length (24-hour) days in the corresponding element of X.

The `days` function converts between `duration` and `double` values. To display a duration in units of days, set its `Format` property to `'d'`.

Examples

Create Duration Array of Fixed-Length Days

```
X = magic(2);
D = days(X)
```

```
D =

    1 day   3 days
   4 days   2 days
```

Add each number of fixed-length days to the current date and time.

```
t = datetime('now') + D
```

```
t =

   24-Feb-2015 10:14:16   26-Feb-2015 10:14:16
   27-Feb-2015 10:14:16   25-Feb-2015 10:14:16
```

Convert Durations to Numeric Array of Days

Create a `duration` array.

```
X = hours(23:20:95) + minutes(45)
```

```
X =

   23.75 hrs   43.75 hrs   63.75 hrs   83.75 hrs
```

Convert each duration in `X` to a number of days.

```
D = days(X)
```

```
D =

    0.9896    1.8229    2.6563    3.4896
```

Input Arguments

`X` — Input array

numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

More About

Tips

- `days` creates fixed-length (24 hour) days. To create days that account for Daylight Saving Time shifts when used in calendar calculations, use the `caldays` function.

See Also

`caldays` | `duration`

Introduced in R2014b

dbclear

Clear breakpoints

Syntax

```
dbclear all
dbclear in file ...
dbclear in file ... -completenames
dbclear if error ...
dbclear if warning ...
dbclear if naninf
dbclear if infnan
```

Description

`dbclear all` removes all breakpoints in all MATLAB code files, as well as breakpoints set for errors, caught errors, caught error identifiers, warnings, warning identifiers, and `naninf`/`infnan`.

`dbclear in file ...` and `dbclear in file ... -completenames` formats are listed here:

Format	Action
`dbclear in file`	Removes all breakpoints in `file`. `file` must be the name of a MATLAB program file, and can include a MATLAB partial path.
`dbclear in file -completenames`	If the command includes the `-completenames` option, then `file` need not be on the path, as long as it is a fully qualified file name. (On Microsoft Windows platforms, this is a file name that begins with `\\` or with a drive letter followed by a colon. On UNIX platforms, this is a file name that begins with `/` or `~`.) `file` can include a `>` to specify the path to a particular local function or to a nested function within a code file.
`dbclear in file at lineno`	Removes the breakpoint set at line number `lineno` in `file`.
`dbclear in file at lineno@`	Removes the breakpoint set in the anonymous function at line number `lineno` in `file`.

Format	Action
`dbclear in file at lineno@n`	Removes the breakpoint set in the `n`th anonymous function at line number `lineno` in `file`.
`dbclear in file at locfun`	Removes all breakpoints in local function `locfun` in `file`.

`dbclear if error ...` formats are listed here:

Format	Action
`dbclear if error`	Removes the breakpoints set using the `dbstop if error` and `dbstop if error identifier` statements.
`dbclear if error identifier`	Removes the breakpoint set using `dbstop if error identifier` for the specified `identifier`. Running this produces an error if `dbstop if error` or `dbstop if error all` is set.
`dbclear if caught error`	Removes the breakpoints set using the `dbstop if caught error` and `dbstop if caught error identifier` statements.
`dbclear if caught error identifier`	Removes the breakpoints set using the `dbstop if caught error identifier` statement for the specified identifier. Running this produces an error if `dbstop if caught error` or `dbstop if caught error all` is set.

`dbclear if warning ...` formats are listed here:

`dbclear if warning`	Removes the breakpoints set using the `dbstop if warning` and `dbstop if warning identifier` statements.
`dbclear if warning identifier`	Removes the breakpoint set using `dbstop if warning identifier` for the specified `identifier`. Running this produces an error if `dbstop if warning` or `dbstop if warning all` is set.

`dbclear if naninf` removes the breakpoint set by `dbstop if naninf` or `dbstop if infnan`.

`dbclear if infnan` removes the breakpoint set by `dbstop if infnan` or `dbstop if naninf`.

More About

Tips

The **at** and **in** keywords are optional.

In the syntax, `file` can be a MATLAB program file, or the path to a function within a file. For example

```
dbclear in foo>myfun
```

clears the breakpoint at the `myfun` function in the file `foo.m`.

See Also

`dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbtype` | `dbup` | `filemarker`

Related Examples

- "Disable and Clear Breakpoints"

Introduced before R2006a

dbcont

Resume execution

Syntax

```
dbcont
```

Description

`dbcont` resumes execution of a MATLAB code file from a breakpoint. Execution continues until another breakpoint is encountered, a pause condition is met, an error occurs, or MATLAB software returns to the base workspace prompt.

Note If you want to edit a file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the file. If you edit a file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

See Also

`dbclear` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbtype` | `dbup`

Related Examples

- “Step Through a File”

Introduced before R2006a

dbdown

Reverse workspace shift performed by dbup, while in debug mode

Syntax

```
dbdown
```

Description

dbdown changes the current workspace context to the workspace of the called MATLAB code file when a breakpoint is encountered. You must have issued the dbup function at least once before you issue this function. dbdown is the opposite of dbup.

Multiple dbdown functions change the workspace context to each successively executed MATLAB code file on the stack until the current workspace context is the current breakpoint. It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.

See Also

dbclear | dbcont | dbquit | dbstack | dbstatus | dbstep | dbstop | dbtype | dbup

Related Examples

- "Debugging Process and Features"

Introduced before R2006a

dblquad

Numerically evaluate double integral over rectangle

Compatibility

`dblquad` will be removed in a future release. Use `integral2` instead.

Syntax

```
q = dblquad(fun,xmin,xmax,ymin,ymax)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)
```

Description

`q = dblquad(fun,xmin,xmax,ymin,ymax)` calls the `quad` function to evaluate the double integral `fun(x,y)` over the rectangle `xmin <= x <= xmax`, `ymin <= y <= ymax`. The input argument, `fun`, is a function handle that accepts a vector `x`, a scalar `y`, and returns a vector of integrand values.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples

Pass function handle `@integrnd` to `dblquad`:

```
Q = dblquad(@integrnd,pi,2*pi,0,pi);
```

where the function `integrnd.m` is:

```
function z = integrnd(x, y)
z = y*sin(x)+x*cos(y);
```

Pass anonymous function handle `F` to `dblquad`:

```
F = @(x,y)y*sin(x)+x*cos(y);
Q = dblquad(F,pi,2*pi,0,pi);
```

The `integrnd` function integrates `y*sin(x)+x*cos(y)` over the square `pi <= x <= 2*pi`, `0 <= y <= pi`. Note that the integrand can be evaluated with a vector `x` and a scalar `y`.

Nonsquare regions can be handled by setting the integrand to zero outside of the region. For example, the volume of a hemisphere is:

```
dblquad(@(x,y)sqrt(max(1-(x.^2+y.^2),0)), -1, 1, -1, 1)
```

or

```
dblquad(@(x,y)sqrt(1-(x.^2+y.^2)).*(x.^2+y.^2<=1), -1, 1, -1, 1)
```

More About

- "Anonymous Functions"

See Also

`quad2d` | `quad` | `quadgk` | `quadl` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

Introduced before R2006a

dbmex

Enable MEX-file debugging (on UNIX platforms)

Syntax

```
dbmex on
dbmex off
dbmex stop
```

Description

`dbmex on` enables MEX-file debugging for UNIX platforms.

To use this option, first start MATLAB from a debugger by typing `matlab -D`*`debugger`*, where *`debugger`* is the name of the debugger program. Invoke `dbmex on` before calling your MEX-file. If you have already loaded the MEX-file, remove it from memory using the `clear` function.

`dbmex off` disables MEX-file debugging.

`dbmex stop` returns to the debugger prompt.

More About

- "Debugging on Mac Platforms"
- "Debugging on Linux Platforms"

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbtype` | `dbup`

Introduced before R2006a

dbquit

Quit debug mode

Syntax

```
dbquit
dbquit('all')
dbquit all
```

Description

`dbquit` terminates debug mode. The Command Window then displays the standard prompt (`>>`). The file being processed is *not* completed and no results are returned. All breakpoints remain in effect, unless your file contains (and MATLAB has processed) one or more of the following commands: `clear all`, `clear function`, `clear classes`.

If you debug `file1` and step into `file2`, running `dbquit` terminates debugging for both files. However, if you debug `file3` and also debug `file4`, running `dbquit` terminates debugging for `file4`, but `file3` remains in debug mode until you run `dbquit` again.

`dbquit('all')` or the command form, `dbquit all`, ends debugging for all files at once.

Examples

This example illustrates the use of `dbquit` relative to `dbquit('all')`. Set breakpoints in and run `file1` and `file2`:

```
>> dbstop in file1
>> dbstop in file2
>> file1
K>> file2
K>> dbstack
```

MATLAB software returns

```
K>> dbstack
  In file1 at 11
```

```
  In file2 at 22
```

If you use the `dbquit` syntax

```
K>> dbquit
```

MATLAB ends debugging for `file2` but `file1` is still in debug mode as shown here

```
K>> dbstack
    in file1 at 11
```

Run `dbquit` again to exit debug mode for `file1`.

Alternatively, `dbquit('all')` ends debugging for both files at once:

```
K>> dbstack
  In file1 at 11
  In file2 at 22
dbquit('all')
dbstack
```

returns no result.

See Also

`clear` | `dbclear` | `dbcont` | `dbdown` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbtype` | `dbup`

Introduced before R2006a

dbstack

Function call stack

Syntax

```
dbstack
dbstack(n)
dbstack('-completenames')
[ST,I] = dbstack(...)
```

Description

`dbstack` displays the line numbers and file names of the function calls that led to the current breakpoint, listed in the order in which they were executed. The display lists the line number of the most recently executed function call (at which the current breakpoint occurred) first, followed by its calling function, which is followed by its calling function, and so on. This continues until the topmost MATLAB function is reached. Each line number is a hyperlink you can click to go directly to that line in the Editor. The notation `functionname>localfunctionname` is used to describe the location of the local function.

`dbstack(n)` omits the first `n` frames from the display. This is useful when issuing a `dbstack` from within an error handler, for example.

`dbstack('-completenames')` outputs the "complete name" (the absolute file name and the entire sequence of functions that nests the function in the stack frame) of each function in the stack.

Either none, one, or both `n` and `'-completenames'` can appear. If both appear, the order is irrelevant.

`[ST,I] = dbstack(...)` returns the stack trace information in an `m`-by-`1` structure, `ST`, with the fields:

`file`	The file in which the function appears. This field is the empty string if there is no file.

`name`	Function name within the file.
`line`	Function line number.

The current workspace index is returned in `I`.

If you step past the end of a file, `dbstack` returns a negative line number value to identify that special case. For example, if the last line to be executed is line `15`, then the `dbstack` line number is `15` before you execute that line and `-15` afterwards.

Examples

This example shows the information returned when you issue `dbstack` while debugging a MATLAB code file:

```
dbstack

In /usr/local/matlab/toolbox/matlab/cond.m at line 13
In test1.m at line 2
In test.m at line 3
```

This example shows the information returned when you issue `dbstack` while debugging `lengthofline.m` to get the complete name of the file, the function name, and line number in which the function appears:

```
[ST,I] = dbstack('-completenames')
ST =
    file: 'I:\MATLABFiles\mymfiles\lengthofline.m'
    name: 'lengthofline'
    line: 28
I =
     1
```

More About

Tips

In addition to using `dbstack` while debugging, you can also use `dbstack` within a MATLAB code file outside the context of debugging. In this case, to get and analyze information about the current file stack. For example, to get the name of the calling file, use `dbstack` with an output argument within the file being called. For example:

```
st=dbstack;
```

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstatus` | `dbstep` | `dbstop` | `dbtype` | `dbup` | `evalin` | `mfilename` | `whos`

Introduced before R2006a

dbstatus

List all breakpoints

Syntax

```
dbstatus
dbstatus file
dbstatus('-completenames')
s = dbstatus(...)
```

Description

`dbstatus` lists all the breakpoints in effect including errors, caught errors, warnings, and `naninfs`.

`dbstatus file` displays a list of the line numbers for which breakpoints are set in the specified file, where `file` is a MATLAB code file function name or a MATLAB relative partial path. Each line number is a hyperlink you can click to go directly to that line in the Editor.

`dbstatus('-completenames')` displays, for each breakpoint, the absolute file name and the sequence of functions that nest the function containing the breakpoint.

`s = dbstatus(...)` returns breakpoint information in an `m`-by-`1` structure with the fields listed in the following table. Use this syntax to save breakpoint status and restore it at a later time using `dbstop(s)`—see `dbstop` for an example.

`name`	Function name.
`file`	Full path for file containing breakpoints.
`line`	Vector of breakpoint line numbers.
`anonymous`	Vector of integers representing the anonymous functions in the `line` field. For example, `2` means the second anonymous function in that line. A value of `0` means the breakpoint is at the start of the line, not in an anonymous function.

`expression`	Cell vector of breakpoint conditional expression strings corresponding to lines in the `line` field.
`cond`	Condition string (`'error'`, `'caught error'`, `'warning'`, or `'naninf'`).
`identifier`	When `cond` is `'error'`, `'caught error'`, or `'warning'`, a cell vector of MATLAB message identifier strings for which the particular `cond` state is set.

Use `dbstatus class/function`, `dbstatus private/function`, or `dbstatus class/private/function` to determine the status for methods, private functions, or private methods (for a class named `class`).

In all forms you can further qualify the function name with a local function name, as in `dbstatus function>localfunction`.

More About

Tips

In the syntax, `file` can be a file, or the path to a function within a file. For example

```
Breakpoint for foo>myfun is on line 9
```

means there is a breakpoint at the `myfun` local function, which is line 9 in the file `foo.m`.

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstep` | `dbstop` | `dbtype` | `dbup` | `error` | `warning`

Introduced before R2006a

dbstep

Execute one or more lines from current breakpoint

Syntax

```
dbstep
dbstep nlines
dbstep in
dbstep out
```

Description

This function allows you to debug a MATLAB code file by following its execution from the current breakpoint. At a breakpoint, the `dbstep` function steps through execution of the current file one line at a time or at the rate specified by `nlines`.

`dbstep` executes the next executable line of the current file. `dbstep` steps over the current line, skipping any breakpoints set in functions called by that line.

`dbstep nlines` executes the specified number of executable lines.

`dbstep in` steps to the next executable line. If that line contains a call to another MATLAB code file function, execution will step to the first executable line of the called function. If there is no call to a MATLAB code file on that line, `dbstep in` is the same as `dbstep`.

`dbstep out` runs the rest of the function and stops just after leaving the function.

For all forms, MATLAB software also stops execution at any breakpoint it encounters.

Note If you want to edit a file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the file. If you edit a file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstop` | `dbtype` | `dbup`

Related Examples

- "Step Through a File"

Introduced before R2006a

dbstop

Set breakpoints for debugging

Syntax

```
dbstop in file
dbstop in file at location
dbstop in file if expression
dbstop in file at location if expression
dbstop if condition
dbstop(s)
```

Description

`dbstop in file` sets a breakpoint at the first executable line in file. When you run file, MATLAB enters debug mode and pauses execution at the first executable line.

`dbstop in file at location` sets a breakpoint at the specified location. MATLAB execution pauses immediately before that location, unless the location is an anonymous function. If the location is an anonymous function, then execution pauses just after the breakpoint.

`dbstop in file if expression` sets a breakpoint at the first executable line of the file. Execution pauses only if `expression` evaluates to `1` (true).

`dbstop in file at location if expression` sets a breakpoint at the specified location. Execution pauses at or just before that location only if the expression evaluates to true.

`dbstop if condition` pauses execution at the line that meets the specified condition.

`dbstop(s)` restores breakpoints you previously saved to s. The files containing the saved breakpoints must be on the search path or in the current folder. MATLAB assigns breakpoints by line number; therefore, the lines in the file must be the same as when you saved the breakpoints, or the results are unpredictable.

Input Arguments

`file`

File specification, typically for a MATLAB function, specified as a string. The file specification can include a partial path, but must be in a folder on the search path, or in the current folder.

If the command includes the `-completenames` option, then the file need not be on the search path, as long as the file specification is a "fully qualified name" on page 1-2004. In addition, `file` can include a filemarker (`>`) to specify the path to a particular local function or to a nested function within the file.

If file is not a MATLAB code file (for instance, it is a built-in or MDL-file), then MATLAB issues a warning. MATLAB cannot stop *in* the file, so it pauses before executing the file.

`location`

Location in `file` where you want to set a breakpoint, specified as one of the following:

- lineno

  Line number in `file` specified as a string. The default is `1`.
- lineno@n

  `n`th anonymous function on line number, `lineno`, specified as a string. The default value of `n` is `1`.
- subfun

  Name of a local function in `file`, specified as a string.

`expression`

Code that evaluates (as if by `eval`) to a scalar logical value of `1` or `0`, (true or false, respectively).

`condition`

Condition that causes execution to pause when that condition evaluates to true. Specify `condition` as one of the following:

- `error`— Run-time error that occurs outside a `try/catch` block. You cannot resume execution after an uncaught run-time error.

If you want execution to pause only if a specific error occurs, specify the message id. For example:

- `dbstop if error` stops execution at the first run-time error that occurs outside a `try/catch` block.
- `dbstop if error MATLAB:ls:InputsMustBeStrings` pauses execution at the first run-time error outside a `try/catch` block that has a message ID of `MATLAB:ls:InputsMustBeStrings`.

- `caught error` — Run-time error that occurs within the `try` portion of a `try/catch` block. If you want execution to stop only if a specific error occurs, specify the message id. See the `error` condition for an example of specifying a message id.
- `warning`— Run-time warning occurs. If you want execution to pause only if a specific warning occurs, specify the message id. See the `error` condition for an example of specifying a message id.

  This condition has no effect if you disabled warnings with the `warning off all` command or if you disabled warnings for the specified `id`. For more information about disabling warnings, see `warning`.
- `naninf` and `infnan`

  The code returns an infinite value (`Inf`) or a value that is not a number (`NaN`) as a result of an operator, function call, or scalar assignment. The `naninf` and `infnan` conditions have identical effects.

`s`

Breakpoints previously saved to a structure using `s=dbstatus`.

Examples

Stop at First Executable Line

This example shows how to set a breakpoint and pause execution at the first executable line in `buggy.m`.

1 Create a file, `buggy.m`, which contains these statements.

```
function z = buggy(x)
```

```
n = length(x);
z = (1:n)./x;
```

2 Issue the `dbstop` command.

```
dbstop in buggy
```

3 Run buggy.

```
buggy(2:5)
```

MATLAB displays the line where it pauses and enters debug mode.

```
2   n = length(x);
K>>
```

4 Advance to the next line in the file.

```
dbstep
```

5 Examine the value of `n` in the Variable Editor.

```
openvar('n')
```

6 Quit debug mode.

```
dbquit
```

Stop if Error

This example shows how to set a breakpoint and pause execution if a run-time error occurs.

1 Create a file, `buggy.m`, which contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

2 Issue these statements in the Command Window:

```
dbstop if error
buggy(magic(3))
```

Because `buggy.m` works on vectors only, the input in step 2 results in a run-time error. MATLAB goes into debug mode, paused at line 3 in `buggy.m`.

```
Error using  ./
```

```
Matrix dimensions must agree.

Error in buggy at 3
z = (1:n)./x;
3   z = (1:n)./x;
```

3 Quit debug mode.

```
dbquit
```

Stop if InfNaN

This example shows how to set a breakpoint and pause execution if the code returns a NaN value.

1 Create a file, `buggy.m`, containing these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

2 Issue these commands.

```
dbstop if naninf
buggy(0:2)
```

If any of the elements of the input, x, is zero, a division by zero occurs in `buggy.m`. Therefore, MATLAB returns the following message.

```
NaN/Inf breakpoint hit for buggy on line 3.
```

MATLAB is in debug mode, paused at line 3.

Stop at Function in File

This example shows how to set a breakpoint in the `collatzall` program file at the first executable line in the `collatzplot_new` function.

1 Copy `collatzall.m` from the MATLAB examples folder to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','collatzall.m'),'.','f')
```

2 Open `collatzall.m` in the Editor.

```
open collatzall.m
```

3 Set the breakpoint.

```
dbstop in collatzall>collatzplot_new
```

Stop at File That Is Not a MATLAB Code File

This example shows how to set a breakpoint at the built-in function `clear` when you run `myfile.m`.

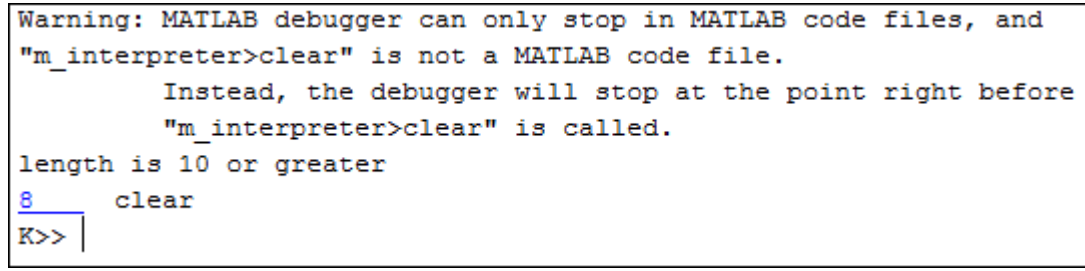
1 Create a file, `myfile.m`, in the Editor, containing these statements.

```
function myfile(x)
n = length(x);
if n < 10
    disp('length is less than 10')
else
    disp('length is 10 or greater')
end
clear
disp('Value of n is cleared')
```

2 Issue these commands to set the breakpoint.

```
x = [1 2 3 4 5 6 7 8 9 10]
dbstop in clear; myfile(x)
```

MATLAB issues a warning, and pauses before the call to the `clear` function. The warning appears as shown in this image.



Restore Saved Breakpoints

This example shows how to save, and then restore, saved breakpoints.

1. Copy `collatzall.m` from the MATLAB examples folder to your current folder.

   ```
   copyfile(fullfile(matlabroot,'help','techdoc','matlab_env', ...
       'examples','collatzall.m'),'.','f')
   ```

2. Open `collatzall.m` in the Editor.

   ```
   open collatzall.m
   ```

3. Set breakpoints from the Command Window.

   ```
   dbstop at 12 in collatzall
   dbstop if error
   ```

4. Run `dbstatus`.

   ```
   dbstatus
   ```

   MATLAB describes the breakpoints you set.

   ```
   Breakpoint for collatzall>collatzplot_new is on line 12.
   Stop if error.
   ```

5. Save the breakpoints to the structure, `s`, and then save `s` to the MAT-file, `myfilebrkpnts`.

   ```
   s=dbstatus('-completenames');
   save myfilebrkpnts s
   ```

   Using `s=dbstatus('-completenames')` saves absolute paths and the breakpoint function nesting sequence.

6. Clear all breakpoints.

   ```
   dbclear collatzall
   ```

7. Restore the breakpoints by loading the MAT-file.

   ```
   load myfilebrkpnts
   dbstop(s)
   ```

   The file (or files) containing the breakpoints must be on the search path or in the current folder.

8. Verify the breakpoints.

   ```
   dbstatus collatzall
   ```

More About

fully qualified name

- On Microsoft Windows platforms, a file name that begins with two back slashes (`\\`) or with a drive letter followed by a colon (`:`).
- On UNIX platforms, a file name that begins with a slash (`/`) or a tilde (`~`).

Tips

- Use `dbcont` or `dbstep` to resume execution after a breakpoint pauses execution. Use `dbquit` to exit debug mode.
- If you debug a file that MATLAB uses when running and debugging files, and that file is not a MATLAB code file, then some debugging features do not operate as expected. For instance, typing `help functionname` at the debug (`K>>`) prompt will not return help.
- MATLAB can become unresponsive when it stops at a breakpoint while displaying a modal dialog box or figure created by your program. Use **Ctrl+C** to exit debug mode and return to the MATLAB prompt (`>>`).
- If MATLAB pauses and displays a hyperlinked line number in the Command Window, click the hyperlink, The file opens in the Editor at the line where MATLAB paused execution. For an example of such a link, see the image in "Stop at File That Is Not a MATLAB Code File" on page 1-2002
- You can set breakpoints only at executable lines in saved files that are in the current folder or in folders on the search path.
- `dbstop` **`if warning`** has no effect when you disable warnings using `warning off all`. Similarly, `dbstop` **`if warning`** `identifier` has no effect when you disable warnings for the specified message identifier. See `warning` for more information about `off`, `all`, and `warning off identifier`

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbtype` | `dbup`

Introduced before R2006a

dbtype

List text file with line numbers

Syntax

```
dbtype filename
dbtype filename start:end
```

Description

The `dbtype` command is used to list a text file with line numbers, which is helpful when setting breakpoints in a MATLAB code file with `dbstop`.

`dbtype filename` displays the contents of the specified text file, with the line number preceding each line. `filename` must be the full path name of a file, or a MATLAB relative partial path.

`dbtype filename start:end` displays the portion of the file specified by a range of line numbers from `start` to `end`.

You cannot use `dbtype` for built-in functions.

Examples

To see only the input and output arguments for a function, that is, the first line of the file, use the syntax

```
dbtype filename 1
```

For example,

```
dbtype addpath 1
```

returns

```
1     function oldpath = addpath(varargin)
```

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbup`

Introduced before R2006a

dbup

Shift current workspace to workspace of caller, while in debug mode

Syntax

```
dbup
```

Description

This function allows you to examine the calling MATLAB code file to determine what caused the arguments to be passed to the called function.

`dbup` changes the current workspace context, while the user is in the debug mode, to the workspace of the calling file.

Multiple `dbup` functions change the workspace context to each previous calling file on the stack until the base workspace context is reached. (It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.)

More About

Tips

If your receive an error message such as the following, it means that the parent workspace is under construction so that the value of `x` is unavailable:

```
??? Reference to a called function result under construction x
```

- "Problems Viewing Variable Values from the Parent Workspace"

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbtype`

Introduced before R2006a

dde23

Solve delay differential equations (DDEs) with constant delays

Syntax

```
sol = dde23(ddefun,lags,history,tspan)
sol = dde23(ddefun,lags,history,tspan,options)
```

Arguments

`ddefun`	Function handle that evaluates the right side of the differential equations $y'(t) = f\left(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k)\right)$. The function must have the form `dydt = ddefun(t,y,Z)` where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(t-\tau_j)$ for delay τ_j = `lags(j)`. The output is a column vector corresponding to $f\left(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k)\right)$.
`lags`	Vector of constant, positive delays τ_1, ..., τ_k.
`history`	Specify `history` in one of three ways: • A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector • A constant column vector, if $y(t)$ is constant • The solution `sol` from a previous integration, if this call continues that integration
`tspan`	Interval of integration from `t0=tspan(1)` to `tf=tspan(end)` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = dde23(ddefun,lags,history,tspan)` integrates the system of DDEs

$$y'(t) = f\left(t, y(t), y(t-\tau_1), ..., y(t-\tau_k)\right)$$

on the interval $[t_0,t_f]$, where τ_1, ..., τ_k are constant, positive delays and t_0,t_f. The input argument, `ddefun`, is a function handle.

"Parameterizing Functions" explains how to provide additional parameters to the function `ddefun`, if necessary.

`dde23` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `dde23` has the following fields.

`sol.x`	Mesh selected by `dde23`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'dde23'`

`sol = dde23(ddefun,lags,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and "Types of DDEs" in the MATLAB documentation for details.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Jumps'` option to solve problems with discontinuities in the history or solution. Set this option to a vector that contains the locations of discontinuities in the solution prior to `t0` (the history) or in coefficients of the equations at known values of t after `t0`.

Use the `'Events'` option to specify a function that `dde23` calls to find where functions $g\left(t, y(t), y(t-\tau_1), ..., y(t-\tau_k)\right)$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the kth event function in `events`:

- `value(k)` is the value of the kth event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and 0 otherwise.
- `direction(k) = 0` if you want `dde23` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the `'Events'` option and events are detected, the output structure `sol` also includes fields:

`sol.xe`	Row vector of locations of all events, i.e., times when an event function vanished
`sol.ye`	Matrix whose columns are the solution values corresponding to times in `sol.xe`
`sol.ie`	Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

Examples

This example solves a DDE on the interval [0, 5] with lags 1 and 0.2. The function `ddex1de` computes the delay differential equations, and `ddex1hist` computes the history for `t <= 0`.

Note The file, ddex1.m, contains the complete code for this example. To see the code in an editor, type `edit ddex1` at the command line. To run it, type `ddex1` at the command line.

```
sol = dde23(@ddex1de,[1, 0.2],@ddex1hist,[0, 5]);
```

This code evaluates the solution at 100 equally spaced points in the interval `[0,5]`, then plots the result.

```
tint = linspace(0,5);
yint = deval(sol,tint);
```

```
plot(tint,yint);
```

`ddex1` shows how you can code this problem using local functions. For more examples see `ddex2`.

More About

Algorithms

`dde23` tracks discontinuities and integrates with the explicit Runge-Kutta (2,3) pair and interpolant of `ode23`. It uses iteration to take steps longer than the lags.

References

[1] Shampine, L.F. and S. Thompson, "Solving DDEs in MATLAB," *Applied Numerical Mathematics*, Vol. 37, 2001, pp. 441-458.

[2] Kierzenka, J., L.F. Shampine, and S. Thompson, "Solving Delay Differential Equations with DDE23," available at www.mathworks.com/dde_tutorial.

See Also

`ddesd` | `ddensd` | `ddeget` | `ddeset` | `deval` | `function_handle`

Introduced before R2006a

ddeget

Extract properties from delay differential equations options structure

Syntax

```
val = ddeget(options,'name')
val = ddeget(options,'name',default)
```

Description

`val = ddeget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = ddeget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = ddeget(opts,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `opts`.

See Also

`dde23` | `ddesd` | `ddensd` | `ddeset`

Introduced before R2006a

ddensd

Solve delay differential equations (DDEs) of neutral type

Syntax

```
sol = ddensd(ddefun,dely,delyp,history,tspan)
sol = ddensd(ddefun,dely,delyp,history,tspan,options)
```

Description

`sol = ddensd(ddefun,dely,delyp,history,tspan)` integrates a system of delay differential equations of neutral type, that has the form

$$y'(t) = f(t, y(t), y(dy_1),..., y(dy_p), y'(dyp_1),..., y'(dyp_q))$$

where

- t is the independent variable representing time.
- dy_i is any of p solution delays.
- dyp_j is any of q derivative delays.

`sol = ddensd(ddefun,dely,delyp,history,tspan,options)` replaces default integration parameters with those specified in `options`, a structure created with the `ddeset` function.

Examples

Neutral DDE with Two Delays

Solve the following neutral DDE, presented by Paul, for $0 \le t \le \pi$:
$y'(t) = 1 + y(t) - 2y(t/2)^2 - y'(t - \pi)$
with history: $y(t) = \cos(t)$ for $t \le 0$.

Create a new program file in the editor. This file will contain a main function and four local functions.

Define the first-order DDE as a local function.

```
function yp = ddefun(t,y,ydel,ypdel)
    yp = 1 + y - 2*ydel^2 - ypdel;
end
```

Define the solution delay as a local function.

```
function dy = dely(t,y)
    dy = t/2;
end
```

Define the derivative delay as a local function.

```
function dyp = delyp(t,y)
    dyp = t-pi;
end
```

Define the solution history as a local function.

```
function y = history(t)
    y = cos(t);
end
```

Define the interval of integration and solve the DDE using the `ddensd` function. Add this code to the main function.

```
tspan = [0 pi];
sol = ddensd(@ddefun,@dely,@delyp,@history,tspan);
```

Evaluate the solution at 100 equally spaced points between 0 and π. Add this code to the main function.

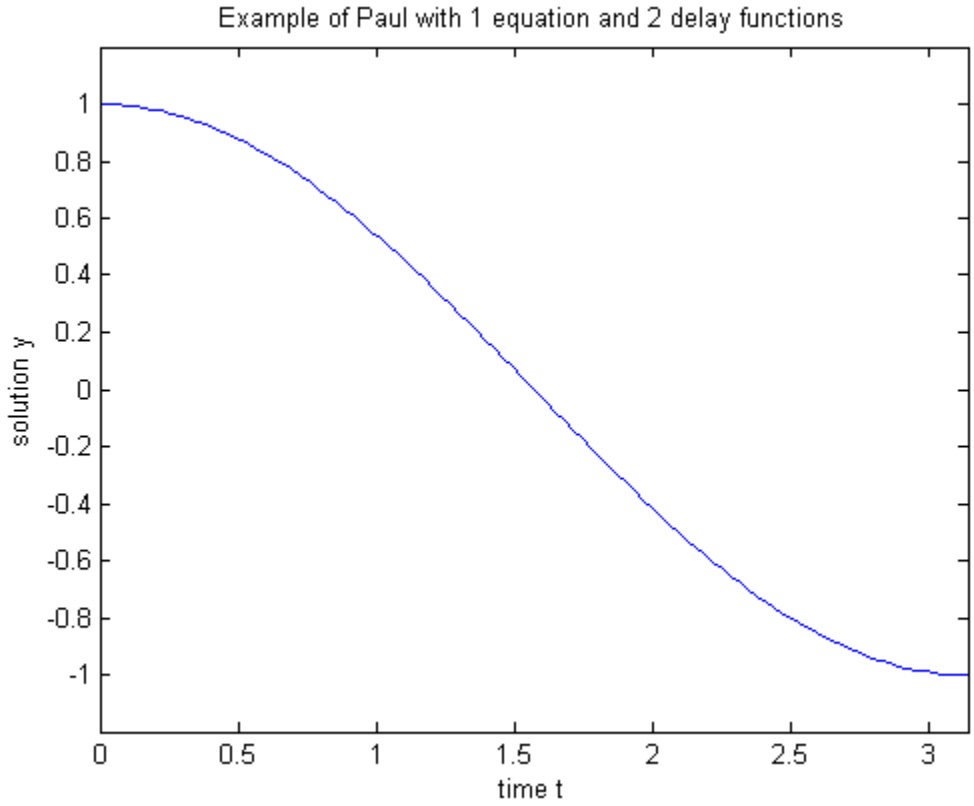
```
tn = linspace(0,pi);
yn = deval(sol,tn);
```

Plot the results. Add this code to the main function.

```
plot(tn,yn);
xlim([0 pi]);
ylim([-1.2 1.2]);
```

```
xlabel('time t');
ylabel('solution y');
```

Run your program to calculate the solution and display the plot.



The file, `ddex4.m`, contains the complete code for this example. To see the code in an editor, type `edit ddex4` at the command line.

Input Arguments

`ddefun` — Derivative function

function handle

Derivative function, specified as a function handle whose syntax is `yp = ddefun(t,y,ydel,ypdel)`. The arguments for `ddefun` are described in the table below.

ddefun Argument	Description
`t`	A scalar value representing the current value of time, t.
`y`	A vector that represents $y(t)$ in Equation 1-1. The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.
`ydel`	A matrix whose columns, `ydel(:,i)`, represent $y(dy_i)$. The size of this matrix is `n`-by-`p`, where `n` is the number of equations in the system you want to solve, and `p` is the number of $y(dy)$ terms in Equation 1-1.
`ypdel`	A matrix whose columns, `ypdel(:,j)` represent $y'(dyp_j)$. The size of this matrix is `n`-by-`q`, where `n` is the number of equations in the system you want to solve, and `q` is the number of $y'(dyp)$ terms in Equation 1-1.
`yp`	The result returned by `ddefun`. It is an `n`-by-`1` vector whose elements represent the right side of Equation 1-1.

`dely` — Solution delays

function handle | vector

Solution delays, specified as a function handle, which returns $dy_1,\ldots, dy_p$ in Equation 1-1. Alternatively, you can pass constant delays in the form of a vector.

If you specify `dely` as a function handle, the syntax must be `dy = dely(t,y)`. The arguments for this function are described in the table below.

dely Argument	Description
`t`	A scalar value representing the current value of time, t.
`y`	A vector that represents $y(t)$ in Equation 1-1. The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.
`dy`	A vector returned by the `dely` function whose values are the solution delays, dy_i , in Equation 1-1. The size of this vector is `p`-by-`1`, where `p` is the number of solution delays in the equation. Each element must be less than or equal to t.

If you want to specify constant solution delays having the form $dy_i = t - \tau_i$, then `dely` must be a vector, where `dely(i)` = τ_i. Each value in this vector must be greater than or equal to zero.

If *dy* is not present in the problem, set `dely` to `[]`.

Data Types: `function_handle` | `single` | `double`

`delyp` — Derivative delays

function handle | vector

Derivative delays, specified as a function handle, which returns dyp_1,..., dyp_q in Equation 1-1. Alternatively, you can pass constant delays in the form of a vector.

If `delyp` is a function handle, its syntax must be `dyp = delyp(t,y)`. The arguments for this function are described in the table below.

delyp Argument	Description
`t`	A scalar value representing the current value of time, *t*.
`y`	A vector that represents *y*(*t*) in Equation 1-1. The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.
`dyp`	A vector returned by the `delyp` function whose values are the derivative delays, dyp_j, in Equation 1-1. The size of this vector must be `q`-by-`1`, where `q` is the number of solution delays, dyp_j, in the equation. Each element of `dyp` must be less than *t*. There is one exception to this restriction: if you are solving an initial value DDE, the value of `dyp` can equal *t* at $t = t_0$. For more information, see “Initial Value Neutral Delay Differential Equations” on page 1-2019.

If you want specify constant derivative delays having the form $dyp_j = t - \tau_j$, then `delyp` must be a vector, where `delyp(j)` = τ_j. Each value in this vector must be greater than zero. An exception to this restriction occurs when you solve initial value problems for DDEs of neutral type. In such cases, a value in `delyp` can equal zero at $t = t_0$. See “Initial Value Neutral Delay Differential Equations” on page 1-2019 for more information.

If *dyp* is not present in the problem, set `delyp` to `[]`.

Data Types: `function_handle` | `single` | `double`

`history` — Solution history

function handle | column vector | structure (`sol`, from previous integration) | 1-by-2 cell array

Solution history, specified as a function handle, column vector, `sol` structure (from a previous integration), or a cell array. This is the solution at $t \le t_0$.

- If the history varies with time, specify the solution history as a function handle whose syntax is `y = history(t)`. This function returns an `n`-by-`1` vector that approximates the solution, $y(t)$, for $t <= t_0$. The length of this vector, `n`, is the number of equations in the system you want to solve.
- If $y(t)$ is constant, you can specify `history` as an `n`-by-`1` vector of the constant values.
- If you are calling `ddensd` to continue a previous integration to t_0, you can specify history as the output, `sol`, from the previous integration.
- If you are solving an initial value DDE, specify history as a cell array, `{y0, yp0}`. The first element, `y0`, is a column vector of initial values, $y(t_0)$. The second element, yp_0, is a column vector whose elements are the initial derivatives, $y'(t_0)$. These vectors must be consistent, meaning that they satisfy Equation 1-1 at t_0. See "Initial Value Neutral Delay Differential Equations" on page 1-2019 for more information.

Data Types: `function_handle` | `single` | `double` | `struct` | `cell`

`tspan` — Interval of integration

1-by-2 vector

Interval of integration, specified as the vector `[t0 tf]`. The first element, `t0`, is the initial value of t. The second element, `tf`, is the final value of t. The value of `t0` must be less than `tf`.

Data Types: `single` | `double`

`options` — Optional integration parameters

structure returned by `ddeset`

Optional integration parameters, specified as a structure created and returned by the `ddeset` function. Some commonly used properties are: `'RelTol'`, `'AbsTol'`, and `'Events'`. See the `ddeset` reference page for more information about specifying `options`.

Output Arguments

`sol` — Solution

structure

Solution, returned as a structure containing the following fields.

`sol.x`	Mesh selected by `ddensd`.
`sol.y`	An approximation to $y(t)$ at the mesh points.
`sol.yp`	An approximation to $y'(t)$ at the mesh points.
`sol.solver`	A string identifying the solver, `'ddensd'`.

You can pass `sol` to the `deval` function to evaluate the solution at specific points. For example, `y = deval(sol, 0.5*(sol.x(1) + sol.x(end)))` evaluates the solution at the midpoint of the interval of integration.

More About

Initial Value Neutral Delay Differential Equations

An initial value DDE has $dy_i \geq t_0$ and $dyp_j \geq t_0$, for all i and j. At $t = t_0$, all delayed terms reduce to $y(dy_i) = y(t_0)$ and $y'(dyp_j) = y'(t_0)$:

$$y'(t_0) = f(t_0, y(t_0), y(t_0),..., y(t_0), y'(t_0),..., y'(t_0))$$

For $t > t_0$, all derivative delays must satisfy $dyp < t$.

When you solve initial value neutral DDEs, you must supply $y'(t_0)$ to `ddensd`. To do this, specify history as a cell array `{Y0,YP0}`. Here, `Y0` is the column vector of initial values, $y(t_0)$, and `YP0` is a column vector of initial derivatives, $y'(t_0)$. These vectors must be consistent, meaning that they satisfy Equation 1-2 at t_0.

Algorithms

For information about the algorithm used in this solver, see Shampine [2].

References

[1] Paul, C.A.H. "A Test Set of Functional Differential Equations." *Numerical Analysis Reports*. No. 243. Manchester, UK: Math Department, University of Manchester, 1994.

[2] Shampine, L.F. "Dissipative Approximations to Neutral DDEs." *Applied Mathematics & Computation*. Vol. 203, Number 2, 2008, pp. 641–648.

See Also

`dde23` | `ddesd` | `ddeset` | `deval` | `function_handle`

ddesd

Solve delay differential equations (DDEs) with general delays

Syntax

```
sol = ddesd(ddefun,delays,history,tspan)
sol = ddesd(ddefun,delays,history,tspan,options)
```

Arguments

`ddefun`

Function handle that evaluates the right side of the differential equations $y'(t) = f(t,y(t),y(d(1)),...,y(d(k)))$. The function must have the form

```
dydt = ddefun(t,y,Z)
```

where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(d(j))$ for delay $d(j)$ given as component j of `delays(t,y)`. The output is a column vector corresponding to $f(t,y(t),y(d(1)),...,y(d(k)))$.

`delays`

Function handle that returns a column vector of delays $d(j)$. The delays can depend on both t and $y(t)$. `ddesd` imposes the requirement that $d(j) \le t$ by using `min`$(d(j),t)$.

If all the delay functions have the form $d(j) = t - \tau_j$, you can set the argument `delays` to a constant vector `delays`$(j) = \tau_j$. With delay functions of this form, `ddesd` is used exactly like `dde23`.

`history`

Specify `history` in one of three ways:

- A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector
- A constant column vector, if $y(t)$ is constant
- The solution `sol` from a previous integration, if this call continues that integration

`tspan`	Interval of integration from `t0=tspan(1)` to `tf=tspan(end)` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = ddesd(ddefun,delays,history,tspan)` integrates the system of DDEs

$$y'(t) = f\left(t, y(t), y(d(1)), \ldots, y(d(k))\right)$$

on the interval $[t_0,t_f]$, where delays $d(j)$ can depend on both t and $y(t)$, and $t_0 < t_f$. Inputs `ddefun` and `delays` are function handles. See the `function_handle` reference page for more information.

"Parameterizing Functions" explains how to provide additional parameters to the functions `ddefun`, `delays`, and `history`, if necessary.

`ddesd` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `ddesd` has the following fields.

`sol.x`	Mesh selected by `ddesd`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'ddesd'`

`sol = ddesd(ddefun,delays,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and "Types of DDEs" in the MATLAB documentation for details.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Events'` option to specify a function that `ddesd` calls to find where functions $g(t,y(t),y(d(1)),\ldots,y(d(k)))$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the `k`th event function in `events`:

- `value(k)` is the value of the `k`th event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and `0` otherwise.
- `direction(k) = 0` if you want `ddesd` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the `'Events'` option and events are detected, the output structure `sol` also includes fields:

`sol.xe`	Row vector of locations of all events, i.e., times when an event function vanished
`sol.ye`	Matrix whose columns are the solution values corresponding to times in `sol.xe`
`sol.ie`	Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

Examples

The equation

```
sol = ddesd(@ddex1de,@ddex1delays,@ddex1hist,[0,5]);
```

solves a DDE on the interval `[0,5]` with delays specified by the function `ddex1delays` and differential equations computed by `ddex1de`. The history is evaluated for $t \le 0$ by the function `ddex1hist`. The solution is evaluated at 100 equally spaced points in `[0,5]`:

```
tint = linspace(0,5);
yint = deval(sol,tint);
```

and plotted with

```
plot(tint,yint);
```

This problem involves constant delays. The `delay` function has the form

```
function d = ddex1delays(t,y)
%DDEX1DELAYS  Delays for using with DDEX1DE.
d = [ t - 1
      t - 0.2];
```

The problem can also be solved with the syntax corresponding to constant delays

```
delays = [1, 0.2];
sol = ddesd(@ddex1de,delays,@ddex1hist,[0, 5]);
```

or using `dde23`:

```
sol = dde23(@ddex1de,delays,@ddex1hist,[0, 5]);
```

For more examples of solving delay differential equations see ddex2 and ddex3.

References

[1] Shampine, L.F., "Solving ODEs and DDEs with Residual Control," *Applied Numerical Mathematics*, Vol. 52, 2005, pp. 113-127.

See Also

`dde23` | `ddeget` | `ddensd` | `ddeset` | `deval` | `function_handle`

ddeset

Create or alter delay differential equations options structure

Syntax

```
options = ddeset('name1',value1,'name2',value2,...)
options = ddeset(oldopts,'name1',value1,...)
options = ddeset(oldopts,newopts)
ddeset
```

Description

`options = ddeset('name1',value1,'name2',value2,...)` creates an integrator options structure `options` in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify the property. `ddeset` ignores case for property names.

`options = ddeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = ddeset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`ddeset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `ddeget` to query the `options` structure for the value of a specific property.

DDE Properties

The following sections describe the properties that you can set using `ddeset`. There are several categories of properties:

- Error control
- Solver output
- Step size
- Event location
- Discontinuities

Error Control Properties

At each step, the DDE solvers estimate an error `e`. The `dde23` function estimates the local truncation error, and the other solvers estimate the residual. In either case, this error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

```
|e(i)|*max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For the absolute error tolerance, the scaling of the solution components is important: if `|y|` is somewhat smaller than `AbsTol`, the solver is not constrained to obtain any correct digits in `y`. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.

The following table describes the error control properties.

DDE Error Control Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the solution vector `y`. It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except

Property	Value	Description
		those smaller than thresholds `AbsTol(i)`. The default, `1e-3`, corresponds to 0.1% accuracy. The estimated error in each integration step satisfies `|e(i)| max(RelTol*abs(y(i)),AbsTol(i))`.
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the individual components of the solution vector. `AbsTol(i)` is a threshold below which the value of the `i`th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components. If `AbsTol` is a vector, the length of `AbsTol` must be the same as the length of the solution vector `y`. If `AbsTol` is a scalar, the value applies to all components of `y`.
`NormControl`	`on` | {`off`}	Control error relative to norm of solution. Set this property `on` to request that the solvers control the error in each integration step with `norm(e)<= max(RelTol*norm(y),AbsTol)`. By default, the solvers use a more stringent component-wise error control.

Solver Output Properties

You can use the solver output properties to control the output that the solvers generate.

DDE Solver Output Properties

Property	Value	Description

`OutputFcn`	Function handle {`@odeplot`}	The output function is a function that the solver calls after every successful integration step. To specify an output function, set `'OutputFcn'` to a function handle. For example,
		
		`options = ddeset('OutputFcn',...`
		`@myfun)`

Property	Value	Description
		sets 'OutputFcn' to @myfun, a handle to the function myfun. See the function_handle reference page for more information. The output function must be of the form `status = myfun(t,y,flag)` "Parameterizing Functions" explains how to provide additional parameters to myfun, if necessary. The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately: • init — The solver calls myfun(tspan,y0,'init') before beginning the integration to allow the output function to initialize. tspan is the input argument to the solvers. y0 is the initial value of the solution, either from history(t0) or specified in the initialY option. • {none} — The solver calls status = myfun(t,y) after each integration step on which output is requested. t contains points where output was generated during the step, and y is the numerical solution at the points in t. If t is a vector, the ith column of y corresponds to the ith element of t. myfun must return a status output value of 0 or 1. If status = 1, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button. • done — The solver calls myfun([],[],'done') when integration is complete to allow the output function to perform any cleanup chores. You can use these general purpose output functions or you can edit them to create your own. Type help functionname at the command line for more information.

Property	Value	Description
		• `odeplot` – time series plotting (default when you call the solver with no output argument and you have not specified an output function) • `odephas2` – two-dimensional phase plane plotting • `odephas3` – three-dimensional phase plane plotting • `odeprint` – print solution as the solver computes it

`OutputSel`	Vector of indices	Vector of indices specifying which components of the solution vector the solvers pass to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using
		`options = ddeset...`
		`('OutputFcn',@odeplot,...`
		`'OutputSel',[1 3]);`
		By default, the solver passes all components of the solution to the output function.

`Stats`	`on` | {`off`}	Specifies whether the solver should display statistics about its computations. By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays: • The number of successful steps • The number of failed attempts • The number of times the DDE function was called

Step Size Properties

The step size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step size properties.

DDE Step Size Properties

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size. `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the solver bases the initial step size on the slope of the solution at the initial time `tspan(1)`. The initial step size is limited by the shortest delay. If the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.
`MaxStep`	Positive scalar {`0.1* abs(t0-tf)`}	Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep`: • When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. (See "Error Control Properties" on page 1-2026 for a description of the error tolerance properties.) • To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some DDE problems, the times of specific events are important. While solving a problem, the solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property.

DDE Events Property

String	Value	Description

`Events`	Function handle	A function handle that includes one or more event functions. For `dde23` and `ddesd`, this function has the following syntax:
		`[value,isterminal,direction] = events(t,y,YDEL)`
		For `ddensd`, the syntax is:
		`[value,isterminal,direction] = events(t,y,YDEL,YPDEL)`
		The output arguments, `value`, `isterminal`, and `direction`, are vectors for which the `i`th element corresponds to the `i`th event function:
		• `value(i)` is the value of the `i`th event function.
		• `isterminal(i) = 1` if you want the integration to terminate at a zero of this event function, and `0` otherwise.
		• `direction(i) = 0` if you want the solver to locate all zeros (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.
		If you specify an events function and events are detected, the solver returns three additional fields in the solution structure `sol`:
		• `sol.xe` is a row vector of times at which events occur.
		• `sol.ye` is a matrix whose columns are the solution values corresponding to times in `sol.xe`.
		• `sol.ie` is a vector containing indices that specify which event occurred at the corresponding time in `sol.xe`.

String	Value	Description
		For examples that use an event function while solving ordinary differential equation problems, see “Event Location” (`ballode`) and “Advanced Event Location” (`orbitode`), in the MATLAB Mathematics documentation.

Discontinuity Properties

The solver functions can solve problems with discontinuities in the history or in the coefficients of the equations. The following properties enable you to provide these solvers with a different initial value, and, for `dde23`, locations of known discontinuities. For more information, see “Discontinuities in DDEs”.

The following table describes the discontinuity properties.

DDE Discontinuity Properties

String	Value	Description
`Jumps`	Vector	Location of discontinuities. Points t where the history or solution may have a jump discontinuity in a low-order derivative. This applies only to the `dde23` solver.
`InitialY`	Vector	Initial value of solution. By default the initial value of the solution is the value returned by `history` at the initial point. Supply a different initial value as the value of the `InitialY` property.

Examples

To create an options structure that changes the relative error tolerance of the solver from the default value of `1e-3` to `1e-4`, enter

```
options = ddeset('RelTol',1e-4);
```

To recover the value of `'RelTol'` from `options`, enter

```
ddeget(options,'RelTol')

ans =
```

```
1.0000e-004
```

See Also

dde23 | ddensd | ddesd | ddeget | function_handle

Introduced before R2006a

deal

Distribute inputs to outputs

Compatibility

Beginning with MATLAB Version 7.0 software, you can access the contents of cell arrays and structure fields without using the `deal` function. See Example 3, below.

Syntax

```
[Y1, Y2, Y3, ...] = deal(X)
[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)
[S.field] = deal(X)
[X{:}] = deal(A.field)
[Y1, Y2, Y3, ...] = deal(X{:})
[Y1, Y2, Y3, ...] = deal(S.field)
```

Description

`[Y1, Y2, Y3, ...] = deal(X)` copies the single input to all the requested outputs. It is the same as `Y1 = X`, `Y2 = X`, `Y3 = X`, `...`

`[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)` is the same as `Y1 = X1`; `Y2 = X2`; `Y3 = X3`; `...`

Examples

Example 1 — Assign Data From a Cell Array

Use `deal` to copy the contents of a 4-element cell array into four separate output variables.

```
C = {rand(3) ones(3,1) eye(3) zeros(3,1)};
[a,b,c,d] = deal(C{:})
```

```
a =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

b =
    1
    1
    1

c =
    1    0    0
    0    1    0
    0    0    1

d =
    0
    0
    0
```

Example 2 — Assign Data From Structure Fields

Use `deal` to obtain the contents of all the name fields in a structure array:

```
A.name = 'Pat'; A.number = 176554;
A(2).name = 'Tony'; A(2).number = 901325;
[name1,name2] = deal(A(:).name)

name1 =
   Pat

name2 =
   Tony
```

Example 3 — Doing the Same Without deal

Beginning with MATLAB Version 7.0 software, you can, in most cases, access the contents of cell arrays and structure fields without using the `deal` function. The two commands shown below perform the same operation as those used in the previous two examples, except that these commands do not require `deal`.

```
[a,b,c,d] = C{:}
```

```
[name1,name2] = A(:).name
```

More About

Tips

`deal` is most useful when used with cell arrays and structures via comma-separated list expansion. Here are some useful constructions:

`[S.field] = deal(X)` sets all the fields with the name `field` in the structure array `S` to the value `X`. If `S` doesn't exist, use `[S(1:m).field] = deal(X)`.

`[X{:}] = deal(A.field)` copies the values of the field with name `field` to the cell array `X`. If `X` doesn't exist, use `[X{1:m}] = deal(A.field)`.

`[Y1, Y2, Y3, ...] = deal(X{:})` copies the contents of the cell array `X` to the separate variables `Y1, Y2, Y3, ...`

`[Y1, Y2, Y3, ...] = deal(S.field)` copies the contents of the fields with the name `field` to separate variables `Y1, Y2, Y3, ...`

See Also

`cell` | `iscell` | `celldisp` | `struct` | `isstruct` | `fieldnames` | `isfield` | `orderfields` | `rmfield` | `cell2struct` | `struct2cell`

Introduced before R2006a

deblank

Strip trailing blanks from end of string

Syntax

```
str = deblank(str)
c = deblank(c)
```

Description

`str = deblank(str)` removes all trailing whitespace and null characters from the end of character string `str`. A whitespace is any character for which the `isspace` function returns logical 1 (`true`).

`c = deblank(c)` when `c` is a cell array of strings, applies `deblank` to each element of `c`.

The `deblank` function is useful for cleaning up the rows of a character array.

Examples

Example 1 – Removing Trailing Blanks From a String

Compose a string `str` that contains space, tab, and null characters:

```
NL = char(0);    TAB = char(9);
str = [NL 32 TAB NL 'AB' 32 NL 'CD' NL 32 TAB NL 32];
```

Display all characters of the string between | symbols:

```
['|' str '|']
ans =
    |    AB  CD       |
```

Remove trailing whitespace and null characters, and redisplay the string:

```
newstr = deblank(str);
```

```
['|' newstr '|']
ans =
    |    AB  CD|
```

Example 2– Removing Trailing Blanks From a Cell Array of Strings

Create a 2-by-2 cell array in which each cell contains a word with trailing blanks:

```
A{1,1} = 'MATLAB    ';
A{1,2} = 'SIMULINK    ';
A{2,1} = 'Toolboxes    ';
A{2,2} = 'MathWorks    '
A =
    'MATLAB    '       'SIMULINK    '
    'Toolboxes    '    'MathWorks    '
```

Remove the trailing blanks and redisplay the cell array:

```
A = deblank(A);

A
A =
    'MATLAB'       'SIMULINK'
    'Toolboxes'    'MathWorks'
```

See Also

`strjust` | `strtrim`

Introduced before R2006a

dec2base

Convert decimal to base N number in string

Syntax

```
str = dec2base(d, base)
str = dec2base(d, base, n)
```

Description

`str = dec2base(d, base)` converts the nonnegative integer `d` to the specified `base`. `d` must be a nonnegative integer smaller than 2^52, and `base` must be an integer between 2 and 36. The returned argument `str` is a string.

`str = dec2base(d, base, n)` produces a representation with at least `n` digits.

Examples

The expression `dec2base(23, 2)` converts 23_{10} to base 2, returning the string `'10111'`.

See Also

`base2dec`

Introduced before R2006a

dec2bin

Convert decimal to binary number in string

Syntax

```
str = dec2bin(d)
str = dec2bin(d,n)
```

Description

`str = dec2bin(d)` returns the binary representation of `d` as a string. `d` must be a nonnegative integer. If `d` is greater than 2^52, `dec2bin` might not return an exact representation of `d`.

`str = dec2bin(d,n)` produces a binary representation with at least `n` bits.

The output of `dec2bin` is independent of the endian settings of the computer you are using.

Examples

Decimal 23 converts to binary 010111:

```
dec2bin(23)
ans =
    10111
```

See Also

`bin2dec` | `dec2hex`

dec2hex

Convert decimal to hexadecimal number in string

Syntax

```
str = dec2hex(d)
str = dec2hex(d, n)
```

Description

`str = dec2hex(d)` converts the decimal integer `d` to its hexadecimal representation stored in a MATLAB string. `d` must be a nonnegative integer. If `d` is an integer greater than 2^52, `dec2hex` might not return an exact representation. MATLAB converts noninteger inputs, such as those of class `double` or `char`, to their integer equivalents before converting to hexadecimal.

`str = dec2hex(d, n)` produces a hexadecimal representation with at least `n` digits.

Examples

To convert decimal 1023 to hexadecimal,

```
dec2hex(1023)
ans =
    3FF

dec2hex(1023, 6)
ans =
0003FF
```

Convert 2-by-5 array `A` to hexadecimal:

```
A = [3487,    125,  8997,  1433,  189; ...
      771,  84832,   118,  9366,  212];

 A(:)                    dec2hex(A)
ans =                    ans =
```

```
  3487                  00D9F
   771                  00303
   125                  0007D
 84832                  14B60
  8997                  02325
   118                  00076
  1433                  00599
  9366                  02496
   189                  000BD
   212                  000D4
```

See Also

`dec2bin` | `format` | `hex2dec` | `hex2num`

decic

Compute consistent initial conditions for ode15i

Syntax

```
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options)
[y0mod,yp0mod,resnrm] =
decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0...)
```

Description

`[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)` uses the inputs `y0` and `yp0` as initial guesses for an iteration to find output values that satisfy the requirement $f(t0, y0\text{mod}, yp0\text{mod}) = 0$, i.e., `y0mod` and `yp0mod` are consistent initial conditions. `odefun` is a function handle. The function `decic` changes as few components of the guesses as possible. You can specify that `decic` holds certain components fixed by setting `fixed_y0(i) = 1` if no change is permitted in the guess for `y0(i)` and 0 otherwise. `decic` interprets `fixed_y0 = []` as allowing changes in all entries. `fixed_yp0` is handled similarly.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, if necessary.

You cannot fix more than `length(y0)` components. Depending on the problem, it may not be possible to fix this many. It also may not be possible to fix certain components of `y0` or `yp0`. It is recommended that you fix no more components than necessary.

`[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options)` computes as above with default tolerances for consistent initial conditions, `AbsTol` and `RelTol`, replaced by the values in options, a structure you create with the `odeset` function.

`[y0mod,yp0mod,resnrm] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0...)` returns the norm of

`odefun(t0,y0mod,yp0mod)` as `resnrm`. If the norm seems unduly large, use `options` to decrease `RelTol` (`1e-3` by default).

Examples

The files, `ihb1dae.m` and `iburgersode.m`, provide examples which use `decic` to solve implicit ODEs.

See Also

`ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

deconv

Deconvolution and polynomial division

Syntax

```
[q,r] = deconv(v,u)
```

Description

`[q,r] = deconv(v,u)` deconvolves vector `u` out of vector `v`, using long division. The quotient is returned in vector `q` and the remainder in vector `r` such that `v = conv(u,q)+r`.

If `u` and `v` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division. The result of dividing `v` by `u` is quotient `q` and remainder `r`.

Examples

If

```
u = [1   2   3   4]
v = [10   20   30]
```

the convolution is

```
c = conv(u,v)
c =
    10     40     100     160     170     120
```

Use deconvolution to recover `v`:

```
[q,r] = deconv(c,u)
q =
    10    20    30
r =
    0     0     0     0     0     0
```

This gives a quotient equal to `v` and a zero remainder.

More About

Algorithms

`deconv` uses the `filter` primitive.

See Also

`conv` | `residue`

Introduced before R2006a

del2

Discrete Laplacian

Syntax

```
L = del2(U)
L = del2(U,h)
L = del2(U,h1,...,hN)
```

Description

`L = del2(U)` returns a discrete approximation of Laplace's differential operator applied to `U` using the default spacing, `h = 1`, between all points.

`L = del2(U,h)` specifies a uniform, scalar spacing, `h`, between points in all dimensions of `U`.

`L = del2(U,h1,...,hN)` specifies the spacing, `h1,...,hN`, between points in each corresponding dimension of `U`. For each dimension, specify the spacing as a scalar or a vector of coordinates. The number of spacing inputs must equal the number of dimensions in `U`.

Examples

Second Derivative of Vector

Calculate the acceleration of an object from a vector of position data.

Create a vector of position data.

```
p = [1 3 6 10 16 18 29];
```

To find the acceleration of the object, use `del2` to calculate the second numerical derivative of `p`. Use the default spacing `h = 1` between data points.

```
L = 4*del2(p)

L =

     1     1     1     2    -4     9    22
```

Each value of `L` is an approximation of the instantaneous acceleration at that point.

Second Derivative of Cosine Vector

Calculate the discrete 1-D Laplacian of a cosine vector.

Define the domain of the function.

```
x = linspace(-2*pi,2*pi);
```

This produces 100 evenly spaced points in the range $-2\pi \le x \le 2\pi$.

Create a vector of cosine values in this domain.

```
U = cos(x);
```

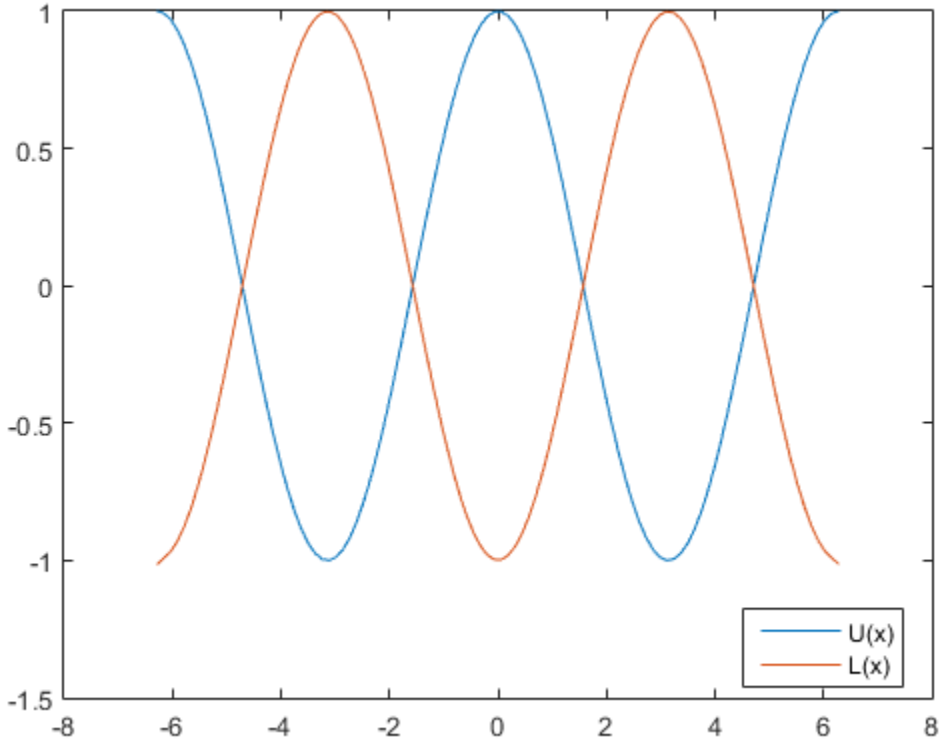
Calculate the Laplacian of `U` using `del2`. Use the domain vector `x` to define the 1-D coordinate of each point in `U`.

```
L = 4*del2(U,x);
```

Analytically, the Laplacian of this function is equal to $\Delta U = -\cos(x)$.

Plot the results.

```
plot(x,U,x,L)
legend('U(x)','L(x)','Location','Best')
```



The graph of U and L agrees with the analytic result for the Laplacian.

Laplacian of Multivariate Function

Calculate and plot the discrete Laplacian of a multivariate function.

Define the x and y domain of the function.

```
[x,y] = meshgrid(-5:0.25:5,-5:0.25:5);
```

Define the function $U(x,y) = \frac{1}{3}\left(x^4 + y^4\right)$ over this domain.

```
U = 1/3.*(x.^4+y.^4);
```

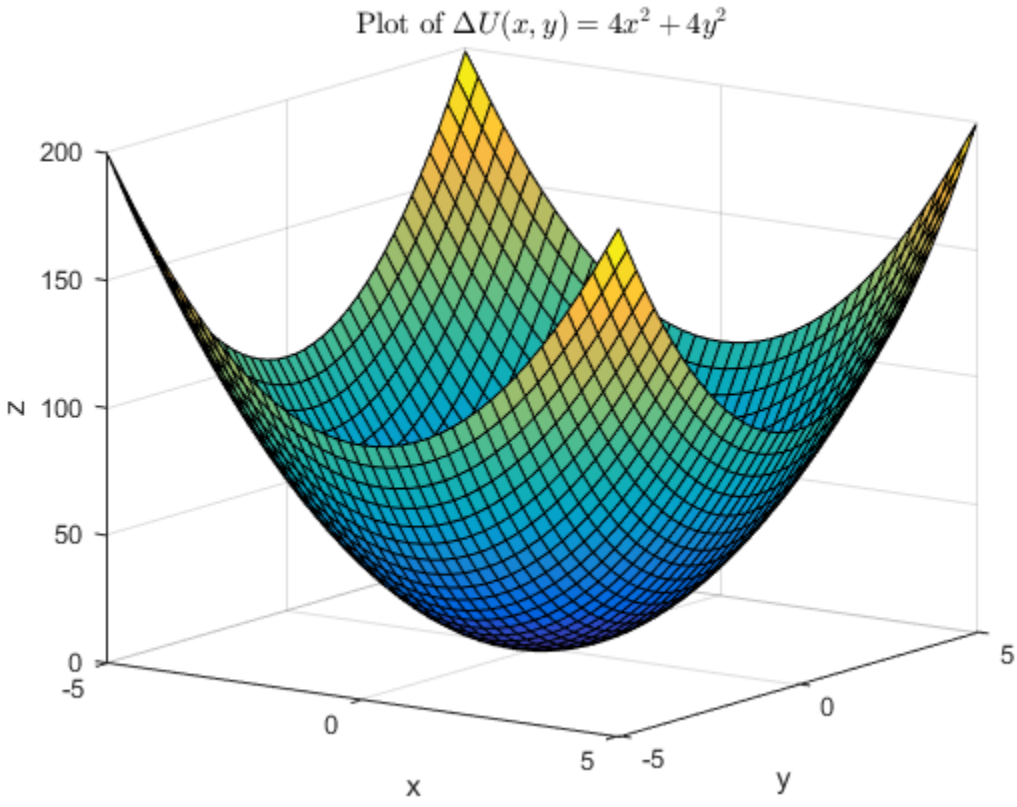
Calculate the Laplacian of this function using `del2`. The spacing between the points in `U` is equal in all directions, so you can specify a single spacing input, `h`.

```
h = 0.25;
L = 4*del2(U,h);
```

Analytically, the Laplacian of this function is equal to $\Delta U(x,y) = 4x^2 + 4y^2$.

Plot the discrete Laplacian, `L`.

```
figure
surf(x,y,L)
grid on
title('Plot of $\Delta U(x,y) = 4x^2+4y^2$','Interpreter','latex')
xlabel('x')
ylabel('y')
zlabel('z')
view(35,14)
```



The graph of `L` agrees with the analytic result for the Laplacian.

Laplacian of Natural Logarithm Function

Calculate the discrete Laplacian of a natural logarithm function.

Define the x and y domain of the function on a grid of real numbers.

```
[x,y] = meshgrid(-5:5,-5:0.5:5);
```

Define the function $U(x,y) = \frac{1}{2}\log\left(x^2 y\right)$ over this domain.

```
U = 0.5*log(x.^2.*y);
```

The logarithm is complex-valued when the argument `y` is negative.

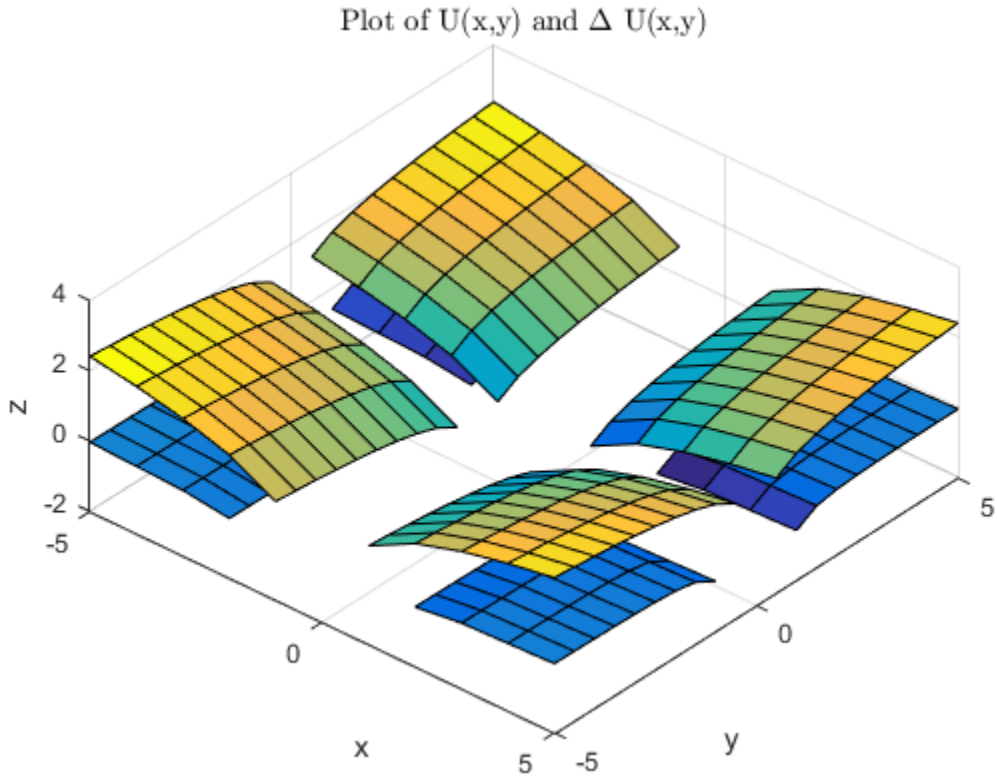
Use `del2` to calculate the discrete Laplacian of this function. Specify the spacing between grid points in each direction.

```
hx = 1;
hy = 0.5;
L = 4*del2(U,hx,hy);
```

Analytically, the Laplacian is equal to $\Delta U(x,y) = -\left(1/x^2 + 1/2y^2\right)$. This function is not defined on the lines $x = 0$ or $y = 0$.

Plot the real parts of `U` and `L` on the same graph.

```
figure
surf(x,y,real(L))
hold on
surf(x,y,real(U))
grid on
title('Plot of U(x,y) and $\Delta$ U(x,y)','Interpreter','latex')
xlabel('x')
ylabel('y')
zlabel('z')
view(41,58)
```



The top surface is `U` and the bottom surface is `L`.

Input Arguments

`U` — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

h — Spacing in all dimensions

1 (default) | scalar

Spacing in all dimensions, specified as 1 (default), or a scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

h1,...,hN — Spacing in each dimension

scalars | vectors

Spacing in each dimension, specified as scalars or vectors. The number of spacing inputs must be equal to the number of dimensions in U. Each spacing input defines the spacing between points in the corresponding dimension of U:

- Use a scalar to specify a uniform spacing.
- Use a vector to specify a nonuniform spacing. The coordinate vector gives the position of each point and must have the same number of elements as the corresponding dimension of U (a one-to-one match of coordinates and points).

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

L — Discrete Laplacian approximation

vector | matrix | multidimensional array

Discrete Laplacian approximation, returned as a vector, matrix, or multidimensional array. L is the same size as the input, U.

More About

Laplace's differential operator

If a matrix U is a function *U(x,y)* that is evaluated at the points of a square grid, then `4*del2(U)` is a finite difference approximation of Laplace's differential operator applied to *U*,

$$L = \frac{\Delta U}{4} = \frac{1}{4}\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \right).$$

For functions of more variables, *U(x,y,z,...)*, the discrete Laplacian `del2(U)` calculates second-derivatives in each dimension,

$$L = \frac{\Delta U}{2N} = \frac{1}{2N}\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} + \ldots \right),$$

where N is the number of dimensions in U and $N \geq 2$.

Algorithms

If the input `U` is a matrix, the interior points of `L` are found by taking the difference between a point in `U` and the average of its four neighbors:

$$L_{ij} = \left[\frac{\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \right)}{4} - u_{i,j} \right].$$

Then, `del2` calculates the values on the edges of `L` by linearly extrapolating the second differences from the interior. This formula is extended for multidimensional `U`.

See Also

`diff` | `gradient`

Introduced before R2006a

DelaunayTri class

Superclasses: TriRep

(Will be removed) Delaunay triangulation in 2-D and 3-D

Note: `DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Description

`DelaunayTri` creates a Delaunay triangulation object from a set of points. You can incrementally modify the triangulation by adding or removing points. In 2-D triangulations you can impose edge constraints. You can perform topological and geometric queries, and compute the Voronoi diagram and convex hull.

Definitions

The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

Construction

.DelaunayTri	(Will be removed) Construct Delaunay triangulation

Methods

convexHull	(Will be removed) Convex hull

inOutStatus	(Will be removed) Status of triangles in 2-D constrained Delaunay triangulation
nearestNeighbor	(Will be removed) Point closest to specified location
pointLocation	(Will be removed) Simplex containing specified location
voronoiDiagram	(Will be removed) Voronoi diagram

Inherited methods

baryToCart	(Will be removed) Convert point coordinates from barycentric to Cartesian
cartToBary	(Will be removed) Convert point coordinates from cartesian to barycentric
circumcenters	(Will be removed) Circumcenters of specified simplices
edgeAttachments	(Will be removed) Simplices attached to specified edges
edges	(Will be removed) Triangulation edges
faceNormals	(Will be removed) Unit normals to specified triangles
featureEdges	(Will be removed) Sharp edges of surface triangulation

freeBoundary	(Will be removed) Facets referenced by only one simplex
incenters	(Will be removed) Incenters of specified simplices
isEdge	(Will be removed) Test if vertices are joined by edge
neighbors	(Will be removed) Simplex neighbor information
size	(Will be removed) Size of triangulation matrix
vertexAttachments	(Will be removed) Return simplices attached to specified vertices

Properties

`Constraints`	`Constraints` is a `numc`-by-2 matrix that defines the constrained edge data in the triangulation, where `numc` is the number of constrained edges. Each constrained edge is defined in terms of its endpoint indices into `X`. The constraints can be specified when the triangulation is constructed or can be imposed afterwards by directly editing the constraints data. This feature is only supported for 2-D triangulations.
`X`	The dimension of `X` is `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside. If column vectors of `x`,`y` or `x`,`y`,`z` coordinates are used to construct the triangulation, the data is consolidated into a single matrix `X`.

`Triangulation`	Triangulation is a matrix representing the set of simplices (triangles or tetrahedra etc.) that make up the triangulation. The matrix is of size `mtri`-by-`nv`, where `mtri` is the number of simplices and `nv` is the number of vertices per simplex. The triangulation is represented by standard simplex-vertex format; each row specifies a simplex defined by indices into `X`, where `X` is the array of point coordinates.

Instance Hierarchy

`DelaunayTri` is a subclass of `TriRep`.

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

`scatteredInterpolant` | `delaunayTriangulation` | `triangulation`

DelaunayTri

Class: DelaunayTri

(Will be removed) Construct Delaunay triangulation

Note: `DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
DT = DelaunayTri()
DT = DelaunayTri(X)
DT = DelaunayTri(x,y)
DT = DelaunayTri(x,y,z)
DT = DelaunayTri(..., C)
```

Description

`DT = DelaunayTri()` creates an empty Delaunay triangulation.

`DT = DelaunayTri(X)`, `DT = DelaunayTri(x,y)` and `DT = DelaunayTri(x,y,z)` create a Delaunay triangulation from a set of points. The points can be specified as an `mpts`-by-`ndim` matrix `X`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, where `ndim` is 2 or 3. Alternatively, the points can be specified as column vectors `(x,y)` or `(x,y,z)` for 2-D and 3-D input.

`DT = DelaunayTri(..., C)` creates a constrained Delaunay triangulation. The edge constraints `C` are defined by an `numc`-by-2 matrix, `numc` being the number of constrained edges. Each row of `C` defines a constrained edge in terms of its endpoint indices into the point set `X`. This feature is only supported for 2-D triangulations.

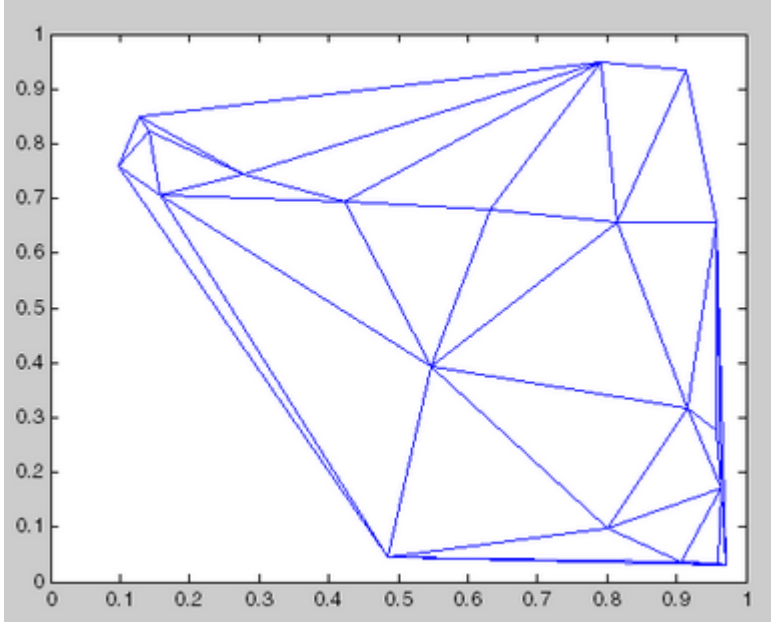
Definitions

The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

Examples

Compute the Delaunay triangulation of twenty random points located within a unit square.

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y)
triplot(dt);
```



For more examples, type `help demoDelaunayTri` at the MATLAB command-line prompt.

See Also

`scatteredInterpolant` | `delaunayTriangulation` | `triangulation`

delaunay

Delaunay triangulation

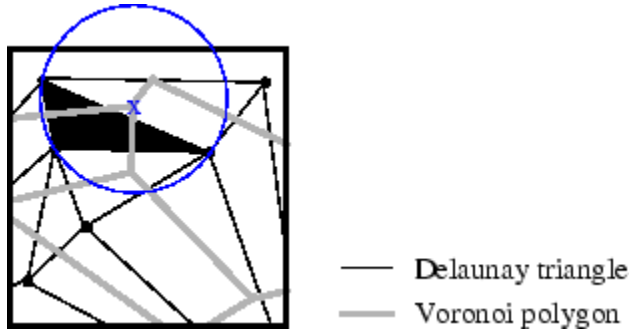
Note: Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `delaunay`.

Syntax

```
TRI = delaunay(X,Y)
TRI = delaunay(X,Y,Z)
TRI = delaunay(X)
```

Definitions

`delaunay` creates a Delaunay triangulation of a set of points in 2-D or 3-D space. A 2-D Delaunay triangulation ensures that the circumcircle associated with each triangle contains no other point in its interior. This definition extends naturally to higher dimensions.



Description

`TRI = delaunay(X,Y)` creates a 2-D Delaunay triangulation of the points (`X`,`Y`), where `X` and `Y` are column-vectors. `TRI` is a matrix representing the set of triangles that make up the triangulation. The matrix is of size `mtri`-by-3, where `mtri` is the number of triangles. Each row of `TRI` specifies a triangle defined by indices with respect to the points.

`TRI = delaunay(X,Y,Z)` creates a 3-D Delaunay triangulation of the points (`X`,`Y`,`Z`), where `X`, `Y`, and `Z` are column-vectors. `TRI` is a matrix representing the set of tetrahedra that make up the triangulation. The matrix is of size `mtri`-by-4, where `mtri` is the number of tetrahedra. Each row of `TRI` specifies a tetrahedron defined by indices with respect to the points.

`TRI = delaunay(X)` creates a 2-D or 3-D Delaunay triangulation from the point coordinates `X`. This variant supports the definition of points in matrix format. `X` is of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, $2 \leqq$ `ndim` $\leqq 3$. The output triangulation is equivalent to that of the dedicated functions supporting the 2-input or 3-input calling syntax.

`delaunay` produces an isolated triangulation, useful for applications like plotting surfaces via the `trisurf` function. If you wish to query the triangulation; for example, to perform nearest neighbor, point location, or topology queries, use `delaunayTriangulation` instead.

Visualization

Use one of these functions to plot the output of `delaunay`:

`triplot`	Displays the triangles defined in the `m`-by-3 matrix `TRI`.
`trisurf`	Displays each triangle defined in the m-by-3 matrix TRI as a surface in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example `trisurf(TRI,x,y,zeros(size(x)))`
`trimesh`	Displays each triangle defined in the m-by-3 matrix TRI as a mesh in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example,

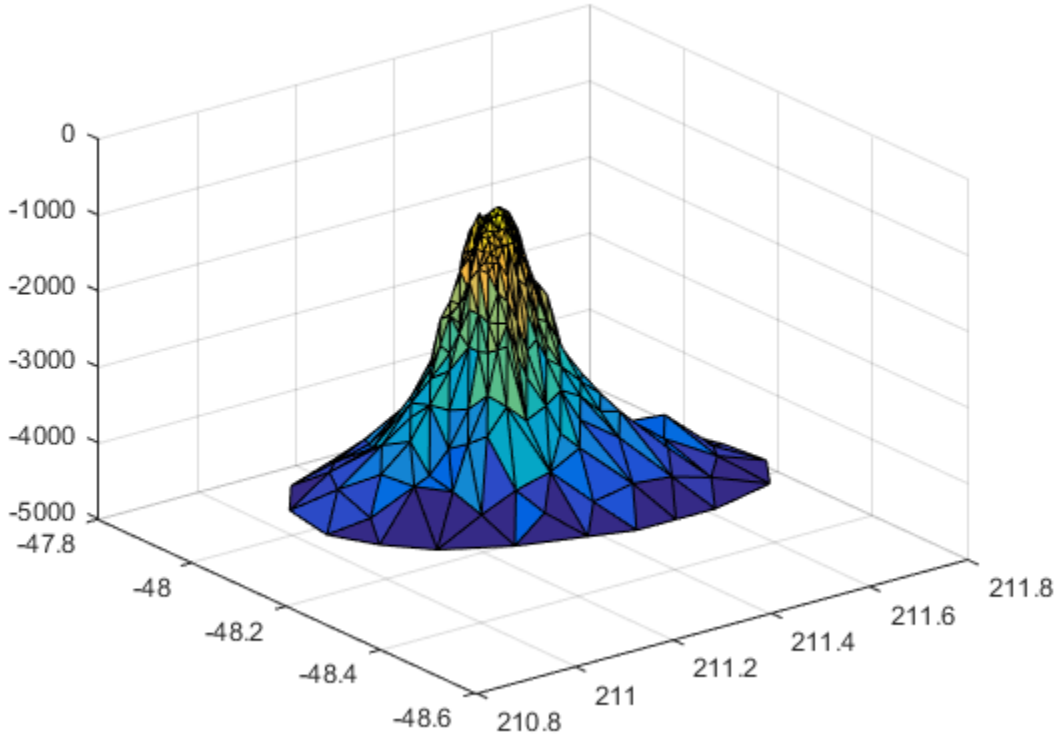
	`trimesh(TRI,x,y,zeros(size(x)))` produces almost the same result as `triplot`, except in 3-D space.
`tetramesh`	Plots a triangulation composed of tetrahedra.

Examples

Plot Delaunay Triangulation

Plot the Delaunay triangulation of a large dataset.

```
load seamount
tri = delaunay(x,y);
trisurf(tri,x,y,z);
```



See Also

`scatteredInterpolant` | `plot` | `triplot` | `delaunayTriangulation` | `trimesh` | `trisurf`

Introduced before R2006a

delaunayn

N-D Delaunay triangulation

Syntax

```
T = delaunayn(X)
T = delaunayn(X, options)
```

Description

`T = delaunayn(X)` computes a set of simplices such that no data points of `X` are contained in any circumspheres of the simplices. The set of simplices forms the Delaunay triangulation. `X` is an `m`-by-`n` array representing m points in n-dimensional space. `T` is a `numt`-by-(`n`+1) array where each row contains the indices into `X` of the vertices of the corresponding simplex.

`T = delaunayn(X, options)` specifies a cell array of strings `options`. The default options are:

- `{'Qt','Qbb','Qc'}` for 2- and 3-dimensional input
- `{'Qt','Qbb','Qc','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options used. If `options` is `{''}`, no options are used, not even the default.

Visualization

Plotting the output of `delaunayn` depends of the value of `n`:

- For `n = 2`, use `triplot`, `trisurf`, or `trimesh` as you would for `delaunay`.
- For `n = 3`, use `tetramesh`.

  For more control over the color of the facets, use `patch` to plot the output.
- You cannot plot `delaunayn` output for `n > 3`.

Examples

3-D Delaunay Triangulation

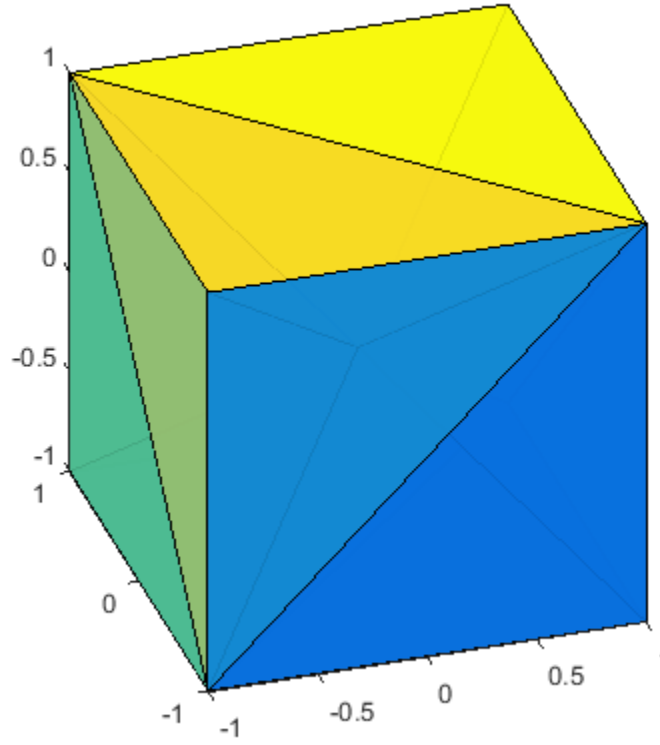
This example generates an n-dimensional Delaunay triangulation, where `n = 3`.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);  % A cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
X = [x(:) y(:) z(:)];
Tes = delaunayn(X)
```

```
Tes =

     4     3     9     1
     4     9     2     1
     7     9     3     1
     7     5     9     1
     7     9     4     3
     7     8     4     9
     6     2     9     1
     6     9     5     1
     6     4     9     2
     6     4     8     9
     6     9     7     5
     6     8     7     9
```

You can use `tetramesh` to visualize the tetrahedrons that form the corresponding simplex. `camorbit` rotates the camera position to provide a meaningful view of the figure.

```
tetramesh(Tes,X);
camorbit(20,0)
```



See Also

`delaunayTriangulation` | `convhulln` | `tetramesh` | `voronoin` | `camorbit`

Introduced before R2006a

delaunayTriangulation class

Superclasses: triangulation

Delaunay triangulation in 2-D and 3-D

Description

Use the `delaunayTriangulation` class to create a 2-D or 3-D triangulation from a set of points. When your points are in 2-D, you can specify edge constraints .

You can perform a variety of topological and geometric queries on a `delaunayTriangulation`, including any `triangulation` query. For example, locate a facet that contains a specific point, find the vertices of the convex hull, or compute the Voronoi Diagram.

Construction

`DT = delaunayTriangulation(P)` creates a Delaunay triangulation from the points in `P`. Matrix `P` has 2 or 3 columns, depending on whether your points are in 2-D or 3-D space.

`DT = delaunayTriangulation(P,C)` specifies the edge constraints in matrix `C`. In this case, `P` specifies points in 2-D. Each row of `C` defines the start and end vertex IDs of a constrained edge.

`DT = delaunayTriangulation(x,y)` creates a 2-D Delaunay triangulation from the point coordinates in the column vectors, `x` and `y`.

`DT = delaunayTriangulation(x,y,C)` specifies the edge constraints in matrix `C`.

`DT = delaunayTriangulation(x,y,z)` creates a 3-D Delaunay triangulation from the point coordinates in the column vectors, `x`, `y`, and `z`.

`DT = delaunayTriangulation()` creates an empty Delaunay triangulation.

Input Arguments

`P`

Input points, specified as a matrix whose columns are the *x*, *y*, (and possibly *z*) coordinates of the triangulation points. The row numbers of `P` are the vertex IDs in the triangulation.

`x`

x-coordinates vector, specified as a column vector containing the *x*-coordinates of the triangulation points.

`y`

y-coordinates vector, specified as a column vector containing the *y*-coordinates of the triangulation points.

`z`

z-coordinates vector, specified as a column vector containing the *z*-coordinates of the triangulation points.

`C`

Vertex IDs of constrained edges, specified as a 2-column matrix. Each row of `C` corresponds to a constrained edge and contains two IDs:

- `C(j,1)` is the ID of the vertex at the start of an edge.
- `C(j,2)` is the ID of the vertex at end of the edge.

You can specify edge constraints for 2-D triangulations only.

Properties

`Points`

Points in the triangulation, represented as a matrix containing the following information:

- Each row in `DT.Points` contains the coordinates of a vertex.
- Each row number of `DT.Points` is a vertex ID.

`ConnectivityList`

Triangulation connectivity list, represented as a matrix. This matrix contains the following information:

- Each row represents a triangle or tetrahedron in the triangulation.
- Each row number of `DT.ConnectivityList` is a "Triangle or Tetrahedron ID" on page 1-2075.
- Each element is a vertex ID.

`Constraints`

Constrained edges, represented as a two-column matrix of vertex IDs. Each row of `DT.Constraints` corresponds to a constrained edge and contains two IDs:

- `DT.Constraints(j,1)` is the ID of the vertex at the start of an edge.
- `DT.Constraints(j,2)` is the ID of the vertex at end of the edge.

`DT.Constraints` is an empty matrix when the triangulation has no constrained edges.

Methods

convexHull	Convex hull
isInterior	Test if triangle is in interior of 2-D constrained Delaunay triangulation
voronoiDiagram	Voronoi diagram

Inherited Methods

barycentricToCartesian	Converts point coordinates from barycentric to Cartesian
cartesianToBarycentric	Converts point coordinates from Cartesian to barycentric

circumcenter	Circumcenter of triangle or tetrahedron
edgeAttachments	Triangles or tetrahedra attached to specified edge
edges	Triangulation edges
faceNormal	Triangulation face normal
featureEdges	Triangulation sharp edges
freeBoundary	Triangulation facets referenced by only one triangle or tetrahedron
incenter	Incenter of triangle or tetrahedron
isConnected	Test if two vertices are connected by edge
nearestNeighbor	Vertex closest to specified location
neighbors	Neighbors to specified triangle or tetrahedron
pointLocation	Triangle or tetrahedron containing specified point
size	Size of triangulation connectivity list
vertexAttachments	Triangles or tetrahedra attached to specified vertex
vertexNormal	Triangulation vertex normal

Definitions

Delaunay Triangulation

In a 2-D Delaunay triangulation, the circumcircle associated with each triangle does not contain any points in its interior. Similarly, a 3-D Delaunay triangulation does not have any points in the interior of the circumsphere associated with each tetrahedron. This definition extends to N-D, although `delaunayTriangulation` supports only 2-D and 3-D.

Vertex ID

A row number of the matrix, `DT.Points`. Use this ID to refer a specific vertex in the triangulation.

Triangle or Tetrahedron ID

A row number of the matrix, `DT.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

2-D Delaunay Triangulation

Create a 2-D `delaunayTriangulation` for 30 random points.

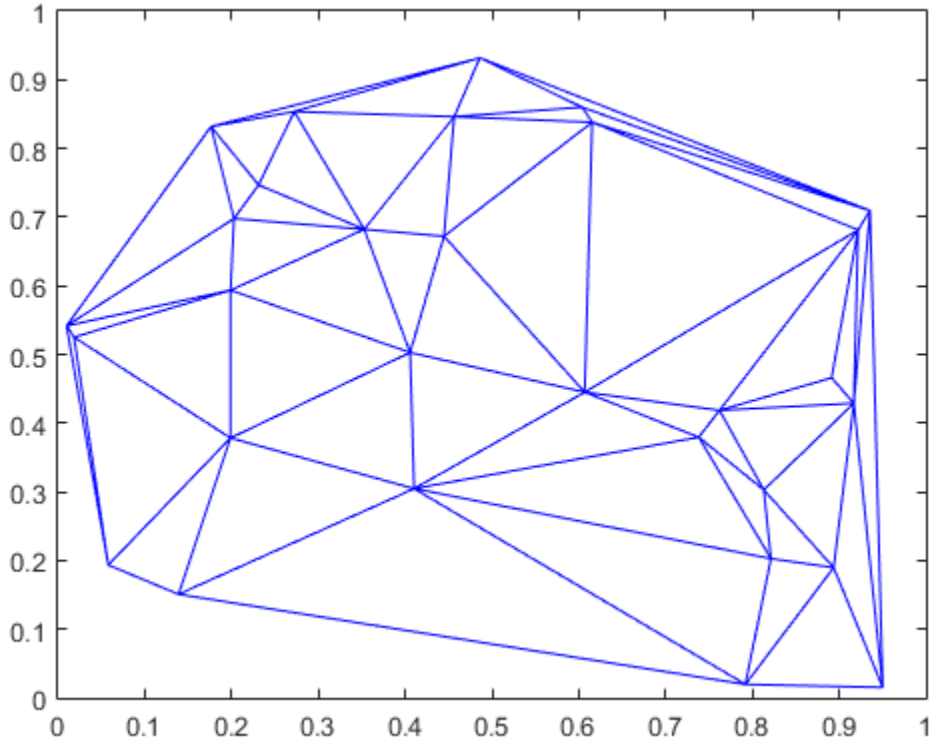
```
P = gallery('uniformdata',[30 2],0);
DT = delaunayTriangulation(P)
```

```
DT =
```

```
  delaunayTriangulation with properties:

             Points: [30x2 double]
  ConnectivityList: [50x3 double]
       Constraints: []
```

Plot the triangulation.

```
figure
triplot(DT)
```



3-D Delaunay Triangulation

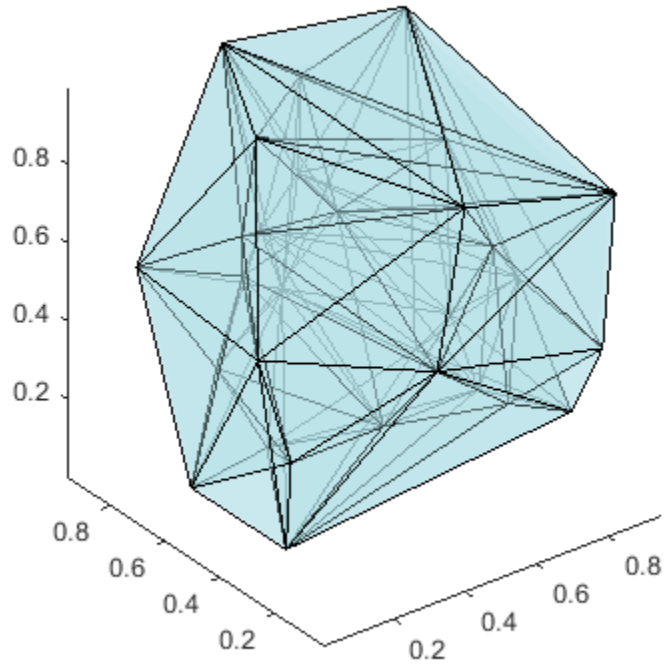
Create a 3-D `delaunayTriangulation` for 30 random points.

```
x = gallery('uniformdata',[30 1],0);
y = gallery('uniformdata',[30 1],1);
z = gallery('uniformdata',[30 1],2);
DT = delaunayTriangulation(x,y,z)
```

```
DT =

  delaunayTriangulation with properties:

             Points: [30x3 double]
    ConnectivityList: [111x4 double]
         Constraints: []
```

Plot the triangulation at 30% opacity with a light blue face color.

```
faceColor  = [0.6875 0.8750 0.8984];
figure
tetramesh(DT,'FaceColor',faceColor,'FaceAlpha',0.3);
```



See Also

`delaunay` | `delaunayn` | `triangulation`

convexHull

Class: delaunayTriangulation

Convex hull

Syntax

```
K = convexHull(DT)
[K,v] = convexHull(DT)
```

Description

`K = convexHull(DT)` returns the vertices of the convex hull.

`[K,v] = convexHull(DT)` also returns the area or volume bounded by the convex hull.

Input Arguments

DT

A Delaunay triangulation, see `delaunayTriangulation`.

Output Arguments

K

Convex hull vertices, returned as a matrix of vertex IDs. The shape of `K` depends on whether your triangulation is 2-D or 3-D:

- When `DT` is 2-D, `K` is a column vector containing the sequence of vertex IDs around the convex hull.
- When `DT` is 3-D, `K` is a triangulation connectivity list containing the triangles on the convex hull.

v

Area or volume bounded by the convex hull, returned as a scalar value.

Definitions

Vertex ID

A row number of the matrix, `DT.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Convex Hull in 2-D Space

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[10,1],0);
y = gallery('uniformdata',[10,1],1);
DT = delaunayTriangulation(x,y);
```

Calculate the convex hull.

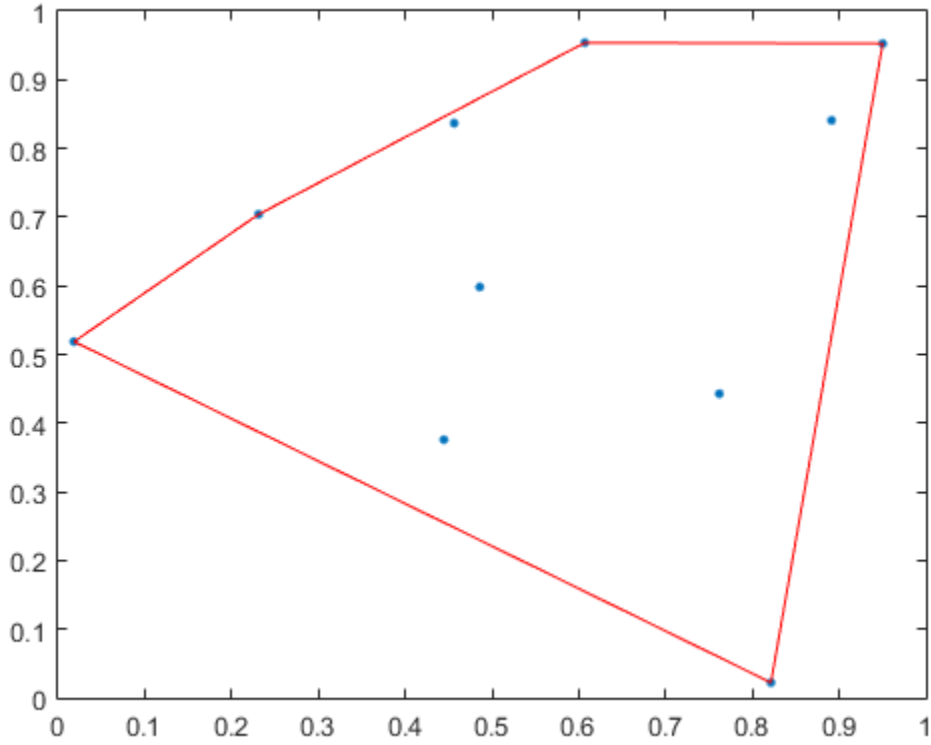
```
k = convexHull(DT)
```

```
k =

     1
     3
     2
     8
     9
     1
```

Plot the points and highlight the convex hull in red.

```
figure
```

```
plot(DT.Points(:,1),DT.Points(:,2), '.','markersize',10);
hold on
plot(DT.Points(k,1),DT.Points(k,2),'r')
hold off
```



Convex Hull in 3-D Space

Use `convexHull` to calculate the convex hull of a set of random points within a unit cube.

Create a Delaunay triangulation from a set of random points.

```
P = gallery('uniformdata',[25,3],1);
```

```
DT = delaunayTriangulation(P);
```

Calculate the convex hull and the volume bounded by the convex hull.

```
[K,v] = convexHull(DT);
```

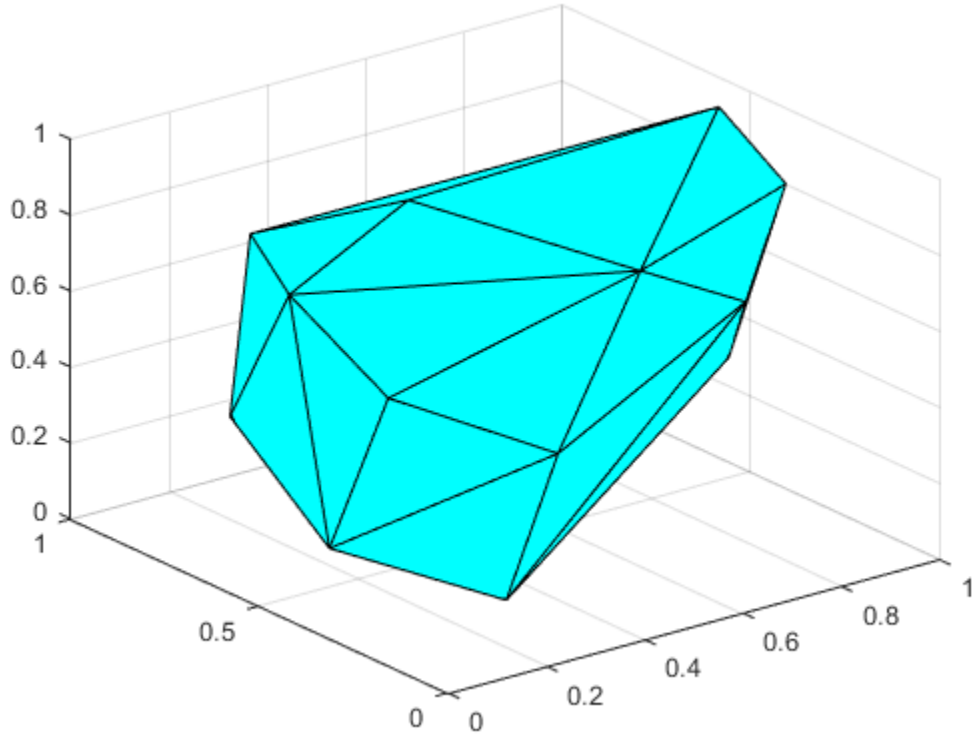
Examine the volume.

```
v
```

```
v =

    0.3561
```

Plot the convex hull.

```
trisurf(K,DT.Points(:,1),DT.Points(:,2),DT.Points(:,3),...
       'FaceColor','cyan')
```



See Also

`convhull` | `convhulln` | `triangulation` | `voronoiDiagram`

isInterior

Class: delaunayTriangulation

Test if triangle is in interior of 2-D constrained Delaunay triangulation

Syntax

```
tf = isInterior(DT)
```

Description

`tf = isInterior(DT)` returns an array of logical values that indicate whether the triangles in a constrained Delaunay triangulation are inside the bounded geometric domain. A triangle, `DT.ConnectivityList(j,:)`, is classified as inside the domain when `tf(j)` is `true`. Otherwise, the triangle is outside the domain.

Input Arguments

`DT`

A 2-D `delaunayTriangulation` that has a set of constrained edges that define a bounded geometric domain.

Output Arguments

`tf`

Logical values, returned as a column vector. Element `tf(j)` is `true` when the triangle whose ID is `j` is inside the domain of `DT`.

Definitions

Triangle ID

A row number of the matrix, `DT.ConnectivityList`. You use this ID to refer a specific triangle.

Examples

Find and Plot Triangles within a Boundary

Create a geometric domain whose shape is a square frame.

```
outerprofile = [-5 -5; -3 -5; -1 -5;  1 -5;
                 3 -5;  5 -5;  5 -3;  5 -1;
                 5  1;  5  3;  5  5;  3  5;
                 1  5; -1  5; -3  5; -5  5;
                -5  3; -5  1; -5 -1; -5 -3];

innerprofile = outerprofile.*0.5;
profile = [outerprofile; innerprofile];
```

Define the edge constraints.

```
outercons = [(1:19)' (2:20)'; 20 1;];
innercons = [(21:39)' (22:40)'; 40 21];
C = [outercons; innercons];
```

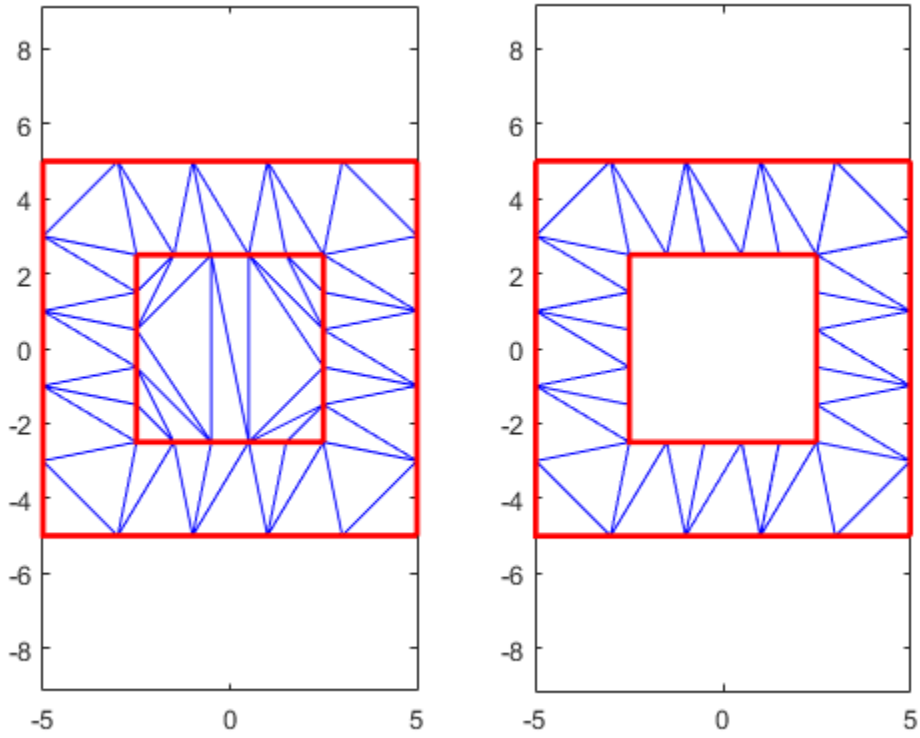
Create the constrained Delaunay triangulation.

```
DT = delaunayTriangulation(profile,C);
```

Plot the triangulation.

```
figure
subplot(1,2,1)
triplot(DT)

% Highlight the inner square in red.
hold on
plot(DT.Points(innercons',1),DT.Points(innercons',2),...
     '-r','LineWidth',2)
```

```
% Highlight the outer square in red and resize the |x| and |y| axes to make
% the plot square.
plot(DT.Points(outercons',1),DT.Points(outercons',2), ...
     '-r','LineWidth', 2)
axis equal

% Plot only the triangles that lie inside of the domain.
hold off
subplot(1,2,2)
inside = isInterior(DT);
triplot(DT.ConnectivityList(inside, :),DT.Points(:,1),DT.Points(:,2))

% Highlight the inner and outer squares in red.
hold on
plot(DT.Points(outercons',1),DT.Points(outercons',2), ...
     '-r','LineWidth', 2)
plot(DT.Points(innercons',1),DT.Points(innercons',2), ...
     '-r','LineWidth', 2)
axis equal
hold off
```



See Also

`triangulation`

voronoiDiagram

Class: delaunayTriangulation

Voronoi diagram

Syntax

```
[V,R] = voronoiDiagram(DT)
```

Description

`[V,R] = voronoiDiagram(DT)` returns the Voronoi vertices, `V`, and the Voronoi regions, `R`, of the points, `DT.Points`.

The *Voronoi diagram* of a set of points, such as `DT.Points`, decomposes the space around each point, `DT.Points(j,:)`, into a region of influence, `R{j}`. Locations within the region, `R{j}`, are closer to point `j` than any other point in `DT.Points`. The region of influence is called the *Voronoi region*. The collection of all the Voronoi regions is the Voronoi diagram.

The Voronoi regions associated with points that lie on the convex hull of `DT.Points` are unbounded. Bounding edges of these regions radiate to infinity. The vertex at infinity is represented by the first vertex in `V`.

Input Arguments

DT

A Delaunay triangulation, see `delaunayTriangulation`.

Output Arguments

V

Voronoi vertices, returned as a matrix. Each row of `V` contains the coordinates of a Voronoi vertex.

R

Voronoi regions, returned as a vector cell array the same length as `DT.Points`. The elements of `R` are row numbers of `V`. The coordinates of the Voronoi vertices bounding a region are `V(R{j},:)`. The Voronoi region associated with the point `DT.Points(j)` is `R{j}`.

Examples

Compute the Voronoi Diagram of a 2-D Triangulation

Create a Delaunay triangulation from a set of points.

```
P = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ];
DT = delaunayTriangulation(P);
```

Calculate the Voronoi vertices and regions.

```
[V,R] = voronoiDiagram(DT);
```

Examine the connectivity of the Voronoi region associated with the third point in the triangulation.

```
R{3}
```

```
ans =

     1    10     7     4
```

Examine the coordinates of the Voronoi vertices bounding the region.

```
V(R{3},:)
```

```
ans =

       Inf       Inf
    0.7000   -1.6500
```

```
   -0.0500   -0.5250
   -1.7500    0.7500
```

The `Inf` values indicate that the region contains points on the convex hull.

See Also

`convexHull` | `triangulation` | `voronoi` | `voronoin`

delete

Delete files or objects

Syntax

```
delete('fileName1', 'filename2', ...)
delete fileName
delete(h)
delete(handle_array)
```

Description

`delete('fileName1', 'filename2', ...)` deletes the files `fileName1`, `fileName2`, and so on, from the disk. `fileName` is a string and can be an absolute path or a path relative to the current folder. `fileName` also can include wildcards (`*`).

`delete fileName` is the command syntax. Delete multiple files by appending filenames, separated by spaces. When filenames contain space characters, you must use the function form.

`delete(h)` deletes the graphics object `h`. If `h` an array, the function deletes all objects in the array. `h` can also contain the figure `Number`.

The function deletes objects without requesting verification, even if when the objects are figure windows.

`delete(handle_array)` is a method of the `handle` class. It removes from memory the handle objects referenced by `handle_array`.

When deleted, any references to the objects in `handle_array` become invalid. To remove the handle variables, use the `clear` function.

As `delete` does not ask for confirmation, to avoid accidentally losing files or graphics objects, be sure to specify accurately the items to delete. To move files to a different location when running `delete`, use the **General** preference for **Deleting files**, or the `recycle` function.

The `delete` function deletes files and graphics objects only. To delete folders, use `rmdir`.

Examples

Delete all files with a `.mat` extension in the `/mytests/` folder:

```
delete('/mytests/*.mat')
```

Create a figure and an axes, and then delete the axes:

```
fig = figure;
ax = axes;
...
delete(ax)
```

The axes is deleted, but the figure remain. The axes variable `ax` remains in the workspace, but no longer refers to an object.

Use an array to delete all the line objects created by `plot`:

```
H = plot(magic(5));
```

Pass the array `H` to `delete`:

```
delete(H)
```

More About

- "Specify File Names"

See Also

`dir` | `rmdir` | `clear` | `recycle`

Introduced before R2006a

delete (COM)

Remove COM control or server

Syntax

```
delete(h)
```

Description

`delete(h)` releases all interfaces derived from the specified COM server or control, and then deletes the server or control itself.

The `release` function, releases and invalidates an interface, but does not delete the server or control.

COM functions are available on Microsoft Windows systems only.

See Also

`save (COM)` | `load (COM)` | `release` | `actxcontrol` | `actxserver`

Introduced before R2006a

delete

Class: FTP

Remove file on FTP server

Syntax

```
delete(ftpobj,filename)
```

Description

`delete(ftpobj,filename)` removes the specified file from the current folder on the FTP server associated with ftpobj.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`filename`

String enclosed in single quotation marks that specifies the name of the file to delete.

Examples

Suppose that a hypothetical host, `ftp.testsite.com`, contains `myfile.m`. Connect to the server and delete the file:

```
test=ftp('ftp.testsite.com');
delete(test,'myfile.m');
```

See Also

`rmdir` | `ftp`

Introduced before R2006a

delete (serial)

Remove serial port object from memory

Syntax

```
delete(obj)
```

Description

`delete(obj)` removes `obj` from memory, where `obj` is a serial port object or an array of serial port objects.

Examples

This example creates the serial port object `s` on a Windows platform, connects `s` to the device, writes and reads text data, disconnects `s` from the device, removes `s` from memory using `delete`, and then removes `s` from the workspace using `clear`.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

More About

Tips

When you delete `obj`, it becomes an *invalid* object. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command. If multiple references to `obj` exist in the workspace, then deleting one reference invalidates the remaining references.

If `obj` is connected to the device, it has a `Status` property value of `open`. If you issue `delete` while `obj` is connected, then the connection is automatically broken. You can also disconnect `obj` from the device with the `fclose` function.

See Also

`clear` | `isvalid` | `fclose` | `Status`

Introduced before R2006a

deleteproperty

Remove custom property from COM object

Syntax

```
deleteproperty(h,'name')
```

Description

`deleteproperty(h,'name')` deletes property specified in string, `name`, from custom properties belonging to object or interface, `h`.

You can only delete properties created with the `addproperty` function.

COM functions are available on Microsoft Windows systems only.

Examples

This example removes a custom property from an instance of the MATLAB sample control.

1 Create an instance of the control:

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f);
get(h)
```

```
     Label: 'Label'
    Radius: 20
```

MATLAB also displays interfaces.

2 Add a custom property named `Position` and assign a value:

```
addproperty(h,'Position');
h.Position = [200 120];
get(h)
```

```
        Label: 'Label'
       Radius: 20
     Position: [200 120]
```

3 Delete the custom property `Position`:

```
deleteproperty(h,'Position');
get(h)
```

```
     Label: 'Label'
    Radius: 20
```

MATLAB displays the original list of properties:

See Also

`addproperty` | `set (COM)` | `get (COM)` | `inspect`

Introduced before R2006a

delevent

Remove `tsdata.event` objects from `timeseries` object

Syntax

```
ts = delevent(ts,event)
ts = delevent(ts,events)
ts = delevent(ts,event,n)
```

Description

`ts = delevent(ts,event)` removes the `tsdata.event` object from the `ts.events` property, where `event` is an event name string.

`ts = delevent(ts,events)` removes the `tsdata.event` object from the `ts.events` property, where `events` is a cell array of event name strings.

`ts = delevent(ts,event,n)` removes the `n`th `tsdata.event` object from the `ts.events` property. `event` is the name of the `tsdata.event` object.

Examples

The following example shows how to remove an event from a `timeseries` object:

1. Create a time series.

   ```
   ts = timeseries(rand(5,4))
   ```

2. Create an event object called `'test'` such that the event occurs at time `3`.

   ```
   e = tsdata.event('test',3)
   ```

3. Add the event object to the time series `ts`.

   ```
   ts = addevent(ts,e)
   ```

4. Remove the event object from the time series `ts`.

   ```
   ts = delevent(ts,'test')
   ```

See Also

addevent | timeseries | tsdata.event

Introduced before R2006a

delsamplefromcollection

Remove sample from `tscollection` object

Syntax

```
tsc = delsamplefromcollection(tsc,'Index',N)
tsc = delsamplefromcollection(tsc,'Value',Time)
```

Description

`tsc = delsamplefromcollection(tsc,'Index',N)` deletes samples from the `tscollection` object tsc. N specifies the indices of the tsc time vector that correspond to the samples you want to delete.

`tsc = delsamplefromcollection(tsc,'Value',Time)` deletes samples from the `tscollection` object tsc. Time specifies the time values that correspond to the samples you want to delete.

See Also

`addsampletocollection` | `tscollection`

Introduced before R2006a

demo

Access product examples in Help browser

Syntax

```
demo
demo type
demo type name
```

Description

`demo` displays the list of MATLAB examples in the Help browser.

`demo type` lists the examples for the specified product. Valid values for `type` are `matlab` or `simulink`.

`demo type name` lists the examples for products other than MATLAB or Simulink. Valid values for `type` include `matlab`, `simulink`, `toolbox`, or `blockset`.

Examples

MATLAB Examples

```
demo matlab
```

Statistics and Machine Learning Toolbox Examples

```
demo toolbox statistics
```

Communications System Toolbox Examples

```
demo toolbox 'communications system'
```

Simulink Control Design Examples

```
demo simulink 'simulink control design'
```

Input Arguments

`type` — Product name or type

`matlab` (default) | `simulink` | `toolbox` | `blockset`

Product name or type, specified as one of these strings: `matlab`, `simulink`, `toolbox`, or `blockset`. For products other than MATLAB or Simulink, you must also specify a name input that corresponds to the product name.

`name` — Product name other than MATLAB or Simulink

string

Product name other than MATLAB or Simulink, specified as a string. If `name` requires multiple words, enclose it in single quotes.

More About

Tips

- To access third-party and custom examples without using the `demo` command, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.

See Also

`doc` | `echodemo` | `grabcode` | `help`

Introduced before R2006a

depdir

List dependent folders for function or P-file

Note: `depdir` will be removed in a future release. Use `matlab.codetools.requiredFilesAndProducts` instead.

Syntax

```
list = depdir('file_name')
[list, prob_files, prob_sym, prob_strings] = depdir('file_name')
[...] = depdir('file_name1', 'file_name2',...)
```

Description

The `depdir` function lists the folders of all the functions that a specified function or P-file needs to operate. This function is useful for finding all the folders that need to be included with a run-time application and for determining the run-time path.

`list = depdir('file_name')` creates a cell array of strings containing the folders of all the function and P-files that `file_name.m` or `file_name.p` uses. This includes the second-level files that are called directly by `file_name`, as well as the third-level files that are called by the second-level files, and so on.

`[list, prob_files, prob_sym, prob_strings] = depdir('file_name')` creates three additional cell arrays containing information about any problems with the `depdir` search. `prob_files` contains filenames that `depdir` was unable to parse. `prob_sym` contains symbols that `depdir` was unable to find. `prob_strings` contains callback strings that `depdir` was unable to parse.

`[...] = depdir('file_name1', 'file_name2',...)` performs the same operation for multiple files. The dependent folders of all files are listed together in the output cell arrays.

Examples

```
list = depdir('mesh')
```

See Also

`matlab.codetools.requiredFilesAndProducts`

Introduced before R2006a

depfun

List dependencies of function or P-file

Compatibility

`depfun` will be removed in a future release. Use `matlab.codetools.requiredFilesAndProducts` instead.

Syntax

```
list = depfun(fun)
[list,builtins] = depfun(fun)
[list,builtins,classes] = depfun(fun)
[list,builtins,classes,prob_files] = depfun(fun)
[list,builtins,classes,prob_files,~,eval_strings] = depfun(fun)
[list,builtins,classes,prob_files,~,eval_strings,called_from] =
depfun(fun)
[list,builtins,classes,prob_files,~,eval_strings,called_from,
opaque_classes] = depfun(fun)

___ = depfun(fun1,...,funN)

___ = depfun( ___ ,option1,...,optionM)
```

Description

`list = depfun(fun)` returns the paths of MATLAB program files that `fun` requires to run, where `fun` is a function, a P-file, or a figure file for a user interface. The output list includes second-level functions that `fun` calls directly, functions that the second-level functions call, and so on. The `depfun` function also displays a report in the Command Window.

Note:

- `depfun` does not always list all dependent files. For example, `depfun` does not list files hidden in callbacks or files whose names are constructed dynamically for evaluation.

- The output list often includes extra files that are not called when the function is actually evaluated.
- If `depfun` returns a parse error for any of the required functions, then the rest of the output of `depfun` might be incomplete. Correct the problematic files and invoke `depfun` again.

`[list,builtins] = depfun(fun)` returns the built-in functions that `fun` requires.

`[list,builtins,classes] = depfun(fun)` returns the MATLAB classes that `fun` requires.

`[list,builtins,classes,prob_files] = depfun(fun)` returns the files that `depfun` cannot parse, find, or access. If there are problematic files (that is, if `prob_files` is not empty), then the rest of the output of `depfun` might be incomplete. Correct the problematic files, and then invoke `depfun` again.

`[list,builtins,classes,prob_files,~,eval_strings] = depfun(fun)` returns a list of files that contain calls to `eval` or related functions (`evalin`, `feval`, or `evalc`). These calls potentially use functions that are not in `list`.

The fifth output argument for `depfun` is not implemented, and returns an empty structure array. To request arguments later in the list, use the tilde symbol (~) as a placeholder. The tilde suppresses the creation of an extra, empty variable.

`[list,builtins,classes,prob_files,~,eval_strings,called_from] = depfun(fun)` returns a cell array of indices that maps each function in `list` to the set of functions that call it. That is, `list{called_from{k}}` returns the paths to the functions that call the function in `list{k}`.

`[list,builtins,classes,prob_files,~,eval_strings,called_from, opaque_classes] = depfun(fun)` returns the opaque classes that `fun` requires, such as Java or COM classes.

`___ = depfun(fun1,...,funN)` returns the functions required for multiple functions `fun1,...,funN`. You can request any of the outputs from the previous syntaxes.

`___ = depfun( ___ ,option1,...,optionM)` modifies the output as described by the specified options. For example, specify `'-toponly'` to list only the functions that `fun` calls directly. You can precede the options with one or more function names.

Examples

Identify Required Functions

Identify the MATLAB program files that `strtok.m` requires to run.

```
fun = 'strtok.m';
list = depfun(fun);
```

MATLAB displays a summary report in the Command Window and stores the list of files in variable `list`.

Identify Top-Level Dependencies Only

List only the program files that `strtok.m` calls directly.

```
fun = 'strtok.m';
list = depfun(fun,'-toponly')
```

```
============================================================
depfun report summary:(top only)
------------------------------------------------------------
-> trace list:           2 files  (total)
                         1 files  (total arguments)
                         0 files  (arguments off MATLABPATH)
                         0 files  (argument duplicates on MATLABPATH)
------------------------------------------------------------
Notes: 1. Use argument  '-quiet' to not print this summary.
       2. Use arguments '-print','file' to produce a full
          report in file.
       3. Use argument  '-all' to display all possible
          left hand side arguments in the report(s).
============================================================

list =

    'matlabroot\toolbox\matlab\strfun\strtok.m'
    'matlabroot\toolbox\matlab\strfun\@cell\strtok.m'
```

Determine Which File Invokes Each Function

One of the functions that `strtok.m` depends upon is `num2cell`. Determine which file in the dependency list calls `num2cell`.

Identify the functions that `strtok.m` requires (`list`) and the files that invoke each of those functions (`called_from`).

```
fun = 'strtok.m';
[list,builtins,classes,prob_files,~,eval_strings,...
    called_from] = depfun(fun);
```

Find the index for `num2cell` within the dependency list.

```
num2cell_path = which('num2cell');
num2cell_index = find(ismember(list,num2cell_path))
```

```
num2cell_index =
     5
```

Identify the file that calls `num2cell`.

```
calls_num2cell = list{called_from{num2cell_index}}
```

```
calls_num2cell =
     matlabroot\toolbox\matlab\elmat\repmat.m
```

Input Arguments

`fun` — Function name

string | cell array of strings

Function name, specified as a string or a cell array of strings.

`fun` must be on the MATLAB path, as determined by the `which` function. If the path contains any relative folders, then files in those folders also will have a relative path in the output.

Example: `'plot.m'`

Example: `'plot.m','mesh.m'`

Example: `{'plot.m','mesh.m'}`

Data Types: `char` | `cell`

`option` — Reporting option

`'-toponly'` | `'-verbose'` | `'-quiet'` | `'-print'` | ...

Reporting option, specified as one of the following strings.

Option	Description
`'-toponly'`	Identify only the files used directly by the specified function(s), `fun`.
`'-verbose'`	Display additional internal messages.
`'-quiet'`	Display only error and warning messages, and not a summary report.
`'-print',filename`	Print a full report to the specified file.
`'-all'`	Display all outputs in the report, but return only the specified output arguments.
`'-expand'`	Include both indices and full paths in an exported report for the `call` or `called_from` list. Requires the `'-print'` option. This information does not appear in the Command Window report.
`'-calltree'`	Replaces the `called_from` list with a list of functions that each file calls, as derived from the `called_from` list.

Example: `'-print','myreport.txt'`

Data Types: `char`

Output Arguments

`list` — Paths to required files

cell array of strings

Paths to required files, returned as a cell array of strings.

`builtins` — Required MATLAB built-in functions

cell array of strings

Required MATLAB built-in functions, returned as a cell array of strings.

`classes` — Required MATLAB classes
cell array of strings

Required MATLAB classes, returned as a cell array of strings.

`prob_files` — Files that `depfun` cannot parse or access
structure

Files that `depfun` cannot parse or access, returned as a structure with these fields:

- `name` — Path to the file
- `listindex` — Index of the file in `list`
- `errmsg` — Error message that describes the problem
- `errid` — Error identifier, if present

`eval_strings` — Files that call evaluation functions
cell array of strings

Files that call evaluation functions, returned as a cell array of strings.

`called_from` — Indices to files that call each function in `list`
cell array

Indices to files that call each function in list, returned as a cell array. Each element of the cell array is a numeric array of indices that correspond to elements in `list`. The list and called_from outputs have the same length.

`opaque_classes` — Paths to opaque classes
cell array of strings

Paths to opaque classes, such as Java or COM classes, returned as a cell array of strings.

More About

- "Identify Program Dependencies"

Introduced before R2006a

det

Matrix determinant

Syntax

```
d = det(A)
```

Description

`d = det(A)` returns the determinant of square matrix `A`.

Examples

Calculate Determinant of Matrix

Create a 3-by-3 square matrix, `A`.

```
A = [1 -2 4; -5 2 0; 1 0 3]
```

```
A =

     1    -2     4
    -5     2     0
     1     0     3
```

Calculate the determinant of `A`.

```
d = det(A)
```

```
d =

   -32
```

The determinant of `A` is -32.

Determine if Matrix Is Singular

Examine why the determinant is not an accurate measure of singularity.

Create a 10-by-10 matrix by multiplying an identity matrix, `eye(10)`, by a small number.

```
A = eye(10)*0.0001;
```

The matrix `A` has very small entries along the main diagonal. However, `A` is *not* singular, because it is a multiple of the identity matrix.

Calculate the determinant of `A`.

```
d = det(A)
```

```
d =

   1.0000e-40
```

The determinant is extremely small. A tolerance test of the form `abs(det(A)) < tol` is likely to flag this matrix as singular. Although the determinant of the matrix is close to zero, `A` is actually not ill conditioned. Therefore, `A` is not close to being singular. The determinant of a matrix can be arbitrarily close to zero without conveying information about singularity.

To investigate if `A` is singular, use either the `cond` or `rcond` functions.

Calculate the condition number of `A`.

```
c = cond(A)
```

```
c =

     1
```

The result confirms that `A` is not ill conditioned.

Compute Determinant of Inverse of Ill-Conditioned Matrix

Examine how to calculate the determinant of the matrix inverse `A^(-1)`, for an ill-conditioned matrix `A`, without explicitly calculating `A^(-1)`.

Create a 10-by-10 Hilbert matrix, `A`.

```
A = hilb(10);
```

Find the condition number of `A`.

```
c = cond(A)

c =

   1.6025e+13
```

The large condition number suggests that A is close to being singular, so calculating `inv(A)` might produce inaccurate results. Therefore, the inverse determinant calculation `det(inv(A))` is also inaccurate.

Calculate the determinant of the inverse of A by exploiting the fact that $\det\left(A^{-1}\right)=\frac{1}{\det(A)}$.

```
d1 = 1/det(A)

d1 =

   4.6202e+52
```

This method avoids computing the inverse of the matrix, A.

Calculate the determinant of the exact inverse of the Hilbert matrix, A, using `invhilb`. Compare the result to `d1` to find the relative error in `d1`.

```
d = det(invhilb(10));
relError = abs(d1-d)/abs(d)

relError =

   1.1462e-04
```

The relative error in `d1` is reasonably small. Avoiding the explicit computation of the inverse of A minimizes it.

For comparison, also calculate the determinant of the inverse of A by explicitly calculating the inverse. Compare the result to `d` to see the relative error.

```
d2 = det(inv(A));
relError2 = abs(d2-d)/abs(d)

relError2 =

   1.2427e+08
```

The relative error in the calculation of d2 is many orders of magnitude larger than that of d1.

Find Determinant of Singular Matrix

Examine a matrix that is exactly singular, but which has a large nonzero determinant. In theory, the determinant of any singular matrix is zero, but because of the nature of floating-point computation, this ideal is not always achievable.

Create a 13-by-13 diagonally dominant singular matrix, A.

```
A = diag([24 46 64 78 88 94 96 94 88 78 64 46 24]);
S = diag([-13 -24 -33 -40 -45 -48 -49 -48 -45 -40 -33 -24],1);
A = A + S + rot90(S,2)
```

```
A =

    24   -13     0     0     0     0     0     0     0     0     0     0     0
   -24    46   -24     0     0     0     0     0     0     0     0     0     0
     0   -33    64   -33     0     0     0     0     0     0     0     0     0
     0     0   -40    78   -40     0     0     0     0     0     0     0     0
     0     0     0   -45    88   -45     0     0     0     0     0     0     0
     0     0     0     0   -48    94   -48     0     0     0     0     0     0
     0     0     0     0     0   -49    96   -49     0     0     0     0     0
     0     0     0     0     0     0   -48    94   -48     0     0     0     0
     0     0     0     0     0     0     0   -45    88   -45     0     0     0
     0     0     0     0     0     0     0     0   -40    78   -40     0     0
     0     0     0     0     0     0     0     0     0   -33    64   -33     0
     0     0     0     0     0     0     0     0     0     0   -24    46   -24
     0     0     0     0     0     0     0     0     0     0     0   -13    24
```

A is singular because the rows are linearly dependent. For instance, sum(A) produces a vector of zeros.

Calculate the determinant of A.

```
d = det(A)
```

```
d =

     1.059686712222177e+05
```

The determinant of A is quite large despite the fact that A is singular. In fact, the determinant of A should be exactly zero! The inaccuracy of d is due to an aggregation of

round-off errors in the MATLAB implementation of the LU decomposition, which `det` uses to calculate the determinant. This result demonstrates a few important aspects of calculating numeric determinants. See the "Limitations" on page 1-2118 section for more details.

Input Arguments

A — Input matrix

square numeric matrix

Input matrix, specified as a square numeric matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Limitations

Avoid using `det` to examine if a matrix is singular because of the following limitations. Use `cond` or `rcond` instead.

Limitation	Result
The magnitude of the determinant is typically unrelated to the condition number of a matrix.	The determinant of a matrix can be arbitrarily large or small without changing the condition number.
`det` uses the LU decomposition to calculate the determinant, which is susceptible to floating-point round-off errors.	The determinant calculation is sometimes numerically unstable. For example, `det` can produce a large-magnitude determinant for a singular matrix, even though it should have a magnitude of 0.

More About

Algorithms

`det` computes the determinant from the triangular factors obtained by Gaussian elimination with the `lu` function.

```
[L,U] = lu(X)
s =  det(L)      % This is always +1 or -1
det(X) = s*prod(diag(U))
```

- “Inverses and Determinants”

See Also

cond | condest | inv | lu | mldivide | rcond | rref

Introduced before R2006a

details

Display array details

Syntax

```
details(A)
```

Description

`details(A)` displays detailed information about the array, `A`.

When `A` is a MATLAB object array, `details` displays more information than the default display. This information includes:

- Fully qualified class name, including package names
- Link to class documentation
- Link to `handle` class documentation for classes that subclass `handle`
- List of all properties that have public get access
- List of property values if the array is scalar
- Link to list of public methods
- Link to list of events
- Link to list of all nonhidden superclasses

Examples

Display Object Details

Display object details for a class that overloads its own object display. The `details` function never calls overloaded display methods. Therefore, you can use this function to obtain information about the object array in all cases.

Suppose `PolyNom` is a class that provides a specialized default display for polynomials. Use `details` to display information about the object.

Create an object using the polynomial coefficients:

```
pn = PolyNom([1,2,3,0,4])
```

The overloaded `disp` method displays the code for evaluating the polynomial:

```
pn =

x^4 + 2*x^3 + 3*x^2 + 4
```

Calling the `details` function provides information about the object:

```
details(pn)
```

```
 PolyNom with properties:

    coef: [1 2 3 0 4]

  Methods
```

Input Arguments

A — Input array

scalar or nonscalar array of any type

Input array, specified as a scalar or nonscalar array of any type. The `details` function displays detailed information about this array.

See Also

`classdef` | `disp` | `display`

detrend

Remove linear trends

Syntax

```
y = detrend(x)
y = detrend(x,'constant')
y = detrend(x,'linear',bp)
```

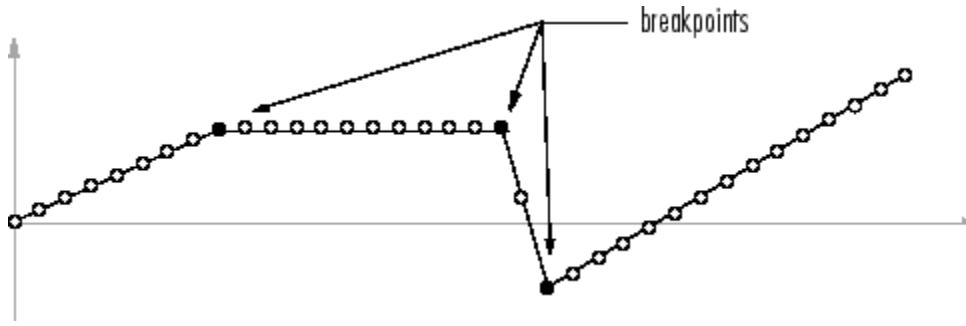
Description

`detrend` removes the mean value or linear trend from a vector or matrix, usually for FFT processing.

`y = detrend(x)` removes the best straight-line fit from vector `x` and returns it in `y`. If `x` is a matrix, `detrend` removes the trend from each column.

`y = detrend(x,'constant')` removes the mean value from vector `x` or, if `x` is a matrix, from each column of the matrix.

`y = detrend(x,'linear',bp)` removes a continuous, piecewise linear trend from vector `x` or, if `x` is a matrix, from each column of the matrix. Vector `bp` contains the indices of the breakpoints between adjacent linear segments. The breakpoint between two segments is defined as the data point that the two segments share.



`detrend(x,'linear')`, with no breakpoint vector specified, is the same as `detrend(x)`.

Examples

```
sig = [0 1 -2 1 0 1 -2 1 0];      % signal with no linear trend
trend = [0 1 2 3 4 3 2 1 0];      % two-segment linear trend
x = sig+trend;                    % signal with added trend
y = detrend(x,'linear',5)         % breakpoint at 5th element

y =

   -0.0000
    1.0000
   -2.0000
    1.0000
    0.0000
    1.0000
   -2.0000
    1.0000
   -0.0000
```

Note that the breakpoint is specified to be the fifth element, which is the data point shared by the two segments.

More About

Algorithms

`detrend` computes the least-squares fit of a straight line (or composite line for piecewise linear trends) to the data and subtracts the resulting function from the data. To obtain the equation of the straight-line fit, use `polyfit`.

See Also

`polyfit`

Introduced before R2006a

deval

Evaluate solution of differential equation problem

Syntax

```
sxint = deval(sol,xint)
sxint = deval(xint,sol)
sxint = deval(sol,xint,idx)
sxint = deval(xint,sol,idx)
[sxint, spxint] = deval(...)
```

Description

`sxint = deval(sol,xint)` and `sxint = deval(xint,sol)` evaluate the solution of a differential equation problem. `sol` is a structure returned by one of these solvers:

- An initial value problem solver (`ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode15i`)
- A delay differential equations solver (`dde23`, `ddesd`, or `ddensd`),
- A boundary value problem solver (`bvp4c` or `bvp5c`).

`xint` is a point or a vector of points at which you want the solution. The elements of `xint` must be in the interval `[sol.x(1),sol.x(end)]`. For each `i`, `sxint(:,i)` is the solution at `xint(i)`.

`sxint = deval(sol,xint,idx)` and `sxint = deval(xint,sol,idx)` evaluate as above but return only the solution components with indices listed in the vector `idx`.

`[sxint, spxint] = deval(...)` also returns `spxint`, the value of the first derivative of the polynomial interpolating the solution.

Note For multipoint boundary value problems, the solution obtained by `bvp4c` or `bvp5c` might be discontinuous at the interfaces. For an interface point `xc`, `deval` returns the average of the limits from the left and right of `xc`. To get the limit values, set the `xint` argument of `deval` to be slightly smaller or slightly larger than `xc`.

Examples

Evaluate van der Pol Equation

This example solves the system `y' = vdp1(t,y)` using `ode45` and plots the first component of the solution.

Solve the system using `ode45`.

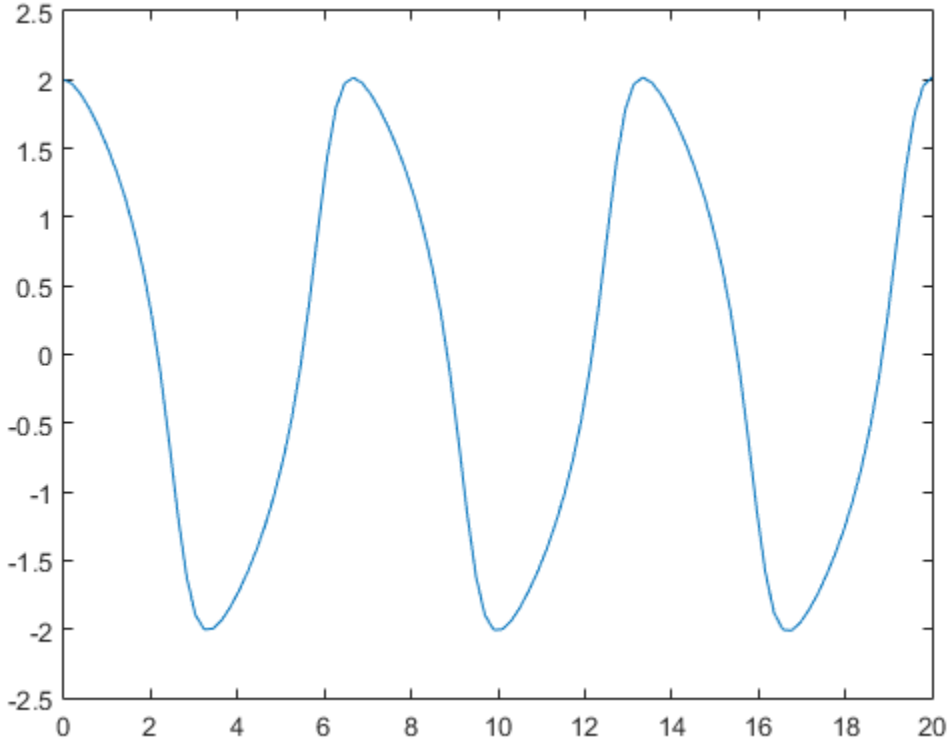
```
sol = ode45(@vdp1,[0 20],[2 0]);
```

Evaluate the first component of the solution at 100 points in the interval $[0, 20]$.

```
x = linspace(0,20,100);
y = deval(sol,x,1);
```

Plot the solution.

```
plot(x,y)
```



See Also

`ode45` | `ode23` | `ode113` | `ode15s` | `ode23s` | `ode23t` | `ode23tb` | `ode15i` | `dde23` | `ddesd` | `ddensd` | `bvp4c` | `bvp5c`

Introduced before R2006a

diag

Create diagonal matrix or get diagonal elements of matrix

Syntax

```
D = diag(v)
D = diag(v,k)

x = diag(A)
x = diag(A,k)
```

Description

`D = diag(v)` returns a square diagonal matrix with the elements of vector `v` on the main diagonal.

`D = diag(v,k)` places the elements of vector `v` on the `k`th diagonal. `k=0` represents the main diagonal, `k>0` is above the main diagonal, and `k<0` is below the main diagonal.

`x = diag(A)` returns a column vector of the main diagonal elements of `A`.

`x = diag(A,k)` returns a column vector of the elements on the `k`th diagonal of `A`.

Examples

Create Diagonal Matrices

Create a 1-by-5 vector.

```
v = [2 1 -1 -2 -5];
```

Use `diag` to create a matrix with the elements of `v` on the main diagonal.

```
D = diag(v)
```

```
D =
```

```
     2     0     0     0     0
     0     1     0     0     0
     0     0    -1     0     0
     0     0     0    -2     0
     0     0     0     0    -5
```

Create a matrix with the elements of `v` on the first super diagonal (`k=1`).

```
D1 = diag(v,1)
```

```
D1 =

     0     2     0     0     0     0
     0     0     1     0     0     0
     0     0     0    -1     0     0
     0     0     0     0    -2     0
     0     0     0     0     0    -5
     0     0     0     0     0     0
```

The result is a 6-by-6 matrix. When you specify a vector of length `n` as an input, `diag` returns a square matrix of size `n+abs(k)`.

Get Diagonal Elements

Get the elements on the main diagonal of a random 6-by-6 matrix.

```
A = randi(10,6)
```

```
A =

     9     3    10     8     7     8
    10     6     5    10     8     1
     2    10     9     7     8     3
    10    10     2     1     4     1
     7     2     5     9     7     1
     1    10    10    10     2     9
```

```
x = diag(A)
```

```
x =

     9
     6
     9
     1
```

```
     7
     9
```

Get the elements on the first subdiagonal (`k=-1`) of `A`. The result has one fewer element than the main diagonal.

```
x1 = diag(A,-1)
```

```
x1 =

    10
    10
     2
     9
     2
```

Calling `diag` twice returns a diagonal matrix composed of the diagonal elements of the original matrix.

```
A1 = diag(diag(A))
```

```
A1 =

     9     0     0     0     0     0
     0     6     0     0     0     0
     0     0     9     0     0     0
     0     0     0     1     0     0
     0     0     0     0     7     0
     0     0     0     0     0     9
```

Input Arguments

`v` — Diagonal elements

vector

Diagonal elements, specified as a vector. If `v` is a vector with `N` elements, then `diag(v,k)` is a square matrix of order `N+abs(k)`.

`diag([])` returns an empty matrix, `[]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

A — Input matrix

matrix

Input matrix. `diag` returns an error if `ndims(A) > 2`.

`diag([])` returns an empty matrix, `[]`.

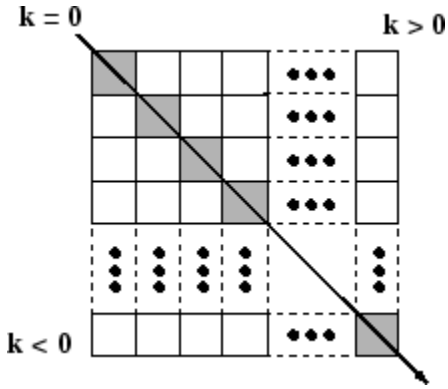
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

k — Diagonal number

integer

Diagonal number, specified as an integer. `k=0` represents the main diagonal, `k>0` is above the main diagonal, and `k<0` is below the main diagonal.

For an m-by-n matrix, `k` is in the range $(-m+1) \le k \le (n-1)$.



More About

Tips

- The `trace` of a matrix is equal to `sum(diag(A))`.

See Also

`blkdiag` | `isdiag` | `istril` | `istriu` | `spdiags` | `tril` | `triu`

Introduced before R2006a

matlab.unittest.diagnostics

Summary of classes in MATLAB Diagnostics Interface

Description

Use diagnostics to communicate relevant information in the event of a failure. To add a diagnostic message to a test case, use the `diagnostic` argument in any of the `matlab.unittest.qualifications` methods. The framework also displays diagnostic messages related to the nature of the qualification failure. The `matlab.unittest.diagnostics` package consists of the following classes:

matlab.unittest.diagnostics.ConstraintDiagnostic	Diagnostics specific to `matlab.unittest` constraints
matlab.unittest.diagnostics.Diagnostic	Fundamental interface class for `matlab.unittest` diagnostics
matlab.unittest.diagnostics.DisplayDiagnostic	Diagnostic using a value's displayed output
matlab.unittest.diagnostics.FunctionHandleDiagnostic	Diagnostic using a function's displayed output
matlab.unittest.diagnostics.StringDiagnostic	Diagnostic using string

The package contains the following event data classes:

matlab.unittest.diagnostics.LoggedDiagnosticEventData	Event data for DiagnosticLogged event listeners

matlab.unittest.diagnostics.ConstraintDiagnostic class

Package: matlab.unittest.diagnostics
Superclasses: matlab.unittest.diagnostics.Diagnostic

Diagnostics specific to `matlab.unittest` constraints

Description

The `ConstraintDiagnostic` class provides various textual fields that are common to most constraints. These fields may be turned on or off depending on their applicability.

The `ConstraintDiagnostic` class is a helper class for displaying diagnostics when using constraints. The `ConstraintDiagnostic` class provides custom constraint authors a way to add a common look and feel to diagnostics produced by the `getDiagnosticFor` method of constraints.

Constraint diagnostics are displayed in the following order: Description, Conditions, Actual Value, and Expected Value.

Properties

`ActVal`

The actual value passed to the constraint for testing.

`ActValHeader`

Header information for the actual value property, `ActVal`, specified as a string. The default header is `'Actual Value:'`.

`Conditions`

Formatted list of conditions, specified as a single string. Each condition starts on a new line and begins with an arrow (`-->`) delimiter. Conditions are added to the list using the `addCondition` and `addConditionsFrom` methods.

`ConditionsCount`

Number of conditions in the condition list. This is a read-only property generated from the conditions list. The conditions list is defined in the `Conditions` property.

Description

General diagnostic information, specified as a string.

DisplayActVal

Indicator whether to display the actual value property, `ActVal`, specified as a boolean. By default, the actual value is not displayed and the value of this property is `false`.

DisplayConditions

Indicator of whether to display the `Conditions` property, specified as a boolean. By default, the conditions are not displayed and the value of this property is `false`. Even if `DisplayConditions` is set to `true`, if there are no conditions on the conditions list, neither the conditions header or the conditions list are displayed.

DisplayDescription

Indicator of whether to display the `Description` property, specified as a boolean. By default, the description is not displayed and the value of this property is `false`.

DisplayExpVal

Indicator whether to display the expected value property, `ExpVal`, specified as a boolean. By default, the expected value is not displayed and the value of this property is `false`.

ExpVal

If applicable, the expected value. This property can be turned off if the associated constraint does not contain an expected value.

ExpValHeader

Header information for the expected value property, `ExpVal`, specified as a string. The default header is `'Expected Value:'`.

Inherited Properties

DiagnosticResult

The `DiagnosticResult` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a string that is defined during evaluation of the `diagnose` method.

Methods

addCondition	Add condition to condition list
addConditionsFrom	Add condition from another `ConstraintDiagnostic` to condition list
getDisplayableString	Convert object to string for display
getPreDescriptionString	Returns text to be displayed prior to description
getPostDescriptionString	Returns text to be displayed following description
getPostConditionString	Returns text to be displayed following conditions list
getPostActValString	Returns text to be displayed following actual value
getPostExpValString	Returns text to be displayed following expected value

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`Diagnostic` | `matlab.unittest.constraints` | `matlab.unittest.diagnostics`

addCondition

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Add condition to condition list

Syntax

```
addCondition(diag, cond)
```

Description

`addCondition(diag, cond)` adds the condition, `cond`, to the condition list. Add conditions to the condition list one at a time. When the condition list is displayed, each condition is preceded by an arrow (`-->`) delimiter and indented.

Input Arguments

`cond`

Condition, specified as a string containing information specific to the cause of the constraint failure or another `Diagnostic` instance, which acts as a "subdiagnostic".

`diag`

`matlab.unittest.diagnostics.Diagnostic` instance.

See Also

`addConditionsFrom`

addConditionsFrom

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Add condition from another `ConstraintDiagnostic` to condition list

Syntax

```
addConditionsFrom(constDiag, otherConstDiag)
```

Description

`addConditionsFrom(constDiag, otherConstDiag)` adds the conditions from the `ConstraintDiagnostic` instance, `constDiag`, to the condition list in the `Diagnostic` instance, `diag`. This is useful when a constraints composes another constraint, and needs to use the conditions produced in the diagnostics of the composed constraint.

Input Arguments

`constDiag`

Diagnostic to add conditions to, specified as a `matlab.unittest.diagnostics.ConstraintDiagnostic` instance

`otherConstDiag`

Diagnostic to add conditions from, specified as a `matlab.unittest.diagnostics.ConstraintDiagnostic` instance

Examples

Add Conditions from a Constraint

```
% This demonstrates a constraint that composes another constraint
```

```
% and uses the addConditionsFrom method to utilize the conditions
% from the composed ConstraintDiagnostic.
classdef IsDouble < matlab.unittest.constraints.Constraint

    properties(Constant, GetAccess=private)
        DoubConst = matlab.unittest.constraints.IsInstanceOf(?double);
    end

    methods
        function tf = satisfiedBy(constraint, actual)
            tf = constraint.DoubConst.satisfiedBy(actual);
        end
        function diag = getDiagnosticFor(constraint, actual)
            diag = ConstraintDiagnostic;

            % Now add conditions from the IsInstanceOf
            % Diagnostic
            otherDiag = constraint.DoubConst.getDiagnosticFor(actual);
            diag.addConditionsFrom(otherDiag)

            % ...
        end
    end
end
```

See Also

addCondition

matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplaya

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Convert object to string for display

Syntax

```
str =
matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplayableString(
value)
```

Description

`str = matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplayableString(value)` converts the object, `obj` to a string, `str` for display in a diagnostic result. This conversion determines if hotlinks should be included in the string and truncates large numeric or cell arrays.

Input Arguments

`value`

Object of arbitrary class

getPreDescriptionString

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Returns text to be displayed prior to description

Syntax

```
str = getPreDescriptionString(constDiag)
```

Description

`str = getPreDescriptionString(constDiag)` returns text to be displayed prior to the description. This method can be overridden to inject strings prior to displaying the `Description` property of the `ConstraintDiagnostic`. The location of this text is tied to the `Description` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostConditionString` | `getPostDescriptionString` | `getPostExpValString`

getPostDescriptionString

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Returns text to be displayed following description

Syntax

```
str = getPostDescriptionString(constDiag)
```

Description

`str = getPostDescriptionString(constDiag)` returns text to be displayed following the description. This method can be overridden to inject strings subsequent to displaying the `Description` property of the `ConstraintDiagnostic`. The location of this text is tied to the `Description` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostConditionString` | `getPostExpValString` | `getPreDescriptionString`

getPostConditionString

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Returns text to be displayed following conditions list

Syntax

```
str = getPostConditionsString(constDiag)
```

Description

`str = getPostConditionsString(constDiag)` returns text to be displayed following the conditions list. This method can be overridden to inject strings subsequent to displaying the `Conditions` property of the `ConstraintDiagnostic`. The location of this text is tied to the `Conditions` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostDescriptionString` | `getPostExpValString` | `getPreDescriptionString`

getPostActValString

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Returns text to be displayed following actual value

Syntax

```
str = getPostActualValString(constDiag)
```

Description

`str = getPostActualValString(constDiag)` returns text to be displayed following the actual value. This method can be overridden to inject strings subsequent to displaying the `ActVal` property of the `ConstraintDiagnostic`. The location of this text is tied to the `ActVal` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostConditionString` | `getPostDescriptionString` | `getPostExpValString` | `getPreDescriptionString`

getPostExpValString

Class: matlab.unittest.diagnostics.ConstraintDiagnostic
Package: matlab.unittest.diagnostics

Returns text to be displayed following expected value

Syntax

```
str = getPostExpValString(constDiag)
```

Description

`str = getPostExpValString(constDiag)` returns text to be displayed following the expected value. This method can be overridden to inject strings subsequent to displaying the `ExpVal` property of the `ConstraintDiagnostic`. The location of this text is tied to the `ExpVal` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostConditionString` | `getPostDescriptionString` | `getPreDescriptionString`

matlab.unittest.diagnostics.Diagnostic class

Package: matlab.unittest.diagnostics

Fundamental interface class for `matlab.unittest` diagnostics

Description

The `Diagnostic` interface class is the means by which the `matlab.unittest` framework and its clients package diagnostic information. All diagnostics are derived from `Diagnostic`, whether they are user-supplied test diagnostics for an individual comparison or diagnostics associated with the `Constraint` used in the comparison.

Classes which derive from `Diagnostic` encode the diagnostic actions to be performed. They produce a diagnostic result that is displayed appropriately by the test running framework. In exchange for meeting this requirement, any `Diagnostic` implementation can be used directly with `matlab.unittest` qualifications. These qualifications execute the diagnostic action and store the result to be utilized by the test running framework.

As a convenience, the framework creates appropriate diagnostic instances for strings and function handles when they are user supplied test diagnostics. To retain good performance, these values are only converted into `Diagnostic` instances when a qualification failure occurs or when the test running framework is explicitly observing passing qualifications. The default test runner does not explicitly observe passing qualifications.

Properties

`DiagnosticResult`

The `DiagnosticResult` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a string that is defined during evaluation of the `diagnose` method.

Methods

diagnose	Execute diagnostic action

join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Use Diagnostic Result

```
import matlab.unittest.constraints.IsEqualTo

% Create a TestCase for interactive use
testCase = matlab.unittest.TestCase;

% Create StringDiagnostic upon failure
testCase.verifyThat(1, IsEqualTo(2), 'User supplied Diagnostic')

% Create FunctionHandleDiagnostic upon failure
testCase.verifyThat(1, IsEqualTo(2), @() system('ps'))

% Usage of user defined Diagnostic upon failure (see definition below)
testCase.verifyThat(1, IsEqualTo(2), ProcessStatusDiagnostic...
    ('Could not close my third party application!'))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Diagnostic definition
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
classdef ProcessStatusDiagnostic < matlab.unittest.diagnostics.Diagnostic
    % ProcessStatusDiagnostic - an example diagnostic
    %
    %   Simple example to demonstrate how to create a custom
    %   diagnostic.

    properties

        % HeaderText - user-supplied header to display
```

```
            HeaderText = '(No header supplied)';
        end

        methods
            function diag = ProcessStatusDiagnostic(header)
                % Constructor - construct a ProcessStatusDiagnostic
                %
                %   The ProcessStatusDiagnostic constructor takes an
                %   optional header to be displayed along with process
                %   information.
                if (nargin >0)
                    diag.HeaderText = header;
                end
            end

            function diagnose(diag)

                [status, processInfo] = system('ps');
                if (status ~= 0)
                    processInfo = sprintf(...
                        ['!!! Could not obtain status diagnostic information!!!'...
                        ' [exit status code: %d]\n%s'], status, processInfo);
                end
                diag.DiagnosticResult = sprintf('%s\n%s', diag.HeaderText,...
                    processInfo);
            end
        end

    end % classdef
```

See Also

`matlab.unittest.plugins.DiagnosticsValidationPlugin` | `matlab.unittest.constraints.Constraint` | `FunctionHandleDiagnostic` | `matlab.unittest.diagnostics` | `StringDiagnostic`

More About

- “Types of Qualifications”

diagnose

Class: matlab.unittest.diagnostics.Diagnostic
Package: matlab.unittest.diagnostics

Execute diagnostic action

Syntax

```
diagnose(diag)
```

Description

`diagnose(diag)` executes diagnostic action for the `matlab.unittest.diagnostics.Diagnostic` instance, `diag`. The `diagnose` method is the means by which individual `Diagnostic` implementations can perform their respective diagnostic evaluations. Each concrete implementation is responsible for populating the `DiagnosticResult` property of the `Diagnostic` object. Typically, text printed to the Command Window during diagnostic evaluation is not considered part of the diagnostic result and is ignored by the testing framework.

See Also

`Diagnostic`

join

Class: matlab.unittest.diagnostics.Diagnostic
Package: matlab.unittest.diagnostics

Join multiple diagnostics into a single array

Syntax

```
diagArray = join(diag1,...,diagN)
```

Description

`diagArray = join(diag1,...,diagN)` joins multiple diagnostics, specified by `diag1` through `diagN`, into a single array, `diagArray`.

Input Arguments

`diag`

Diagnostic content, specified as an instance of a `Diagnostic` object, a string, a function handle, or an arbitrary type.

Output Arguments

`diagArray`

Array of joined diagnostic content.

- If `diagN` is an object that derives from `Diagnostic`, it is included in the array unmodified.
- If `diagN` is a `char`, it is formed into a `StringDiagnostic` and included in the array.
- If `diagN` is a `function_handle`, it is formed into a `FunctionHandleDiagnostic` and included in the array.
- If `diagN` is any other type, it is formed into a `DisplayDiagnostic` and included in the array.

Alternatives

You can use array concatenation join diagnostics into an array if at least one of the values is a diagnostic. The `join` method prevents the need to have any `Diagnostics` in the array. Considering the following example.

```
arbitraryValue = 5;
testCase.verifyThat(false, IsTrue, ...
    ['should have been true', ...
    @() system('ps'), ...
    arbitraryValue, ...
    MyCustomDiagnostic]);
```

Since `MyCustomDiagnostic` is a `Diagnostic`, the other values are correctly converted to diagnostics as well.

Examples

Join Diagnostic Content

```
% The following example creates a diagnostic array of length 4,
% demonstrating standard Diagnostic conversions. Note:
% MyCustomDiagnostic is for example purposes and is not executable
% code.

import matlab.unittest.diagnostics.Diagnostic
import matlab.unittest.constraints.IsTrue

arbitraryValue = 5;
testCase.verifyThat(false, IsTrue, ...
    Diagnostic.join(...
        'should have been true', ...
        @() system('ps'), ...
        arbitraryValue, ...
        MyCustomDiagnostic))
```

See Also

`matlab.mixin.Heterogeneous`

matlab.unittest.diagnostics.DisplayDiagnostic class

Package: matlab.unittest.diagnostics
Superclasses: matlab.unittest.diagnostics.Diagnostic

Diagnostic using a value's displayed output

Description

The `DisplayDiagnostic` class provides a diagnostic result that uses a value's displayed output. This output is the same text displayed using the `display` function. When the diagnostic information is accessible through a variable in the current workspace, the `DisplayDiagnostic` class is a means to provide quick diagnostic information.

Construction

`DisplayDiagnostic(diagValue)` creates a new `DisplayDiagnostic` instance.

Input Arguments

`diagValue`

The value that the `Diagnostic` uses to generate diagnostic information.

The resulting diagnostic information is equivalent to displaying this value at the MATLAB command prompt. The result is packaged for consumption by the testing framework, which may or may not display the information at the command prompt.

Properties

`Value`

The value that the `Diagnostic` uses to generate diagnostic information, specified in the `diagValue` input argument. This property is read-only.

Inherited Properties

`DiagnosticResult`

The `DiagnosticResult` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a string that is defined during evaluation of the `diagnose` method.

Methods

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create `DisplayDiagnostic` Object

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.diagnostics.DisplayDiagnostic

testCase = TestCase.forInteractiveUse;
```

Use a `DisplayDiagnostic` to display diagnostic information upon test failure.

```
testCase.verifyThat(1, IsEqualTo(2), DisplayDiagnostic(inputParser))

Interactive verification failed.

----------------
Test Diagnostic:
----------------
      inputParser with properties:

           FunctionName: ''
          CaseSensitive: 0
          KeepUnmatched: 0
        PartialMatching: 1
           StructExpand: 1
             Parameters: {1x0 cell}
                Results: [1x1 struct]
              Unmatched: [1x1 struct]
          UsingDefaults: {1x0 cell}

--------------------
Framework Diagnostic:
--------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        1         2           -1       -0.5

Actual Value:
         1
Expected Value:
         2
```

In the test diagnostic section of the output, the output from `inputParser` object is the same as MATLAB displays at the command prompt.

See Also

`FunctionHandleDiagnostic` | `matlab.unittest.diagnostics` | `StringDiagnostic`

matlab.unittest.diagnostics.FunctionHandleDiagnostic class

Package: matlab.unittest.diagnostics
Superclasses: matlab.unittest.diagnostics.Diagnostic

Diagnostic using a function's displayed output

Description

The `FunctionHandleDiagnostic` class provides a diagnostic result using a function's displayed output. This output is the same as the text displayed at the command prompt when MATLAB executes the function handle. When the diagnostic information is accessible through information displayed as output of the function handle, the `FunctionHandleDiagnostic` is a means to provide quick diagnostic information.

When using `matlab.unittest` qualifications, a function handle can be supplied directly as a test diagnostic. In this case, the testing framework automatically creates a `FunctionHandleDiagnostic` object.

Construction

`FunctionHandleDiagnostic(fcnHandle)` creates a new `FunctionHandleDiagnostic` instance.

Input Arguments

`fcnHandle`

The function handle that the `Diagnostic` uses to generate diagnostic information.

The resulting diagnostic information is equivalent to output displayed at the MATLAB command prompt. The result is packaged for consumption by the testing framework, which may or may not display the information at the command prompt.

Properties

Fcn

The function handle that the `Diagnostic` uses to generate diagnostic information, specified in the `fcnHandle` input argument. This property is read-only.

Inherited Properties

DiagnosticResult

The `DiagnosticResult` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a string that is defined during evaluation of the `diagnose` method.

Methods

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create FunctionHandleDiagnostic Object

Create a diagnostic result that displays the output of the `dir` function when a test fails.

Create a folder in your current working folder.

```
mkdir('subfolderInCurrentFolder')
```

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.diagnostics.FunctionHandleDiagnostic

testCase = TestCase.forInteractiveUse;
```

Use a `FunctionHandleDiagnostic` to display diagnostic information upon test failure.

```
testCase.verifyThat(1, IsEqualTo(2), FunctionHandleDiagnostic(@dir))
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------

.                         ..                        subfolderInCurrentFolder



---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        1         2           -1       -0.5

Actual Value:
         1
Expected Value:
         2
```

Upon test failure, the diagnostic displays the contents of the current working folder. In this example output, the folder only contains the subfolder `subfolderInCurrentFolder`.

Alternatively, the test framework can create a `FunctionHandleDiagnostic` object for you from a function handle input to the `verifyThat` qualification.

```
testCase.verifyThat(1, IsEqualTo(2), @dir)
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------

.                         ..                         subfolderInCurrentFolder



--------------------
Framework Diagnostic:
--------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        1         2           -1       -0.5

Actual Value:
         1
Expected Value:
         2
```

The testing framework only creates the `FunctionHandleDiagnostic` object as needed, typically only in the event of a test failure.

See Also

`matlab.unittest.plugins.DiagnosticsValidationPlugin` | `matlab.unittest.diagnostics` | `StringDiagnostic`

matlab.unittest.diagnostics.LoggedDiagnosticEventData class

Package: matlab.unittest.diagnostics

Event data for DiagnosticLogged event listeners

Description

The `LoggedDiagnosticEventData` class holds event data for `DiagnosticLogged` event listeners. Invoking the `log` method within your tests triggers the `DiagnosticLogged` event listeners. Only the test framework constructs this class directly.

Properties

`Verbosity`

Verbosity level of the logged message, represented as a `matlab.unittest.Verbosity` enumeration

`Timestamp`

Date and time of the call to the `log` method, represented as a `datetime` value

`Diagnostic`

Diagnostic specified in the call to the `log` method, represented as a string, function handle, or instance of `matlab.unittest.diagnostics.Diagnostic`

`Stack`

Function call stack leading up to the call to the `log` method, represented as a structure array

`DiagnosticResult`

Logged diagnostic messages, represented as a cell array of strings

See Also

`matlab.unittest.TestCase.log` | `matlab.unittest.TestCase` | `matlab.unittest.fixtures.Fixture.log` | `matlab.unittest.fixtures.Fixture`

matlab.unittest.diagnostics.StringDiagnostic class

Package: matlab.unittest.diagnostics
Superclasses: matlab.unittest.diagnostics.Diagnostic

Diagnostic using string

Description

The `StringDiagnostic` class provides a diagnostic result that uses a string. When the diagnostic information is known at the time of construction, the `StringDiagnostic` is a means to provide quick diagnostic information.

When using `matlab.unittest` qualifications, a string can be supplied directly as a test diagnostic. In this case, the testing framework automatically creates a `StringDiagnostic` object.

Construction

`StringDiagnostic(diagString)` creates a new `StringDiagnostic` instance.

Input Arguments

`diagString`

The string that the `Diagnostic` uses to generate diagnostic information.

Properties

Inherited Properties

`DiagnosticResult`

The `DiagnosticResult` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a string that is defined during evaluation of the `diagnose` method.

Methods

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create `StringDiagnostic` Object

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.diagnostics.StringDiagnostic

testCase = TestCase.forInteractiveUse;
```

Use a `StringDiagnostic` to display diagnostic information upon test failure.

```
testCase.verifyThat(1, IsEqualTo(2), ...
    StringDiagnostic('actual was supposed to be equal to expected') )
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
actual was supposed to be equal to expected
```

```
---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        1         2           -1       -0.5

Actual Value:
         1
Expected Value:
         2
```

Alternatively, the test framework can create a `StringDiagnostic` object for you from a string input to the `verifyThat` qualification.

```
testCase.verifyThat(1, IsEqualTo(2), ...
    'actual was supposed to be equal to expected' )

Interactive verification failed.

----------------
Test Diagnostic:
----------------
actual was supposed to be equal to expected

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        1         2           -1       -0.5

Actual Value:
         1
```

```
Expected Value:
         2
```

The testing framework only creates the `StringDiagnostic` object as needed, typically only in the event of a test failure.

See Also

`FunctionHandleDiagnostic` | `matlab.unittest.diagnostics`

dialog

Create empty modal dialog box

The `dialog` function creates an empty modal dialog box, which is a figure window with properties set to make the window look and behave like a dialog box.

Syntax

```
d = dialog
d = dialog(Name,Value)
```

Description

`d = dialog` creates an empty dialog box and returns d, a figure object. Use the `uicontrol` function to add user interface controls to a dialog.

`d = dialog(Name,Value)` specifies one or more figure property names and corresponding values. Use this syntax to override the default properties.

Examples

Dialog Containing Text and a Button

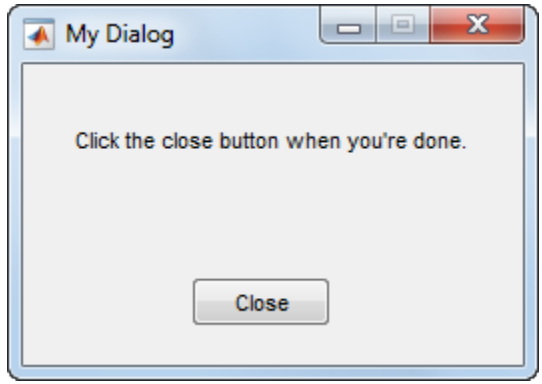
Use the `uicontrol` function to add user interface controls to a dialog box. For instance, create a program file called `mydialog.m` that displays a dialog containing text and a button.

```
function mydialog
    d = dialog('Position',[300 300 250 150],'Name','My Dialog');

    txt = uicontrol('Parent',d,...
               'Style','text',...
               'Position',[20 80 210 40],...
               'String','Click the close button when you''re done.');

    btn = uicontrol('Parent',d,...
               'Position',[85 20 70 25],...
```

```
                'String','Close',...
                'Callback','delete(gcf)');
end
```

Next, run the `mydialog` function from the Command Window.

```
mydialog
```



Dialog That Returns Output

Use the `uiwait` function to return output based on user selections in the dialog box. For instance, create a program file called `choosedialog.m` to perform these tasks:

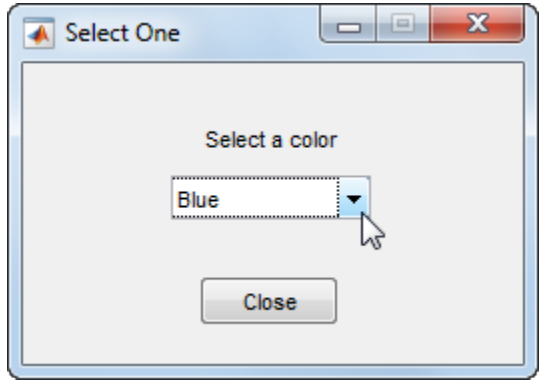
- Call the `dialog` function to create dialog with a specific size, location, and the title, "Select One".
- Call the `uicontrol` function three times to add text, a pop-up menu, and a button, respectively.
- Define the function, `popup_callback`, to serve as the callback function for the button.
- Call the `uiwait` function to wait for the user to close the dialog before returning the output to the command line.

```
function choice = choosedialog

    d = dialog('Position',[300 300 250 150],'Name','Select One');
    txt = uicontrol('Parent',d,...
           'Style','text',...
           'Position',[20 80 210 40],...
           'String','Select a color');
```

```
    popup = uicontrol('Parent',d,...
           'Style','popup',...
           'Position',[75 70 100 25],...
           'String',{'Red';'Green';'Blue'},...
           'Callback',@popup_callback);

    btn = uicontrol('Parent',d,...
           'Position',[89 20 70 25],...
           'String','Close',...
           'Callback','delete(gcf)');

    choice = 'Red';

    % Wait for d to close before running to completion
    uiwait(d);

       function popup_callback(popup,callbackdata)
          idx = popup.Value;
          popup_items = popup.String;
          choice = char(popup_items(idx,:));
       end
end
```

Run the `choosedialog` function from the Command Window. Then, select a color in the dialog.

```
color = choosedialog
```



`choosedialog` returns the last selected color when you close the dialog.

```
color =

Blue
```

Note: The `uiwait` function blocks the MATLAB thread. Although `uiwait` works well in a simple modal dialog, it not recommended for use in more sophisticated applications.

Input Arguments

Name-Value Pair Arguments

Example: `'WindowStyle','normal'` sets the `WindowStyle` property to `'normal'`.

The properties listed here are only a subset. For a complete list, see Figure Properties.

`'Position'` — Location and size of window

`[left bottom width height]`

Location and size of window, specified as the vector, `[left bottom width height]`. This table describes the values in the Position vector. All values are in units specified by the Units property.

Element	Description
`left`	Distance from the left edge of the primary display to the inner left edge of the window. This value can be negative on systems that have more than one monitor.
`bottom`	Distance from the bottom edge of the primary display to the inner bottom edge of the window. This value can be negative on systems that have more than one monitor.
`width`	Distance between the right and left inner edges of the window.
`height`	Distance between the top and bottom inner edges of the window.

Note:

- You cannot specify the Position property when the window is docked.

- To place the full window, including the title bar, menu bar, tool bars, and outer edges, use the `OuterPosition` property.
- On Windows systems, figure windows cannot be less than 104 pixels wide, regardless of the value of the Position property.

Example: `[230 250 570 510]`

Data Types: `double`

`'ButtonDownFcn'` — Button-press callback function

function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

By default, MATLAB sets this command string to be the ButtonDownFcn callback for all dialogs. This command string makes the dialog box close when the user clicks the window.

```
'if isempty(allchild(gcbf)), close(gcbf), end'
```

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

`'WindowStyle'` — Window behavior

`'modal'` (default) | `'normal'` | `'docked'`

Window behavior, specified as one of the following:

- `'modal'` — The window appears on top of all existing windows, making them inaccessible as long as the top window exists and remains modal. However, nothing prevents the display of subsequent windows after the modal window appears.
- `'normal'` — The window is independent of other windows, and the other windows are accessible while the current window exists.

- `'docked'` — The window appears in the desktop or a document window.

Output Arguments

d — Dialog window

figure object

Dialog window, returned as a figure object with these properties values set.

Property	Value
ButtonDownFcn	`'if isempty(allchild(gcbf)), close(gcbf), end'`
Colormap	`[]`
DockControls	`'off'`
HandleVisibility	`'callback'`
IntegerHandle	`'off'`
InvertHardcopy	`'off'`
MenuBar	`'none'`
Number	`[]`
NumberTitle	`'off'`
PaperPositionMode	`'auto'`
Resize	`'off'`
WindowStyle	`'modal'`

See Also

`errordlg` | Figure Properties | `msgbox` | `questdlg` | `waitfor` | `warndlg`

Introduced before R2006a

diary

Save Command Window text to file

Syntax

```
diary
diary('filename')
diary off
diary on
diary filename
```

Description

The `diary` function creates a log of keyboard input and the resulting text output, with some exceptions (see “Tips” on page 1-2172 for details). The output of `diary` is an ASCII file, suitable for searching in, printing, inclusion in most reports and other documents. If you do not specify `filename`, the MATLAB software creates a file named `diary` in the current folder.

`diary` toggles `diary` mode on and off. To see the status of `diary`, type `get(0,'Diary')`. MATLAB returns either `on` or `off` indicating the `diary` status.

`diary('filename')` writes a copy of all subsequent keyboard input and the resulting output (except it does not include graphics) to the named file, where `filename` is the full pathname or `filename` is in the current MATLAB folder. If the file already exists, output is appended to the end of the file. You cannot use a `filename` called `off` or `on`. To see the name of the `diary` file, use `get(0,'DiaryFile')`.

`diary off` suspends the diary.

`diary on` resumes diary mode using the current filename, or the default filename `diary` if none has yet been specified.

`diary filename` is the unquoted form of the syntax.

More About

Tips

Because the output of `diary` is plain text, the file does not exactly mirror input and output from the Command Window:

- Output does not include graphics (figure windows).
- Syntax highlighting and font preferences are not preserved.
- Hidden components of Command Window output such as hyperlink information generated with `matlab:` are shown in plain text. For example, if you enter the following statement

  ```
  str = sprintf('%s%s', ...
     '<a href="matlab:magic(4)">', ...
     'Generate magic square</a>');
  disp(str)
  ```

  MATLAB displays

  Generate magic square

  However, the diary file, when viewed in a text editor, shows

  ```
  str = sprintf('%s%s', ...
     '<a href="matlab:magic(4)">', ...
     'Generate magic square</a>');
  disp(str)
  <a href="matlab:magic(4)">Generate magic square</a>
  ```

  If you view the output of diary in the Command Window, the Command Window interprets the `<a href ...>` statement and displays it as a hyperlink.
- Viewing the output of `diary` in a console window might produce different results compared to viewing `diary` output in the desktop Command Window. One example is using the `\r` option for the `fprintf` function; using the `\n` option might alleviate that problem.
- “Command History”

See Also

`evalc`

Introduced before R2006a

diff

Differences and Approximate Derivatives

Syntax

```
Y = diff(X)
Y = diff(X,n)
Y = diff(X,n,dim)
```

Description

`Y = diff(X)` calculates differences between adjacent elements of `X` along the first array dimension whose size does not equal 1:

- If `X` is a vector of length `m`, then `Y = diff(X)` returns a vector of length `m-1`. The elements of `Y` are the differences between adjacent elements of `X`.

  ```
  Y = [X(2)-X(1) X(3)-X(2) ... X(m)-X(m-1)]
  ```

- If `X` is a nonempty, nonvector p-by-m matrix, then `Y = diff(X)` returns a matrix of size (p-1)-by-m, whose elements are the differences between the rows of `X`.

  ```
  Y = [X(2,:)-X(1,:); X(3,:)-X(2,:); ... X(p,:)-X(p-1,:)]
  ```

- If `X` is a 0-by-0 empty matrix, then `Y = diff(X)` returns a 0-by-0 empty matrix.

`Y = diff(X,n)` calculates the nth difference by applying the `diff(X)` operator recursively `n` times. In practice, this means `diff(X,2)` is the same as `diff(diff(X))`.

`Y = diff(X,n,dim)` is the nth difference calculated along the dimension specified by `dim`. The `dim` input is a positive integer scalar.

Examples

Differences Between Vector Elements

Create a vector, then compute the differences between the elements.

```
X = [1 1 2 3 5 8 13 21];
Y = diff(X)

Y =

     0     1     1     2     3     5     8
```

Note that `Y` has one fewer element than `X`.

Differences Between Matrix Rows

Create a 3-by-3 matrix, then compute the first difference between the rows.

```
X = [1 1 1; 5 5 5; 25 25 25];
Y = diff(X)

Y =

     4     4     4
    20    20    20
```

The output `Y` is a 2-by-3 difference matrix.

Multiple Differences

Create a vector and compute the second-order difference between the elements.

```
X = [0 5 15 30 50 75 105];
Y = diff(X,2)

Y =

     5     5     5     5     5
```

Differences Between Matrix Columns

Create a 3-by-3 matrix, then compute the first-order difference between the columns.

```
X = [1 3 5;7 11 13;17 19 23];
Y = diff(X,1,2)

Y =
```

```
     2     2
     4     2
     2     4
```

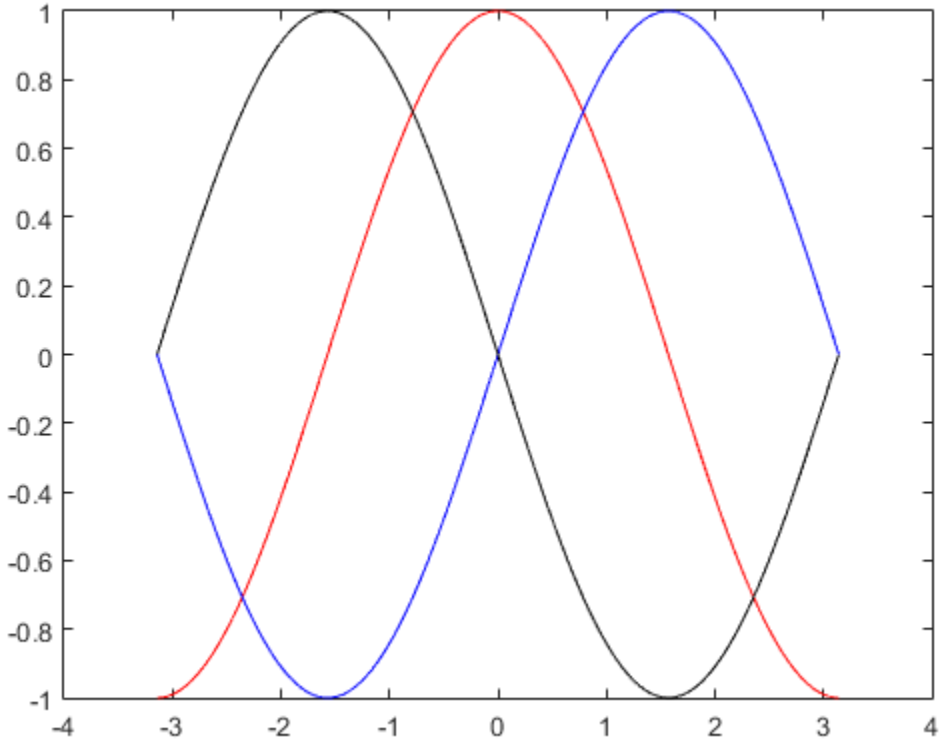
The output `Y` is a 3-by-2 difference matrix.

Approximate Derivatives with diff

Use the `diff` function to approximate partial derivatives with the syntax `Y = diff(f)/h`, where `f` is a vector of function values evaluated over some domain, `X`, and `h` is an appropriate step size.

For example, the first derivative of `sin(x)` with respect to `x` is `cos(x)`, and the second derivative with respect to `x` is `-sin(x)`. You can use `diff` to approximate these derivatives.

```
h = 0.001;       % step size
X = -pi:h:pi;    % domain
f = sin(X);      % range
Y = diff(f)/h;   % first derivative
Z = diff(Y)/h;   % second derivative
plot(X(:,1:length(Y)),Y,'r',X,f,'b', X(:,1:length(Z)),Z,'k')
```



In this plot the blue line corresponds to the original function, `sin`. The red line corresponds to the calculated first derivative, `cos`, and the black line corresponds to the calculated second derivative, `-sin`.

Differences Between Datetime Values

Create a sequence of equally-spaced datetime values, and find the time differences between them.

```
t1 = datetime('now');
t2 = t1 + minutes(5);
t = t1:minutes(1.5):t2
```

```
t =

Columns 1 through 3

   23-Feb-2015 10:07:10   23-Feb-2015 10:08:40   23-Feb-2015 10:10:10

Columns 4 through 4

   23-Feb-2015 10:11:40
```

```
dt = diff(t)
```

```
dt =

   00:01:30   00:01:30   00:01:30
```

`diff` returns a `duration` array.

Input Arguments

`X` — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. `X` can be a numeric array, logical array, datetime array, or duration array.
Complex Number Support: Yes

`n` — Difference order

positive integer scalar | `[]`

Difference order, specified as a positive integer scalar or `[]`. The default value of `n` is 1.

It is possible to specify `n` sufficiently large so that `dim` reduces to a single (`size(X,dim) = 1`) dimension. When this happens, `diff` continues calculating along the next array dimension whose size does not equal 1. This process continues until a 0-by-0 empty matrix is returned.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

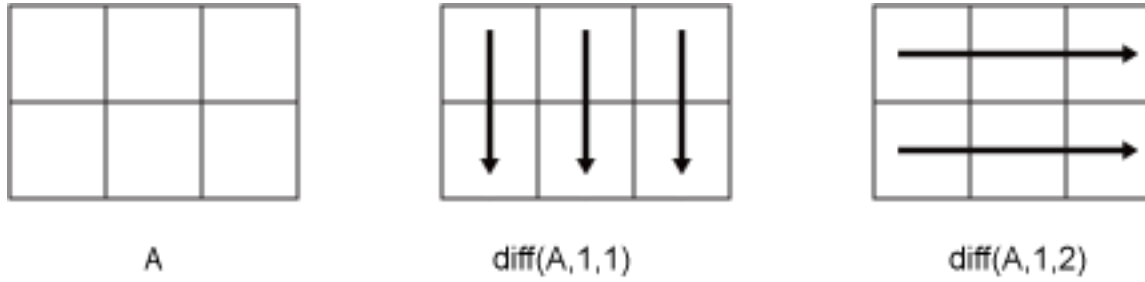
`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional p-by-m input array, `A`:

- `diff(A,1,1)` works on successive elements in the columns of `A` and returns a (p-1)-by-m difference matrix.
- `diff(A,1,2)` works on successive elements in the rows of `A` and returns a p-by-(m-1) difference matrix.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`Y` — Difference array

scalar | vector | matrix | multidimensional array

Difference array, returned as a scalar, vector, matrix, or multidimensional array. If `X` is a nonempty array, then the dimension of `X` acted on by `diff` is reduced in size by `n` in the output.

See Also

`cumsum` | `gradient` | `prod` | `sum`

Introduced before R2006a

diffuse

Calculate diffuse reflectance

Syntax

```
R = diffuse(Nx,Ny,Nz,S)
```

Description

`R = diffuse(Nx,Ny,Nz,S)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` specifies the direction to the light source. You can specify these directions as three vectors`[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

Lambert's Law: `R = cos(PSI)` where `PSI` is the angle between the surface normal and light source.

More About

- "Lighting Overview"

See Also

`specular` | `surfl` | `surfnorm`

dir

List folder contents

Syntax

```
dir
dir name
listing = dir(name)
```

Description

`dir` lists the files and folders in the MATLAB current folder. Results appear in the order returned by the operating system.

`dir name` lists the files and folders that match the string `name`. When name is a folder, `dir` lists the contents of the folder. Specify `name` using absolute or relative path names. You can use wildcards (*).

`listing = dir(name)` returns attributes about `name`.

Input Arguments

`name`

A string value specifying a file or folder name.

Output Arguments

`listing`

Field Name	Description	Class
`name`	File or folder name	char array

Field Name	Description	Class
`date`	Modification date timestamp	char array
`bytes`	Size of the file in bytes	double
`isdir`	`1` if name is a folder; `0` if not	logical
`datenum`	Modification date as serial date number. The value is locale-dependent.	double

Examples

View the Contents of a Folder

View the contents of the `matlab/audiovideo` folder:

```
dir(fullfile(matlabroot, 'toolbox/matlab/audiovideo'))
```

Find Information in the Return Structure

Return the folder listing, restricted to files with a `.m` extension, to the variable `av_files`:

```
av_files = dir(fullfile(matlabroot, ...
               'toolbox/matlab/audiovideo/*.m'))
```

MATLAB returns the information in a structure array:

```
av_files =
25x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

Index into the structure to access a particular item:

```
av_files(3).name
ans =
   audioplayerreg.m
```

– Use the Wildcard Character to Find Multiple Files

View the MAT-files in the current folder that include the term `my_data` by using the wildcard character:

```
dir *my_data*.mat
```

MATLAB returns all file names that match this specification. For instance, it returns the following if they are in the current folder:

```
old_my_data.mat  my_data_final.mat  my_data_test.mat
```

– Exclude Certain Files from the Output

Return the list of files in the current folder, excluding those files that `dir` cannot query:

```
y = dir;
y = y(find(~cellfun(@isempty,{y(:).date})));
```

– Find the Date a File Was Modified

To get the serial date number for the date and time a file was last modified, use the `datenum` field of the structure returned by the `dir` command:

```
DirInfo = dir('startup.m');
filedate = DirInfo.datenum
```

Using the `datenum` function to convert the string returned in the `date` field of the structure is not recommended as this can behave differently in different locales.

```
filedate = datenum(DirInfo.date)
```

Alternatives

Use the Current Folder browser to view the list of files in a folder.

More About

Tips

- To obtain a list of available drives on Microsoft Windows platforms:

  Use the DOS `net use` command in the Command Window:

```
dos('net use')
```

Or

```
[s,r] = dos('net use')
```

MATLAB returns the results to the character array `r`.

- Short DOS file name support

  The MATLAB `dir` function is consistent with the Microsoft Windows operating system `dir` command in that both support short file names generated by DOS.

- Structure Results for Nonexistent Files

  When you run `dir` with an output argument and the results include a nonexistent file or a file that `dir` cannot query for some other reason, then `dir` returns the following default values:

  ```
  date: ''
  bytes: []
  isdir: 0
  datenum: []
  ```

  The most common occurrence is on UNIX platforms when `dir` queries a file that is a symbolic link, which points to a nonexistent target. A nonexistent target is a target that was moved, removed, or renamed. For example, if `my_file` in `my_dir` is a symbolic link to another file that was deleted, then running

  ```
  r = dir('my_dir')
  ```

  includes this result for `my_file`:

  ```
  r(n) =
    name: 'my_file'
    date: ''
    bytes: []
    isdir: 0
    datenum: []
  ```

  where *n* is the index for `my_file`, found by searching `r` by the `name` field.

- "Specify File Names"

See Also

cd | isdir | ls | mkdir | rmdir | what | fileattrib

dir

Class: FTP

View contents of folder on FTP server

Syntax

```
dir(ftpobj)
dir(ftpobj,folder)
details = dir(ftpobj,folder)
```

Description

`dir(ftpobj)` lists the files in current folder on the FTP server associated with ftpobj.

`dir(ftpobj,folder)` lists the files in the specified folder.

`details = dir(ftpobj,folder)` returns the results in a structure array that contains the name, modification date, and size of each file.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`folder`

String enclosed in single quotation marks that specifies the target folder. To specify the folder above the current one, use `'..'`.

Output Arguments

`details`

`m`-by-1 structure array, where `m` is the number of files in the folder. Each element of the structure array contains the following fields:

Field Name	Description	Data Type
`name`	File name	`char`
`date`	Modification date timestamp	`char`
`bytes`	Number of bytes allocated to the file	`double`
`isdir`	If a folder, `isdir = 1`, otherwise `isdir = 0`	`logical`
`datenum`	Modification date as serial date number	`char`

Examples

Connect to the MathWorks FTP server and view the contents:

```
mw=ftp('ftp.mathworks.com');
dir(mw)
```

This code returns:

```
README    incoming  matlab    outgoing  pub       pubs
```

Continuing the previous example, save the folder contents to the structure `m`, close the connection, and view details about the `pub` subfolder:

```
m=dir(mw);
close(mw);

m(5)
```

This code returns:

```
ans =
       name: 'pub'
       date: '13-Aug-2008 00:00:00'
      bytes: 512
      isdir: 1
    datenum: 733633
```

See Also

`mkdir` | `cd` | `ftp` | `rmdir`

Introduced before R2006a

discretize

Group numeric data into bins or categories

Syntax

```
Y = discretize(X,edges)
Y = discretize(X,edges,values)

Y = discretize(X,edges,'categorical')
Y = discretize(X,edges,'categorical',categoryNames)

Y = discretize(___,'IncludedEdge',side)
```

Description

`Y = discretize(X,edges)` returns the indices of the bins that contain the elements of `X`. The `j`th bin contains element `X(i)` if `edges(j) <= X(i) < edges(j+1)` for `1 <= j < N`, where `N` is the number of bins and `length(edges) = N+1`. The last bin contains both edges such that `edges(N) <= X(i) <= edges(N+1)`.

`Y = discretize(X,edges,values)` returns the corresponding element in `values` rather than the bin number. For example, if `X(1)` is in bin 5, then `Y(1)` is `values(5)` rather than `5`. `values` must be a vector with length equal to the number of bins.

`Y = discretize(X,edges,'categorical')` creates a categorical array where each bin is a category. The default category names are of the form "`[A,B)`" (or "`[A,B]`" for the last bin), where `A` and `B` are consecutive values from `edges`.

`Y = discretize(X,edges,'categorical',categoryNames)` names the categories in `Y` using the cell array of strings `categoryNames`. The length of `categoryNames` must be equal to the number of bins.

`Y = discretize(___,'IncludedEdge',side)`, where `side` is `'right'`, specifies that each bin includes the right bin edge, except for the *first* bin which includes both edges. In this case, the `j`th bin contains an element `X(i)` if `edges(j) < X(i) <= edges(j+1)`, where `1 < j <= N` and `N` is the number of bins. The first bin includes the

left edge such that it contains `edges(1) <= X(i) <= edges(2)`. The default for `side` is `'left'`.

Examples

Group Data into Bins

Use `discretize` to group numeric values into discrete bins.

```
X = 1:2:21;
edges = primes(21);
Y = discretize(X,edges)
```

```
Y =

   NaN     2     3     4     4     5     6     6     7     7   NaN
```

Since `1` and `21` fall outside the range of the bins, `Y` contains `NaN` values for those elements.

Assign Bin Values

Use the right edge of each bin as the `values` input. The values of the elements in each bin are always less than the bin's value.

```
X = randi(100,1,10);
edges = 0:25:100;
values = edges(2:end);
Y = discretize(X,edges,values)
```

```
Y =

   100   100    25   100    75    25    50    75   100   100
```

Include Right Edge of Each Bin

Use the `'IncludedEdge'` input to specify that each bin should include its right edge (the first bin includes both edges). Compare the result to the default inclusion of left bin edges.

```
X = 1:2:11;
edges = [1 3 4 7 10 11];
Y = discretize(X,edges,'IncludedEdge','right')


Y =

     1     1     3     3     4     5


Z = discretize(X,edges)


Z =

     1     2     3     4     4     5
```

Group Data into Categorical Array

Group normally distributed data into bins according to the distance from the mean, measured in standard deviations.

```
X = randn(1000,1);
edges = std(X)*(-3:3);
Y = discretize(X,edges, 'categorical', ...
    {'-3sigma', '-2sigma', '-sigma', 'sigma', '2sigma', '3sigma'});
```

`Y` contains undefined categorical values for the elements in `X` that are farther than 3 standard deviations from the mean.

Preview the values in `Y`.

```
Y(1:20)


ans =

     sigma
     2sigma
     -3sigma
     sigma
     sigma
     -2sigma
     -sigma
```

```
sigma
<undefined>
3sigma
-2sigma
<undefined>
sigma
-sigma
sigma
-sigma
-sigma
2sigma
2sigma
2sigma
```

Confirm that approximately 68% of the data falls within one standard deviation of the mean.

```
nnz(Y=='-sigma' | Y=='sigma')/numel(Y)
```

```
ans =

    0.6910
```

Input Arguments

X — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. `X` contains the data to be distributed into bins.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

edges — Bin edges

vector

Bin edges, specified as a nondecreasing, numeric vector. Consecutive elements in `edges` form discrete bins, which `discretize` uses to partition the data in `X`. By default, each bin includes the left bin edge, except for the last bin, which includes both bin edges.

`edges` must have at least two elements, since `edges(1)` is the left edge of the first bin, and `edges(end)` is the right edge of the last bin.

Example: `Y = discretize([1 3 5],[0 2 4 6])` distributes the values 1, 3, and 5 into three bins, which have edges `[0,2)`, `[2,4)`, and `[4,6]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`values` — Bin values

vector

Bin values, specified as a numeric vector. `values` must have the same length as the number of bins, `length(edges)-1`. The elements in `values` replace the normal bin index in the output. That is, if `X(1)` falls into bin 2, then `discretize` returns `Y(1)` as `values(2)` rather than 2.

Example: `Y = discretize(randi(5,10,1),[1 1.5 3 5],diff([1 1.5 3 5]))` returns the widths of the bins, rather than indices ranging from 1 to 3.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

`categoryNames` — Categorical array category names

cell array of strings

Categorical array category names, specified as a cell array of strings. `categoryNames` must have length equal to the number of bins.

Example: `Y = discretize(randi(5,10,1),[1 1.5 3 5],'categorical',{'A' 'B' 'C'})` distributes the data into three categories, `A`, `B`, and `C`.

Data Types: `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Y = discretize(X,edges,'IncludedEdge,'right')`

'IncludedEdge' — Edges to include in each bin

'left' (default) | 'right'

Edges to include in each bin, specified as the comma-separated pair consisting of 'IncludedEdge' and the value 'left' or 'right'.

- If 'right', then each bin includes the right bin edge, except for the first bin, which includes both edges.
- The default value is 'left', so that each bin includes the left bin edge, except for the last bin, which includes both edges.

Example: Y = discretize(randi(11,10,1),1:2:11,'IncludedEdge','right') includes the right bin edge in each bin.

Output Arguments

Y — Bins

vector | matrix | multidimensional array | categorical array

Bins, returned as a numeric vector, matrix, multidimensional array, or categorical array. Y is the same size as X, and each element describes the bin placement for the corresponding element in X.

- Y contains NaN values for out-of-range elements in X (where X(i) < edges(1) or X(i) > edges(end)), or where X contains a NaN.
- If Y is a categorical array, then it contains undefined elements for out-of-range or NaN inputs.
- If values is a vector of an integer data type, then Y contains 0 for out-of-range or NaN inputs.

More About

Tips

- The behavior of discretize is similar to that of the histcounts function. Use histcounts to find the number of elements in each bin. On the other hand, use discretize to find which bin each element belongs to (without counting).

- “Replace Discouraged Instances of hist and histc”

See Also

`categorical` | `histcounts` | `histogram`

Introduced in R2015a

disp

Display value of variable

Syntax

```
disp(X)
```

Description

`disp(X)` displays the value of variable `X` without printing the variable name. Another way to display a variable is to type its name, but this displays a leading "`X =`", which is not always ideal.

If a variable contains an empty array, `disp` returns without displaying anything.

Examples

Display Variable Values

Create a numeric array and a string.

```
A = [15 150];
S = 'Hello World.';
```

Display the value of each variable.

```
disp(A)
```

```
    15   150
```

```
disp(S)
```

```
Hello World.
```

Display Matrix with Column Labels

Display a matrix and label the columns as `Corn`, `Oats`, and `Hay`.

```
X = rand(5,3);
disp('     Corn      Oats      Hay')
disp(X)
```

```
     Corn      Oats      Hay
    0.8147    0.0975    0.1576
    0.9058    0.2785    0.9706
    0.1270    0.5469    0.9572
    0.9134    0.9575    0.4854
    0.6324    0.9649    0.8003
```

Display Hyperlink in Command Window

Display a link to a Web page by including HTML hyperlink code as input to `disp`. For example, display a link to the MathWorks Web site.

```
X = '<a href = "http://www.mathworks.com">MathWorks Web Site</a>';
disp(X)
```

```
MathWorks Web Site
```

Display Multiple Variables on Same Line

Here are three ways to display multiple variable values on the same line in the Command Window.

Concatenate multiple strings together using the `[]` operator. Convert any numeric values to characters using the `num2str` function. Use `disp` to display the string.

```
name = 'Alice';
age = 12;
X = [name,' will be ',num2str(age),' this year.'];
disp(X)
```

```
Alice will be 12 this year.
```

Use `sprintf` to create a string, and then display it with `disp`.

```
name = 'Alice';
age = 12;
X = sprintf('%s will be %d this year.',name,age);
disp(X)
```

```
Alice will be 12 this year.
```

Use `fprintf` to directly display the string without creating a variable. However, to terminate the display properly, you must end the string with the newline (`\n`) metacharacter.

```
name = 'Alice';
age = 12;
fprintf('%s will be %d this year.\n',name,age);
```

```
Alice will be 12 this year.
```

Input Arguments

X — Input array

array

Input array.

To display more than one array, you can use concatenation or the `sprintf` or `fprintf` functions as shown in the example, “Display Multiple Variables on Same Line” on page 1-2197.

See Also

`format` | `fprintf` | `int2str` | `num2str` | `sprintf`

Introduced before R2006a

disp (serial)

Serial port object summary information

Syntax

```
obj
disp(obj)
```

Description

`obj` or `disp(obj)` displays summary information for `obj`, a serial port object or an array of serial port objects.

Examples

The following commands display summary information for the serial port object `s`. on a Windows platform

```
s = serial('COM1')
s.BaudRate = 300
s
```

More About

Tips

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when:

- Creating a serial port object
- Configuring property values using the dot notation

Use the display summary to quickly view the communication settings, communication state information, and information associated with read and write operations.

Introduced before R2006a

display

Display text and numeric expressions

Syntax

```
display(X)
```

Description

`display(X)` prints the value of X. MATLAB implicitly calls `display` after any variable or expression that is not terminated by a semicolon.

To customize the display of objects, overload the `disp` function instead of the `display` function. `display` calls `disp`.

Examples

Display a Matrix

```
X = magic(3);
display(X)
```

```
X =

     8     1     6
     3     5     7
     4     9     2
```

Input Arguments

`X` — Input value
variable | expression

Input value, specified as a variable or expression.

More About

- “Implement disp or disp and display”
- “Overload Functions for Your Class”
- “Relationship Between disp and display”

See Also

`ans` | `disp` | `sprintf`

Introduced before R2006a

dither

Convert image, increasing apparent color resolution by dithering

Syntax

```
X = dither(RGB, map)
X = dither(RGB, map, Qm, Qe)
BW = dither(I)
```

Description

`X = dither(RGB, map)` creates an indexed image approximation of the RGB image in the array `RGB` by dithering the colors in the colormap `map`. The colormap cannot have more than 65,536 colors.

`X = dither(RGB, map, Qm, Qe)` creates an indexed image from `RGB`, where `Qm` specifies the number of quantization bits to use along each color axis for the inverse color map, and `Qe` specifies the number of quantization bits to use for the color space error calculations. If `Qe < Qm`, dithering cannot be performed, and an undithered indexed image is returned in `X`. If you omit these parameters, `dither` uses the default values `Qm = 5`, `Qe = 8`.

`BW = dither(I)` converts the grayscale image in the matrix `I` to the binary (black and white) image `BW` by dithering.

Class Support

`RGB` can be `uint8`, `uint16`, `single`, or `double`. `I` can be `uint8`, `uint16`, `int16`, `single`, or `double`. All other input arguments must be `double`. `BW` is `logical`. `X` is `uint8`, if it is an indexed image with 256 or fewer colors; otherwise, it is `uint16`.

More About

Algorithms

`dither` increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm.

References

[1] Floyd, R. W., and L. Steinberg, "An Adaptive Algorithm for Spatial Gray Scale," *International Symposium Digest of Technical Papers,* Society for Information Displays, 1975, p. 36.

[2] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 469-476.

See Also

`rgb2ind`

divergence

Compute divergence of vector field

Syntax

```
div = divergence(X,Y,Z,U,V,W)
div = divergence(U,V,W)
div = divergence(X,Y,U,V)
div = divergence(U,V)
```

Description

`div = divergence(X,Y,Z,U,V,W)` computes the divergence of a 3-D vector field having vector components `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for the vector components `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements.

`div = divergence(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`div = divergence(X,Y,U,V)` computes the divergence of a 2-D vector field `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`div = divergence(U,V)` assumes `X` and `Y` are determined by the expression

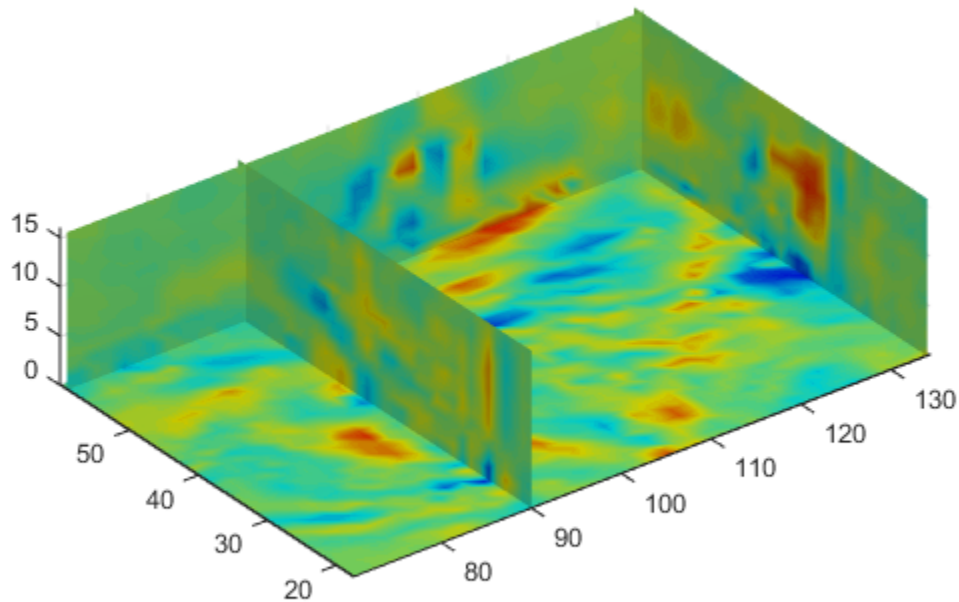
```
[X Y] = meshgrid(1:n,1:m)
```

where `[m,n] = size(U)`.

Examples

This example displays the divergence of vector volume data as slice planes, using color to indicate divergence.

```
load wind
div = divergence(x,y,z,u,v,w);
h = slice(x,y,z,div,[90 134],59,0);
colormap jet
shading interp
daspect([1 1 1])
axis tight
camlight
set([h(1),h(2)],'ambientstrength',.6)
```



More About

- "Overview of Volume Visualization"
- "Displaying Divergence with Stream Tubes"

See Also

`streamtube` | `curl` | `isosurface`

Introduced before R2006a

dlmread

Read ASCII-delimited file of numeric data into matrix

Syntax

```
M = dlmread(filename)
M = dlmread(filename,delimiter)
M = dlmread(filename,delimiter,R1,C1)
M = dlmread(filename,delimiter,[R1 C1 R2 C2])
```

Description

`M = dlmread(filename)` reads an ASCII-delimited numeric data file into matrix `M`. The `dlmread` function detects the delimiter from the file and treats repeated white spaces as a single delimiter.

`M = dlmread(filename,delimiter)` reads data from the file using the specified delimiter and treats repeated delimiter characters as separate delimiters.

`M = dlmread(filename,delimiter,R1,C1)` starts reading at row offset `R1` and column offset `C1`. For example, the offsets `R1=0`, `C1=0` specify the first value in the file.

To specify row and column offsets without specifying a delimiter, use an empty string as a placeholder, for example, `M = dlmread(filename,'',2,1)`.

`M = dlmread(filename,delimiter,[R1 C1 R2 C2])` reads only the range bounded by row offsets `R1` and `R2` and column offsets `C1` and `C2`. Another way to define the range is to use spreadsheet notation, such as `'A1..B7'` instead of `[0 0 6 1]`.

Examples

Read Entire Delimited File

Read the sample file, `count.dat`.

```
M = dlmread('count.dat')
```

```
M =

    11    11     9
     7    13    11
    14    17    20
    11    13     9
    43    51    69
    38    46    76
    61   132   186
    75   135   180
    38    88   115
    28    36    55
    12    12    14
    18    27    30
    18    19    29
    17    15    18
    19    36    48
    32    47    10
    42    65    92
    57    66   151
    44    55    90
   114   145   257
    35    58    68
    11    12    15
    13     9    15
    10     9     7
```

`dlmread` detects the delimiter from the file and returns a matrix.

Read File Containing Empty Delimited Fields

Write two matrices to a file, and then read the entire file using `dlmread`.

Export a matrix to a file named `myfile.txt`. Then, append an additional matrix to the file that is offset one row below the first.

```
X = magic(3);
dlmwrite('myfile.txt',[X*5 X/5],' ')
dlmwrite('myfile.txt',X,'-append', ...
   'roffset',1,'delimiter',' ')
```

View the file contents.

```
type myfile.txt
```

```
40 5 30 1.6 0.2 1.2
15 25 35 0.6 1 1.4
20 45 10 0.8 1.8 0.4

8 1 6
3 5 7
4 9 2
```

Read the entire file using `dlmread`.

```
M = dlmread('myfile.txt')
```

```
M =

   40.0000    5.0000   30.0000    1.6000    0.2000    1.2000
   15.0000   25.0000   35.0000    0.6000    1.0000    1.4000
   20.0000   45.0000   10.0000    0.8000    1.8000    0.4000
    8.0000    1.0000    6.0000         0         0         0
    3.0000    5.0000    7.0000         0         0         0
    4.0000    9.0000    2.0000         0         0         0
```

When `dlmread` imports a file containing nonrectangular data, it fills empty fields with zeros.

Read Delimited File Starting At Specific Row and Column Offset

Create a file named `dlmlist.txt` that contains column headers and space-delimited values.

```
test max min direction
10 27.7 12.4 12
11 26.9 13.5 18
12 27.4 16.9 31
13 25.1 12.7 29
```

Read the numeric values in the file. Specify a space delimiter, a row offset of 1, and a column offset of 0.

```
filename = 'dlmlist.txt';
M = dlmread(filename,' ',1,0)
```

```
M =

   10.0000   27.7000   12.4000   12.0000
   11.0000   26.9000   13.5000   18.0000
   12.0000   27.4000   16.9000   31.0000
   13.0000   25.1000   12.7000   29.0000
```

Read Specific Range from Delimited File

Create a file named `dlmlist.txt` that contains column headers and space-delimited values.

```
test max min direction
10 27.7 12.4 12
11 26.9 13.5 18
12 27.4 16.9 31
13 25.1 12.7 29
```

Read only the last two rows of numeric data from the file.

```
M = dlmread('dlmlist.txt',' ',[3 0 4 3])
```

```
M =

   12.0000   27.4000   16.9000   31.0000
   13.0000   25.1000   12.7000   29.0000
```

Input Arguments

`filename` — File name

string

File name, specified as a string.

Example: `'myFile.dat'`

Data Types: `char`

`delimiter` — Field delimiter character

string

Field delimiter character, specified as a string. Use `'\t'` to specify a tab delimiter.

Example: `','`

Example: ' '

R1 — Starting row offset

0 (default) | nonnegative integer

Starting row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

C1 — Starting column offset

0 (default) | nonnegative integer

Starting column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

R2 — Ending row offset

nonnegative integer

Ending row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

C2 — Ending column offset

nonnegative integer

Ending column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

More About

Tips

- Skip header rows or columns by specifying row and column offsets. All values in the file other than headers must be numeric.

Algorithms

`dlmread` fills empty delimited fields with zero. When the `dlmread` function reads data files with lines that end with a nonspace delimiter, such as a semicolon, it returns a matrix, `M`, that has an additional last column of zeros.

`dlmread` imports any complex number as a whole into a complex numeric field. This table shows valid forms for a complex number.

Form	Example
`±<real>±<imag>i|j`	`5.7-3.1i`
`±<imag>i|j`	`-7j`

Embedded white space in a complex number is invalid and `dlmread` regards it as a field delimiter.

- "Ways to Import Text Files"

See Also

`dlmwrite` | `readtable` | `textscan` | `uiimport`

Introduced before R2006a

dlmwrite

Write matrix to ASCII-delimited file

Syntax

```
dlmwrite(filename,M)
dlmwrite(filename,M,'-append')

dlmwrite( ___ ,Name,Value)

dlmwrite(filename,M,delimiter)
dlmwrite(filename,M,delimiter,row,col)
```

Description

`dlmwrite(filename,M)` writes numeric data in array `M` to an ASCII format file, `filename`, using the default delimiter (`,`) to separate array elements. If the file, `filename`, already exists, `dlmwrite` overwrites the file.

`dlmwrite(filename,M,'-append')` appends the data to the end of the existing file, `filename`.

`dlmwrite( ___ ,Name,Value)` additionally specifies delimiter, newline character, offset, and precision options using one or more name-value pair arguments.

`dlmwrite(filename,M,delimiter)` writes array `M` to the file, `filename`, using the specified delimiter, `delimiter`, to separate array elements.

`dlmwrite(filename,M,delimiter,row,col)` writes the array starting at the specified row and column `row` and `col`, in the destination file. Empty elements separated by `delimiter` fill the leading rows and columns.

Examples

Write Comma-Separated Data

Create an array of sample data, `M`.

```
M = magic(3);
```

Write matrix `M` to a file, `'myFile.txt'`, using the default delimiter (`,`).

```
dlmwrite('myFile.txt',M)
```

View the data in the file.

```
type('myFile.txt')
```

```
8,1,6
3,5,7
4,9,2
```

Write Tab-Delimited Data and Specify Precision

Create an array of sample data, `M`.

```
M = magic(3)*pi
```

```
M =

   25.1327    3.1416   18.8496
    9.4248   15.7080   21.9911
   12.5664   28.2743    6.2832
```

Write matrix `M` to a file, `'myFile.txt'`, delimited by the tab character and using a precision of 3 significant digits.

```
dlmwrite('myFile.txt',M,'delimiter','\t','precision',3)
```

View the data in the file.

```
type('myFile.txt')
```

```
25.1 3.14 18.8
9.42 15.7 22
12.6 28.3 6.28
```

Write and Append Data to File

Create two arrays of sample numeric data.

```
M = magic(5);
N = magic(3);
```

Export matrix `M` to a file and use whitespace as the delimiter.

```
dlmwrite('myFile.txt',M,'delimiter',' ');
```

Append matrix `N` to the file, offset from the existing data by one row. Then, view the file.

```
dlmwrite('myFile.txt',N,'-append',...
'delimiter',' ','roffset',1)
type('myFile.txt')
```

```
17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
11 18 25 2 9

8 1 6
3 5 7
4 9 2
```

Read the data in `'myFile.txt'` using `dlmread`.

```
dlmread('myFile.txt')
```

```
ans =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
     8     1     6     0     0
     3     5     7     0     0
     4     9     2     0     0
```

When `dlmread` reads the two matrices from the file, it pads the smaller matrix with zeros.

Write Data and Specify Precision As String

Create an array of sample numeric data.

```
M = magic(3);
```

Export matrix `M` to a file using a precision of 6 decimal places.

```
dlmwrite('myFile.txt',M,'precision','%.6f');
```

View the data in the file.

```
type('myFile.txt')
```

```
8.000000,1.000000,6.000000
3.000000,5.000000,7.000000
4.000000,9.000000,2.000000
```

Input Arguments

`filename` — Name of file to write

string

Name of file to write, specified as a string.

Example: `'myFile.txt'`

Data Types: `char`

`M` — Numeric data to write

matrix | cell array of numeric values

Numeric data to write, specified as a matrix or a cell array of numeric values with one value per cell.

Example: `[1,2,3;4,5,6]`

Example: `{1,2,3;4,5,6}`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `cell`
Complex Number Support: Yes

`delimiter` — Delimiter to separate array elements

`','` (default) | string

Delimiter to separate array elements, specified as a string containing a single character or control character. Use `'\t'` to produce tab-delimited files.

Example: `';'`

Example: `'\t'`

Data Types: `char`

`row` — Row offset

0 (default) | scalar

Row offset, specified as a scalar. The row offset indicates the number of rows to skip before writing the numeric data. `row` is zero-based, so that `row = 0` instructs MATLAB to begin writing in the first row of the destination file. Skipped rows are populated with the specified delimiter.

`col` — Column offset

0 (default) | scalar

Column offset, specified as a scalar. The column offset indicates the number of columns to skip before writing the numeric data. `col` is zero-based, so that `col = 0` instructs MATLAB to begin writing in the first column of the destination file. Skipped columns are separated with the specified delimiter.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `dlmwrite('myFile.txt',M,'precision',4,'delimiter',' ')` writes the numeric values in array `M` with four significant digits and delimited using the whitespace character.

`'delimiter'` — Delimiter to separate array elements

`','` (default) | string

Delimiter to separate array elements, specified as the comma-separated pair consisting of `'delimiter'` and string containing a single character or control character. Use `'\t'` to produce tab-delimited files.

Example: `'delimiter',';'`

Example: `'delimiter','\t'`

Data Types: `char`

`'roffset'` — Row offset

0 (default) | scalar

Row offset, specified as the comma-separated pair consisting of `'roffset'` and a scalar. The row offset indicates the number of rows to skip before writing the numeric data.

These rows are populated with the specified delimiter. When appending to an existing file, the new data is offset from the end of the existing data.

The row offset is zero-based, so that `'roffset',0` instructs MATLAB to begin writing in the first row of the destination file, which is the default. However, when appending to a file, `'roffset',0` instructs MATLAB to begin writing in the first row immediately following existing data.

Example: `'roffset',2`

`'coffset'` — Column offset

0 (default) | scalar

Column offset from the left side of the destination file, specified as the comma-separated pair consisting of `'coffset'` and a scalar. The column offset indicates the number of columns to skip before writing the numeric data. These columns are separated with the specified delimiter.

The column offset is zero-based, so that `'coffset',0` instructs MATLAB to begin writing in the first column of the destination file, which is the default.

Example: `'coffset',1`

`'precision'` — Numeric precision

5 (default) | scalar | C-style format string

Numeric precision to use in writing data to the file, specified as the comma-separated pair consisting of `'precision'` and a scalar or a C-style format string that begins with `%`, such as `'%10.5f'`. If the value of `precision` is a scalar, then it indicates the number of significant digits.

Example: `'precision',3`

Example: `'precision','%10.5f'`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

`'newline'` — Line terminator

`'pc'` | `'unix'`

Line terminator, specified as the comma-separated pair consisting of `'newline'` and either `'pc'` to use a carriage return/line feed (`CR/LF`), or `'unix'` to use a line feed (`LF`).

Example: `'newline','pc'`

More About

Tips

- `dlmwrite` writes a file that spreadsheet programs can read. Alternatively, if your system has Excel for Windows installed, you can create a spreadsheet using `xlswrite`.

See Also

`dlmread` | `writetable` | `xlswrite`

Introduced before R2006a

dmperm

Dulmage-Mendelsohn decomposition

Syntax

```
p = dmperm(A)
[p,q,r,s,cc,rr] = dmperm(A)
```

Description

`p = dmperm(A)` finds a vector `p` such that `p(j) = i` if column `j` is matched to row `i`, or zero if column `j` is unmatched. If `A` is a square matrix with full structural rank, `p` is a maximum matching row permutation and `A(p,:)` has a zero-free diagonal. The structural rank of `A` is `sprank(A) = sum(p>0)`.

`[p,q,r,s,cc,rr] = dmperm(A)` where `A` need not be square or full structural rank, finds the Dulmage-Mendelsohn decomposition of `A`. `p` and `q` are row and column permutation vectors, respectively, such that `A(p,q)` has a block upper triangular form. `r` and `s` are index vectors indicating the block boundaries for the fine decomposition. `cc` and `rr` are vectors of length five indicating the block boundaries of the coarse decomposition.

`C = A(p,q)` is split into a 4-by-4 set of coarse blocks:

```
A11 A12 A13 A14
0    0  A23 A24
0    0   0  A34
0    0   0  A44
```

where `A12`, `A23`, and `A34` are square with zero-free diagonals. The columns of `A11` are the unmatched columns, and the rows of `A44` are the unmatched rows. Any of these blocks can be empty. In the coarse decomposition, the `(i,j)th` block is `C(rr(i):rr(i+1)-1,cc(j):cc(j+1)-1)`. For a linear system,

- `[A11 A12]` is the underdetermined part of the system—it is always rectangular and with more columns and rows, or 0-by-0,
- `A23` is the well-determined part of the system—it is always square, and

- `[A34 ; A44]` is the overdetermined part of the system—it is always rectangular with more rows than columns, or `0`-by-`0`.

The structural rank of `A` is `sprank(A) = rr(4)-1`, which is an upper bound on the numerical rank of `A`. `sprank(A) = rank(full(sprand(A)))` with probability 1 in exact arithmetic.

The `A23` submatrix is further subdivided into block upper triangular form via the fine decomposition (the strongly connected components of `A23`). If `A` is square and structurally nonsingular, `A23` is the entire matrix.

`C(r(i):r(i+1)-1,s(j):s(j+1)-1)` is the `(i,j)`th block of the fine decomposition. The `(1,1)` block is the rectangular block `[A11 A12]`, unless this block is `0`-by-`0`. The `(b,b)` block is the rectangular block `[A34 ; A44]`, unless this block is `0`-by-`0`, where `b = length(r)-1`. All other blocks of the form `C(r(i):r(i+1)-1,s(i):s(i+1)-1)` are diagonal blocks of `A23`, and are square with a zero-free diagonal.

More About

Tips

If `A` is a reducible matrix, the linear system $Ax = b$ can be solved by permuting `A` to a block upper triangular form, with irreducible diagonal blocks, and then performing block backsubstitution. Only the diagonal blocks of the permuted matrix need to be factored, saving fill and arithmetic in the blocks above the diagonal.

In graph theoretic terms, `dmperm` finds a maximum-size matching in the bipartite graph of `A`, and the diagonal blocks of `A(p,q)` correspond to the strong Hall components of that graph. The output of `dmperm` can also be used to find the connected or strongly connected components of an undirected or directed graph. For more information see Pothen and Fan [1].

References

[1] Pothen, Alex and Chin-Ju Fan "Computing the Block Triangular Form of a Sparse Matrix" *ACM Transactions on Mathematical Software* Vol 16, No. 4 Dec. 1990, pp. 303-324.

See Also

sprank

Introduced before R2006a

doc

Reference page in Help browser

Syntax

```
doc
doc name
```

Description

`doc` opens the Help browser. If the Help browser is already open, but not visible, then `doc` brings it to the foreground and opens a new tab.

`doc name` displays documentation for the functionality specified by `name`, such as a function, class, or block.

- If there is a MathWorks reference page corresponding to `name`, then `doc` displays the page in the Help browser. The `doc` command does not display third-party or custom HTML documentation.
- If there is no reference page corresponding to `name`, then `doc` searches for help text in a file named *`name`*`.m`. When help text is available, `doc` displays it in the Help browser.
- If there is no reference page and no help text associated with `name`, then `doc` searches the documentation for `name` and displays the search results in the Help browser.

Examples

Function Reference Pages

Display the reference page for the `abs` function.

```
doc abs
```

Several products include different versions of `abs`. If your Help preferences support displaying documentation for those products, then the Help browser displays the

MATLAB `abs` reference page and a message with links to other versions of `abs`. This message appears at the top of the page.

Class and Method Reference Pages

Display the reference page for the `handle` class.

```
doc handle
```

Display the reference page for the `findobj` method in the `handle` class.

```
doc handle.findobj
```

Display the reference page for the `Map` class in the `containers` package.

```
doc containers.Map
```

Custom Class Pages

Display formatted help text for a custom class.

MATLAB includes a set of example files that show how to create a class, including a class file named `sads.m`. Add the example folder to the path, and request documentation for `sads`.

```
addpath(...
   fullfile(matlabroot,'help','techdoc','matlab_env',...
   'examples'))
doc sads
```

Display the help for the `steer` method of the `sads` class.

```
doc sads.steer
```

Because the help text follows MATLAB conventions, MATLAB formats the display in the browser.

Input Arguments

`name` — Name of function, class, block, or other functionality

string

Name of a function, class, block, or other functionality, specified as a string. Alternatively, an operator symbol.

Some classes and other packaged items require that you specify the package name. Events, properties, and some methods require that you specify the class name. Separate the components of the name with periods, such as:

```
doc className.name
doc packageName.name
doc packageName.className.name
```

Methods for some classes are not accessible using the `doc` command; instead, use links on the class reference page.

More About

Tips

- To access third-party or custom documentation, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.
- "Ways to Get Function Help"
- "Add Help for Your Program"
- " Display Custom Documentation"

See Also

`help` | `web`

Introduced before R2006a

docsearch

Help browser search

Syntax

```
docsearch
docsearch expression
```

Description

`docsearch` opens the Help browser and displays the documentation home page. If the Help browser is already open, but not visible, then `docsearch` brings it to the foreground.

`docsearch expression` searches MathWorks documentation for pages with words that match the specified expression and highlights them. To clear highlighting, press the **Esc** key. The `docsearch` command does not search third-party or custom documentation.

Examples

Single Words

Find all documentation pages that contain the word *plot*.

```
docsearch plot
```

Multiple Words

Find documentation pages with the words *plot* and *tools*.

```
docsearch plot tools
```

Expand the search to include variations of the word *plot*, such as *plotting* or *plots*, using a wildcard character.

```
docsearch plot* tools
```

Narrow the search to pages that include an exact phrase by enclosing the phrase in quotation marks.

```
docsearch "plot tools"
```

Find pages with either word, but not necessarily both words, using the `OR` operator.

```
docsearch plot OR tools
```

Input Arguments

`expression` — Expression that defines search terms

string

Expression that defines search terms, specified as a string. Expressions can include:

- Quotation marks to specify exact phrases, such as `"plot tools"`.
- Boolean operator keywords in uppercase (listed here in order of precedence): `NOT`, `OR`, `AND`.
- Asterisk (`*`) wildcard characters, except at the beginning of a word or in an exact phrase. Searches require that at least two characters in the expression are *not* wildcard characters.

More About

Tips

- To access third-party or custom documentation, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.
- “Search Syntax and Tips”

See Also

`builddocsearchdb` | `doc`

Introduced before R2006a

dos

Execute DOS command and return output

Syntax

```
status = dos(command)
[status,cmdout] = dos(command)
[status,cmdout] = dos(command,'-echo')
```

Description

`status = dos(command)` executes the specified DOS command for Windows platforms, and waits for the command to finish execution before returning the exit status to the status variable.

`[status,cmdout] = dos(command)` additionally returns the output of the DOS command to cmdout. This syntax is most useful for DOS console commands that do not require user input, such as `dir`.

`[status,cmdout] = dos(command,'-echo')` additionally displays (echoes) the command output in the MATLAB Command Window. This syntax is most useful for DOS console commands that require user input and that run correctly in the MATLAB Command Window, such as `comp`.

Examples

Save DOS Command Exit Status

Issue a DOS command to create a new folder named `mynew` and save the exit status to a variable.

```
command = 'mkdir mynew';
status = dos(command)
```

```
status =
```

```
     0
```

The `status` of zero indicates that the `mynew` folder was created successfully.

Open and Run a Windows UI Command

Open Notepad and immediately return the exit status to MATLAB by appending an ampersand (`&`) to the `notepad` command.

```
status = dos('notepad &')
```

```
status =

     0
```

The `status` of zero indicates that Notepad successfully started.

Save Successful DOS Command Status and Output

Execute the DOS command, `dir`, and view the exit status and command output.

```
[status,cmdout] = dos('dir');
status, cmdout
```

```
status =

     0

cmdout =

 Volume in drive C is OSDisk
 Volume Serial Number is XXX-XXXX

 Directory of C:\my_MATLAB_files

04/10/2012  12:08 PM <DIR>          .
04/10/2012  12:08 PM <DIR>          ..
04/21/2011  09:24 AM               171 base.mat
02/08/2010  05:14 PM                73 baseball.dat
04/10/2012  12:08 PM               474 collatz.asv
04/10/2012  11:56 AM               480 collatz.m
.
.
.
```

When you issue a valid DOS command, `status` indicates success and `cmdout` contains the command output.

Save Unsuccessful DOS Command Status and Output

Attempt to execute a command called `foo`. Then, view the `status` and `results` output arguments.

```
[status,results] = dos('foo');
status, results
```

```
status =

     1


results =

'foo' is not recognized as an internal or external command,
operable program or batch file.
```

When you issue an invalid DOS command, `status` indicates failure and `results` contains the DOS error message.

Display DOS Command Output in MATLAB Command Window

Display command output and prompts in the Command Window as the command executes, and also assign the command output to the `results` variable.

```
[status,results] = dos('comp', '-echo');
```

```
Name of first file to compare: collatz.m
collatz.m
Name of second file to compare: collatz.asv
collatz.asv
Option: /A
/A
Option:

Comparing collatz.m and collatz.asv...
Files compare OK

Compare more files (Y/N) ? N
N
```

```
>>
```

Input Arguments

`command` — MS-DOS® command

string

MS-DOS command, specified as a string. The command can be a Windows UI program that opens a graphical user interface, or a DOS console command that you typically run in a DOS command window. The command executes in a DOS shell, which might not be the shell from which you launched MATLAB.

Example: `'dir'`

Output Arguments

`status` — Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (`&`), then `status` is the exit status upon command launch.
- If `command` does not include the ampersand character (`&`), then `status` is the exit status upon command completion.

`cmdout` — Output of operating system command

string

Output of the operating system command, returned as a string.

Limitations

- DOS does not support UNC path names. Therefore, if the current folder uses a UNC path name, then running `dos` with a DOS `command` that relies on the current folder fails. MATLAB returns this error:

```
Error using dos
DOS commands may not be executed when the current directory is a UNC pathname
```

To work around this limitation, change the folder to a mapped drive before running `dos` or a function that calls `dos`.

More About

Tips

- To execute the operating system command in the background, include the trailing character, `&`, in the `command` argument (for example, `'notepad &'`). The exit status is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

  Note: If `command` includes the trailing `&` character, then `cmdout` is empty.

See Also

`!` (exclamation point) | `computer` | `perl` | `system` | `unix`

Introduced before R2006a

dot

Dot product

Syntax

```
C = dot(A,B)
C = dot(A,B,dim)
```

Description

`C = dot(A,B)` returns the scalar dot product of `A` and `B`.

- If `A` and `B` are vectors, then they must have the same length.
- If `A` and `B` are matrices or multidimensional arrays, then they must have the same size. In this case, the `dot` function treats `A` and `B` as collections of vectors. The function calculates the dot product of corresponding vectors along the first array dimension whose size does not equal 1.

`C = dot(A,B,dim)` evaluates the dot product of `A` and `B` along dimension, `dim`. The `dim` input is a positive integer scalar.

Examples

Dot Product of Real Vectors

Create two simple, three-element vectors.

```
A = [4 -1 2];
B = [2 -2 -1];
```

Calculate the dot product of `A` and `B`.

```
C = dot(A,B)
```

```
C =
```

```
     8
```

The result is `8` since

$$C = \mathrm{A}(1) * \mathrm{B}(1) + \mathrm{A}(2) * \mathrm{B}(2) + \mathrm{A}(3) * \mathrm{B}(3)$$
$$C = 8 + 2 - 2 .$$

Dot Product of Complex Vectors

Create two complex vectors.

```
A = [1+i 1-i -1+i -1-i];
B = [3-4i 6-2i 1+2i 4+3i];
```

Calculate the dot product of `A` and `B`.

```
C = dot(A,B)
```

```
C =

   1.0000 - 5.0000i
```

The result is a complex scalar since `A` and `B` are complex. In general, the dot product of two complex vectors is also complex. An exception is when you take the dot product of a complex vector with itself.

Find the inner product of `A` with itself.

```
D = dot(A,A)
```

```
D =

     8
```

The result is a real scalar. The inner product of a vector with itself is related to the Euclidean length of the vector, `norm(A)`.

Dot Product of Matrices

Create two matrices.

```
A = [1 2 3;4 5 6;7 8 9];
```

```
B = [9 8 7;6 5 4;3 2 1];
```

Find the dot product of A and B.

```
C = dot(A,B)
```

```
C =

    54    57    54
```

The result, C, contains three separate dot products. dot treats the columns of A and B as vectors and calculates the dot product of corresponding columns. So, for example, C(1) = 54 is the dot product of A(:,1) with B(:,1).

Find the dot product of A and B, treating the *rows* as vectors.

```
D = dot(A,B,2)
```

```
D =

    46
    73
    46
```

In this case, D(1) = 46 is the dot product of A(1,:) with B(1,:).

Dot Product of Multidimensional Arrays

Create two multidimensional arrays.

```
A = cat(3,[1 1;1 1],[2 3;4 5],[6 7;8 9])
B = cat(3,[2 2;2 2],[10 11;12 13],[14 15; 16 17])
```

```
A(:,:,1) =

     1     1
     1     1


A(:,:,2) =

     2     3
     4     5
```

```
A(:,:,3) =

     6     7
     8     9


B(:,:,1) =

     2     2
     2     2


B(:,:,2) =

    10    11
    12    13


B(:,:,3) =

    14    15
    16    17
```

Calculate the dot product of `A` and `B` along the third dimension (`dim = 3`).

```
C = dot(A,B,3)
```

```
C =

   106   140
   178   220
```

The result, `C`, contains four separate dot products. The first dot product, `C(1,1) = 106`, is equal to the dot product of `A(1,1,:)` with `B(1,1,:)`.

Input Arguments

`A,B` — Input arrays

numeric arrays

Input arrays, specified as numeric arrays.

Data Types: `single | double`

Complex Number Support: Yes

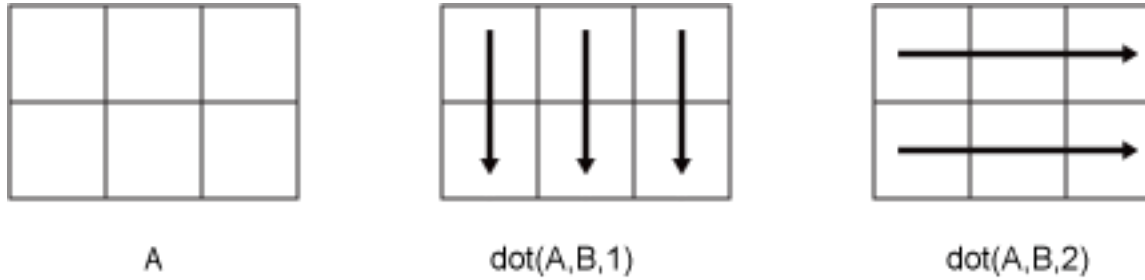
`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first array dimension whose size does not equal 1.

Consider two 2-D input arrays, `A` and `B`:

- `dot(A,B,1)` treats the columns of `A` and `B` as vectors and returns the dot products of corresponding columns.
- `dot(A,B,2)` treats the rows of `A` and `B` as vectors and returns the dot products of corresponding rows.



`dot` returns `conj(A).*B` if `dim` is greater than `ndims(A)`.

More About

Scalar Dot Product

The scalar dot product of two real vectors of length n is equal to

$$u \bullet v = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n .$$

This relation is commutative for real vectors, such that `dot(u,v)` equals `dot(v,u)`. If the dot product is equal to zero, then u and v are perpendicular.

For complex vectors, the dot product involves a complex conjugate. This ensures that the inner product of any vector with itself is real and positive definite.

$$u \bullet v = \sum_{i=1}^{n} \overline{u}_i v_i .$$

Unlike the relation for real vectors, the complex relation is not commutative, so `dot(u,v)` equals `conj(dot(v,u))`.

Algorithms

- When inputs `A` and `B` are real or complex vectors, the `dot` function treats them as column vectors and `dot(A,B)` is the same as `sum(conj(A).*B)`.
- When the inputs are matrices or multidimensional arrays, the `dim` argument determines which dimension the `sum` function operates on. In this case, `dot(A,B)` is the same as `sum(conj(A).*B,dim)`.

See Also

`conj` | `cross` | `sum`

Introduced before R2006a

double

Convert to double precision

Syntax

```
double(x)
```

Description

`double(x)` returns the double-precision value for `X`. If `X` is already a double-precision array, `double` has no effect.

More About

Tips

The `double` function can be overloaded for any object when it makes sense to convert it to a double-precision value.

- "Floating-Point Numbers"

See Also

`cast` | `single` | `str2double` | `typecast`

Introduced before R2006a

dragrect

Drag rectangles with mouse

Syntax

```
[finalrect] = dragrect(initialrect)
[finalrect] = dragrect(initialrect,stepsize)
```

Description

`[finalrect] = dragrect(initialrect)` tracks one or more rectangles anywhere on the screen. The n-by-4 matrix `initialrect` defines the rectangles. Each row of `initialrect` must contain the initial rectangle position as `[left bottom width height]` values. `dragrect` returns the final position of the rectangles in `finalrect`.

`[finalrect] = dragrect(initialrect,stepsize)` moves the rectangles in increments of `stepsize`. The lower left corner of the first rectangle is constrained to a grid of size equal to `stepsize` starting at the lower left corner of the figure, and all other rectangles maintain their original offset from the first rectangle.

`[finalrect] = dragrect(...)` returns the final positions of the rectangles when the mouse button is released. The default step size is `1`.

Examples

Drag a rectangle that is 50 pixels wide and 100 pixels in height.

```
waitforbuttonpress
point1 = get(gcf,'CurrentPoint') % button down detected
rect = [point1(1,1) point1(1,2) 50 100]
[r2] = dragrect(rect)
```

More About

Tips

`dragrect` returns immediately if a mouse button is not currently pressed. Use `dragrect` in a `ButtonDownFcn`, or from the command line in conjunction with `waitforbuttonpress`, to ensure that the mouse button is down when `dragrect` is called. `dragrect` returns when you release the mouse button.

If the drag ends over a figure window, the positions of the rectangles are returned in that figure's coordinate system. If the drag ends over a part of the screen not contained within a figure window, the rectangles are returned in the coordinate system of the figure over which the drag began.

Note: You cannot use normalized figure units with `dragrect`.

See Also

`rbbox` | `waitforbuttonpress`

Introduced before R2006a

drawnow

Update figures and process callbacks

Syntax

```
drawnow
drawnow limitrate
drawnow nocallbacks
drawnow limitrate nocallbacks

drawnow update
drawnow expose
```

Description

`drawnow` updates figures and processes any pending callbacks. Use this command if you modify graphics objects and want to see the updates on the screen immediately.

`drawnow limitrate` limits the number of updates to 20 frames per second. If it has been fewer than 50 milliseconds since the last update, or if the graphics renderer is busy with the previous change, then `drawnow` discards the new updates. Use this command if you are updating graphics objects in a loop and do not need to see every update on the screen. Skipping updates can create faster animations. Pending callbacks are processed, so you can interact with figures during animations.

`drawnow nocallbacks` defers callbacks, such as mouse clicks, until the next full `drawnow` command. Use this option if you want to prevent callbacks from interrupting your code. For more information, see “Actions Equivalent to drawnow” on page 1-2248.

`drawnow limitrate nocallbacks` limits the number of updates to 20 frames per second, skips updates if the renderer is busy, and defers callbacks.

`drawnow update` skips updates if the renderer is busy and defers callbacks.

Note: This syntax is not recommended. Use the `limitrate` option instead.

`drawnow expose` updates figures, but defers callbacks.

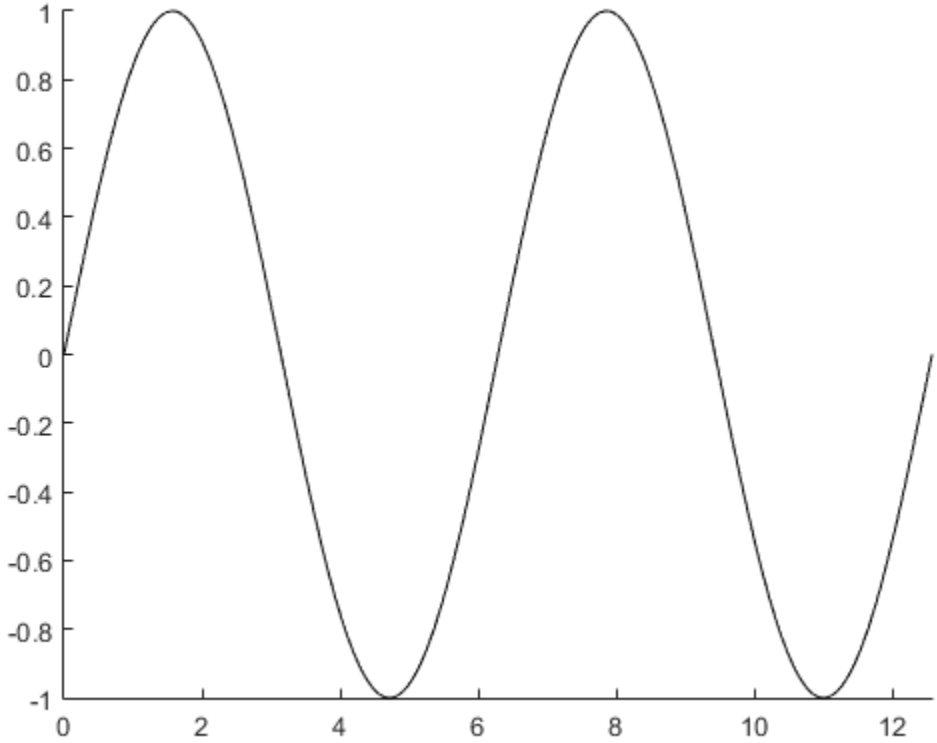
Note: This syntax is not recommended. Use the `nocallbacks` option instead.

Examples

Create Animation of Streaming Data

Create an animation of a line growing as it accumulates 2,000 data points. Use `drawnow` to display the changes on the screen after each iteration through the loop.

```
h = animatedline;
axis([0 4*pi -1 1])
x = linspace(0,4*pi,2000);

for k = 1:length(x)
    y = sin(x(k));
    addpoints(h,x(k),y);
    drawnow
end
```

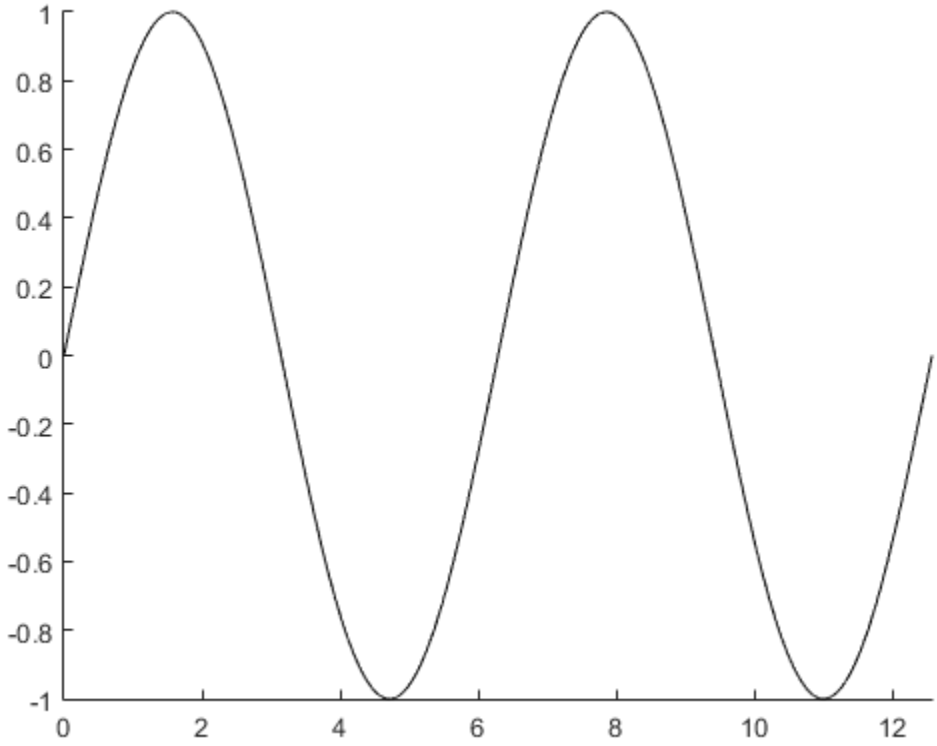


Skip Updates for Faster Animation

Create an animation of a line growing as it accumulates 10,000 points. Since there are 10,000 points, drawing each update on the screen is slow. Create a faster, smooth animation by limiting the number of updates using `drawnow limitrate`. Then, display the final updates on the screen by calling `drawnow` after the loop ends.

```
h = animatedline;
axis([0 4*pi -1 1])
x = linspace(0,4*pi,10000);

for k = 1:length(x)
    y = sin(x(k));
    addpoints(h,x(k),y);
```

```
    drawnow limitrate
end
drawnow
```

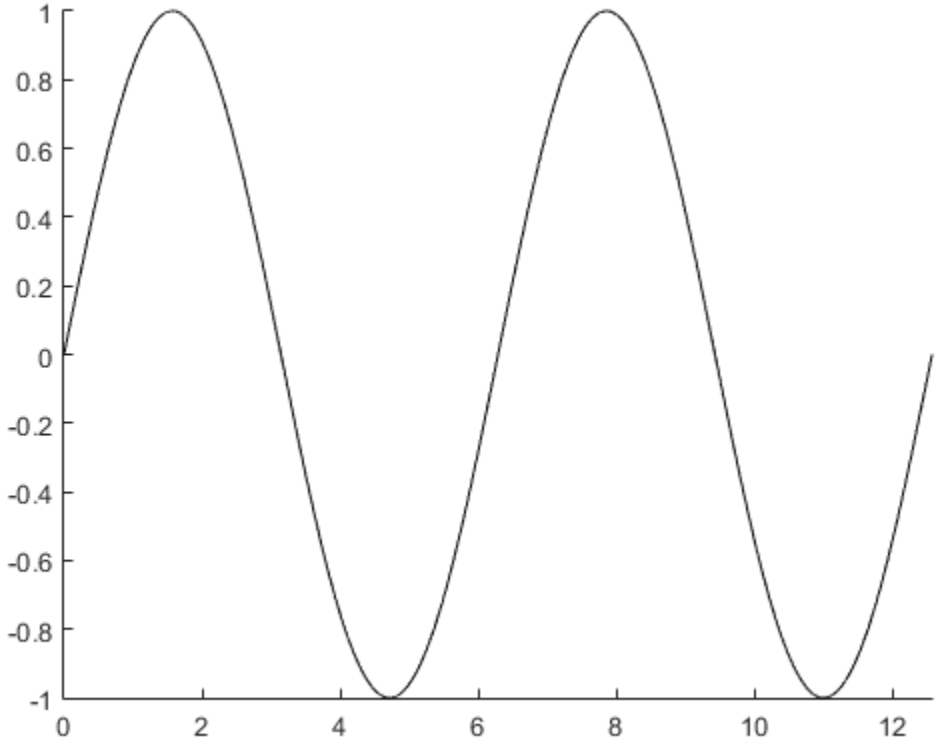


Precompute Data, Then Create Animation

Compute all the data before the animation loop.

```
h = animatedline;
axis([0 4*pi -1 1])
x = linspace(0,4*pi,10000);
y = sin(x);
```

```
for k = 1:length(x)
    addpoints(h,x(k),y(k));
    drawnow limitrate
end
drawnow
```



If you have long computations, precomputing the data can improve perfomance. Precomputing minimizes the computation time by letting the computation run without

interruptions. Additionally, it helps ensure a smooth animation by focusing on only graphics code in the animation loop.

More About

Actions Equivalent to drawnow

These actions are equivalent to calling a full `drawnow` command:

- Returning to the MATLAB prompt.
- Using the `figure`, `getframe`, `input`, `pause`, and `keyboard` functions.
- Using functions that wait for user input, such as `waitforbuttonpress`, `waitfor`, or `ginput`.

Tips

- The `nocallbacks` option always adds interrupting callbacks to the queue. If you want to discard interrupting callbacks, then use the `Interruptible` and `BusyAction` properties instead.

See Also

`pause` | `refreshdata` | `waitfor`

Introduced before R2006a

dsearchn

N-D nearest point search

Syntax

```
k = dsearchn(X,T,XI)
k = dsearchn(X,T,XI,outval)
k = dsearchn(X,XI)
[k,d] = dsearchn(X,...)
```

Description

`k = dsearchn(X,T,XI)` returns the indices `k` of the closest points in `X` for each point in `XI`. `X` is an `m`-by-`n` matrix representing `m` points in n-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in n-dimensional space. `T` is a `numt`-by-`n+1` matrix, a triangulation of the data `X` generated by `delaunayn`. The output `k` is a column vector of length `p`.

`k = dsearchn(X,T,XI,outval)` returns the indices `k` of the closest points in `X` for each point in `XI`, unless a point is outside the convex hull. If `XI(J,:)` is outside the convex hull, then `K(J)` is assigned `outval`, a scalar double. `Inf` is often used for `outval`. If `outval` is `[]`, then `k` is the same as in the case `k = dsearchn(X,T,XI)`.

`k = dsearchn(X,XI)` performs the search without using a triangulation. With large `X` and small `XI`, this approach is faster and uses much less memory.

`[k,d] = dsearchn(X,...)` also returns the distances `d` to the closest points. `d` is a column vector of length `p`.

More About

Algorithms

`dsearchn` is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, “The Quickhull Algorithm for Convex Hulls,” ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469–483.

See Also

`delaunayTriangulation`

Introduced before R2006a

duration

Create duration array from numeric values

The `duration` function creates an array that represents elapsed time in units of fixed length, such as hours, minutes, and seconds. You also can create elapsed times in terms of fixed-length (24-hour) days and fixed-length (365.2425-day long) years.

Syntax

```
D = duration(H,MI,S)
D = duration(H,MI,S,MS)
D = duration(X)

D = duration( ___ ,'Format',displayFormat)
```

Description

`D = duration(H,MI,S)` creates a duration array from numeric arrays containing the number of hours, minutes, and seconds specified by `H`, `MI`, and `S`, respectively.

`D = duration(H,MI,S,MS)` creates a duration array from numeric arrays containing the number of hours, minutes, second, and milliseconds specified by `H`, `MI`, `S`, and `MS`, respectively.

`D = duration(X)` creates a column vector of durations from a numeric matrix.

`D = duration( ___ ,'Format',displayFormat)` additionally specifies the format in which `D` displays. You can use this syntax with any of the arguments from the previous syntaxes.

Examples

Create Duration Array with Default Format

```
D = duration(1,30:33,0)
```

```
D =

   01:30:00    01:31:00    01:32:00    01:33:00
```

Create Duration Array and Specify Format

Create a `duration` array from a matrix and display the values in units of fractional hours.

```
X = magic(3);
D = duration(X,'Format','h')
```

```
D =

   8.0183 hrs
   3.0853 hrs
   4.1506 hrs
```

Input Arguments

`H,MI,S` — Hour, minute, and second arrays

scalar | vector | matrix | multidimensional array

Hour, minute, and second arrays specified as a scalar, vector, matrix or multidimensional array. These arrays must be the same size, or any one can be a scalar.

Example: `12,45,07.451`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`H,MI,S,MS` — Hour, minute, second, and millisecond arrays

scalar | vector | matrix | multidimensional array

Hour, minute, second, and millisecond arrays, each specified as a scalar, vector, matrix or multidimensional array. The arrays must be the same size, or any one can be a scalar.

Example: `12,45,30,35`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

X — Input matrix

matrix

Input matrix, specified in the form `[H,M,S]`.

Example: `[1,30,0]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

displayFormat — Display format of D

`'hh:mm:ss'` (default) | string

Display format of `D`, the duration array, specified as a string. The `displayFormat` value does not change the duration values in `D`, only their display.

To display a duration as a single number that includes a fractional part (for example, 1.234 hours), specify one of the following strings.

Value of `displayFormat`	Description
`'y'`	Number of exact fixed-length years. A fixed-length year is equal to 365.2425 days.
`'d'`	Number of exact fixed-length days. A fixed-length day is equal to 24 hours.
`'h'`	Number of hours
`'m'`	Number of minutes
`'s'`	Number of seconds

To display a duration in the form of a digital timer, specify one of the strings:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

In addition, you can display up to nine fractional second digits by appending up to nine `S` characters. For example, `'hh:mm:ss.SSS'` displays the milliseconds of a duration value to three digits.

Data Types: `char`

Output Arguments

`D` — Output duration

`duration` array

Output duration, returned as a `duration` array.

You can change the display format of a duration by modifying the `Format` property. For example, to display the duration, `D`, as a number of minutes, type:

```
D.Format = 'm'
```

More About

- "Represent Dates and Times in MATLAB"

See Also

`calendarDuration`

Introduced in R2014b

dynamicprops

Abstract class used to derive handle class with dynamic properties

Syntax

```
classdef myclass < dynamicprops
```

Description

`classdef myclass < dynamicprops` makes *`myclass`* a subclass of the `dynamicprops` class, which is a subclass of the `handle` class.

Use the `dynamicprops` class to derive classes that can define dynamic properties (instance properties), which are associated with a specific objects, but have no effect on the objects class definition. Dynamic properties are useful for attaching temporary data to one or more objects.

dynamicprops Methods

This class defines one method `addprop` and, as a subclass of the `handle` class, inherits all the `handle` class methods.

- `addprop` — adds the named property to the specified handle objects. See "Dynamic Properties — Adding Properties to an Instance" for more information.

See Also

`handle`

echo

Display statements during function execution

Syntax

```
echo on
echo off
echo
echo fcnname on
echo fcnname off
echo fcnname
echo on all
echo off all
```

Description

The `echo` command controls the display (or *echoing*) of statements in a function during their execution. Normally, statements in a function file are not displayed on the screen during execution. Command echoing is useful for debugging or for demonstrations, allowing the commands to be viewed as they execute.

The `echo` command behaves in a slightly different manner for script files and function files. For script files, the use of `echo` is simple; echoing can be either `on` or `off`, in which case any script used is affected.

`echo on` turns on the echoing of commands in all script files.

`echo off` turns off the echoing of commands in all script files.

`echo` toggles the echo state.

When you turn on echoing for a script or function file, each line in the file is displayed as it is executed. Since this results in inefficient execution, use `echo` only for debugging.

`echo fcnname on` turns on echoing of the named function file.

`echo fcnname off` turns off echoing of the named function file.

`echo` *`fcnname`* toggles the echo state of the named function file.

`echo on all` sets echoing on for all function files.

`echo off all` sets echoing off for all function files.

More About

Tips

- To avoid confusing syntax, do not use `on` or `off` as a function name.

See Also

`function`

Introduced before R2006a

echodemo

Run example script step-by-step in Command Window

Syntax

```
echodemo filename
echodemo(filename,index)
```

Description

`echodemo filename` runs the script specified by `filename` step-by-step in the Command Window. The file must contain sections defined with two percent signs (`%%`) to enable pausing after each step. At each step, you can click links in the Command Window to proceed or stop. If the Command Window is not large enough to show the links, scroll up to see them.

Caution If variables in your base workspace have the same name as variables that the example file creates, the example could overwrite your data. Preserve your data by saving it to a MAT-file before running the example.

`echodemo(filename,index)` starts with the section number specified by `index`. If the example relies on results of previous steps, using this syntax can produce errors or unexpected results.

Examples

Run Example Script in Command Window

Run the Loma Prieta Earthquake example.

```
echodemo quake
```

Start Script from Specified Section

Start the Loma Prieta Earthquake example from the third section.

```
filename = 'quake';
index = 3;
echodemo(filename,index)
```

This code errors because the example requires variables created in earlier sections.

Input Arguments

`filename` — Script file name

string

Script file name, specified as a string.

When you use the function syntax for `echodemo` and specify its inputs within parentheses, enclose the `filename` input in single quotes.

`index` — Section index

scalar integer

Section index, specified as a scalar integer.

The link text in the Command Window shows the current section number, `n`, and the total number of sections, `m`, as `n/m`.

More About

Tips

- Only use `echodemo` to display scripts, not functions. `echodemo` can run any script that you can execute, but only scripts with sections pause between steps.

See Also

`demo` | `doc` | `publish`

Introduced before R2006a

edgeAttachments

Class: TriRep

(Will be removed) Simplices attached to specified edges

Note: `edgeAttachments(TriRep)` will be removed in a future release. Use `edgeAttachments(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
SI = edgeAttachments(TR, V1, V2)
SI = edgeAttachments(TR, EDGE)
```

Description

`SI = edgeAttachments(TR, V1, V2)` returns the simplices `SI` attached to the edges specified by `(V1, V2)`. `(V1, V2)` represents the start and end vertices of the edges to be queried.

`SI = edgeAttachments(TR, EDGE)` specifies edges in matrix format.

Input Arguments

`TR`	Triangulation representation.
`V1,V2`	Column vectors of vertex indices into the array of points representing the vertex coordinates.
`EDGE`	Matrix specifying edge start and end points. `EDGE` is of size `m`-by-2, `m` being the number of edges to query.

Output Arguments

`SI`	Vector cell array of indices into the triangulation matrix. `SI` is a cell array because the number of simplices associated with each edge can vary.

Definitions

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Example 1

Load a 3-D triangulation to compute the tetrahedra attached to an edge.

```
load tetmesh
trep = TriRep(tet, X);
v1 = [15 21]';
v2 = [936 716]';
t1 = edgeAttachments(trep, v1, v2);
```

You can also specify the input as edges.

```
e = [v1 v2];
t2 = edgeAttachments(trep, e);
isequal(t1,t2);
```

Example 2

Create a triangulation with `DelaunayTri`.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

Query the triangles attached to edge (1,5).

```
t = edgeAttachments(dt, 1,5);
```

```
t{:};
```

See Also

triangulation | edges | delaunayTriangulation

edges

Class: TriRep

(Will be removed) Triangulation edges

Note: `edges(TriRep)` will be removed in a future release. Use `edges(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
E = edges(TR)
```

Description

`E = edges(TR)` returns the edges in the triangulation in an `n`-by-2 matrix. `n` is the number of edges. The vertices of the edges index into `TR.X`, the array of points representing the vertex coordinates.

Input Arguments

`TR`	Triangulation representation.

Output Arguments

`E`	Edge matrix.

Examples

Example 1

Load a 2-D triangulation.

```
load trimesh2d
trep = TriRep(tri, x,y);
```

Return all edges.

```
e = edges(trep);
```

Example 2

Query a 2-D `DelaunayTri`-generated triangulation.

```
X = rand(10,2);
dt = DelaunayTri(X);
e = edges(dt);
```

See Also

`triangulation` | `edgeAttachments` | `delaunayTriangulation`

edit

Edit or create file

Syntax

```
edit
edit file
edit file1 ... fileN
```

Description

`edit` opens a new file called `Untitled` in the Editor. MATLAB does not automatically save `Untitled`.

`edit file` opens the specified file in the Editor. If file does not already exist, MATLAB asks if you want to create it. file can include a partial path, complete path, relative path, or no path. If `file` includes a partial path or no path, `edit` will look for the file on the search path. You must have write permission to the path to create file, otherwise, MATLAB ignores the argument.

You must specify the extension to open `.mat` and `.mdl` files. MATLAB cannot directly edit binary files, such as `.p` and `.mex` files.

`edit file1 ... fileN` opens each file, `file1 ... fileN`, in the Editor.

Examples

Open a New File

```
edit
```

A new file titled `Untitled` opens in the MATLAB Editor (or default editor). `Untitled` does not appear in your Current Folder.

Create a New File

```
mkdir tests
edit tests/new_script.m
```

A dialog box appears, asking if you want to create `new_script.m`. If you select **Yes**, MATLAB creates and opens `tests/new_script.m`.

Open Files

```
edit file1 file2 file3 file4
```

MATLAB sequentially creates and opens the files: `file1`, `file2`, `file3`, and `file4` in sequence.

Input Arguments

`file` — Name of file

string

Name of file, specified as a string. If file specifies a path that contains a nonexistent folder, MATLAB throws an error. Specify multiple files on the same line by separating filenames with a space.

If `file` is overloaded (that is, appears in multiple folders on the search path), then include a partial path to edit the correct page, such as

```
edit folderName/file
```

If you do not specify the extension, then `edit` opens a file with the specified name and a `.m` extension.

```
edit name
```

If the file is part of a class or package, then either specify the path and extension, or separate the components of the name with periods, such as:

```
edit className.name
edit packageName.name
edit packageName.className.name
edit packageName.name
```

Data Types: `char`

More About

- “Change Default Editor”

See Also

`open` | `type`

Introduced before R2006a

eig

Eigenvalues and eigenvectors

Syntax

```
e = eig(A)
[V,D] = eig(A)
[V,D,W] = eig(A)

e = eig(A,B)
[V,D] = eig(A,B)
[V,D,W] = eig(A,B)

[___] = eig(A,balanceOption)
[___] = eig(A,B,algorithm)

[___] = eig(___,eigvalOption)
```

Description

`e = eig(A)` returns a column vector containing the eigenvalues of square matrix `A`.

`[V,D] = eig(A)` returns diagonal matrix `D` of eigenvalues and matrix `V` whose columns are the corresponding right eigenvectors, so that `A*V = V*D`.

`[V,D,W] = eig(A)` also returns full matrix `W` whose columns are the corresponding left eigenvectors, so that `W'*A = D*W'`.

The eigenvalue problem is to determine the solution to the equation $Av = \lambda v$, where A is an `n`-by-`n` matrix, v is a column vector of length `n`, and λ is a scalar. The values of λ that satisfy the equation are the eigenvalues. The corresponding values of v that satisfy the equation are the right eigenvectors. The left eigenvectors, w, satisfy the equation $w'A = \lambda w'$.

`e = eig(A,B)` returns a column vector containing the generalized eigenvalues of square matrices `A` and `B`.

`[V,D] = eig(A,B)` returns diagonal matrix `D` of generalized eigenvalues and full matrix `V` whose columns are the corresponding right eigenvectors, so that `A*V = B*V*D`.

`[V,D,W] = eig(A,B)` also returns full matrix `W` whose columns are the corresponding left eigenvectors, so that `W'*A = D*W'*B`.

The generalized eigenvalue problem is to determine the solution to the equation $Av = \lambda Bv$, where A and B are `n`-by-`n` matrices, v is a column vector of length `n`, and λ is a scalar. The values of λ that satisfy the equation are the generalized eigenvalues. The corresponding values of v are the generalized right eigenvectors. The left eigenvectors, w, satisfy the equation $w'A = \lambda w'B$.

`[ ___ ] = eig(A,balanceOption)`, where `balanceOption` is `'nobalance'`, disables the preliminary balancing step in the algorithm. The default for `balanceOption` is `'balance'`, which enables balancing. The `eig` function can return any of the output arguments in previous syntaxes.

`[ ___ ] = eig(A,B,algorithm)`, where `algorithm` is `'chol'`, uses the Cholesky factorization of `B` to compute the generalized eigenvalues. The default for `algorithm` depends on the properties of `A` and `B`, but is generally `'qz'`, which uses the QZ algorithm.

If `A` is Hermitian and `B` is Hermitian positive definite, then the default for `algorithm` is `'chol'`.

`[ ___ ] = eig( ___ ,eigvalOption)` returns the eigenvalues in the form specified by `eigvalOption` using any of the input or output arguments in previous syntaxes. Specify `eigvalOption` as `'vector'` to return the eigenvalues in a column vector or as `'matrix'` to return the eigenvalues in a diagonal matrix.

Examples

Eigenvalues of Matrix

Use `gallery` to create a symmetric positive definite matrix.

```
A = gallery('lehmer',4)
```

```
A =
```

```
    1.0000    0.5000    0.3333    0.2500
    0.5000    1.0000    0.6667    0.5000
    0.3333    0.6667    1.0000    0.7500
    0.2500    0.5000    0.7500    1.0000
```

Calculate the eigenvalues of `A`. The result is a column vector.

```
e = eig(A)
```

```
e =

    0.2078
    0.4078
    0.8482
    2.5362
```

Alternatively, use `eigvalOption` to return the eigenvalues in a diagonal matrix.

```
D = eig(A,'matrix')
```

```
D =

    0.2078         0         0         0
         0    0.4078         0         0
         0         0    0.8482         0
         0         0         0    2.5362
```

Eigenvalues and Eigenvectors of Matrix

Use `gallery` to create a circulant matrix.

```
A = gallery('circul',3)
```

```
A =

     1     2     3
     3     1     2
     2     3     1
```

Calculate the eigenvalues and right eigenvectors of `A`.

```
[V,D] = eig(A)
```

```
V =
```

```
  -0.5774 + 0.0000i   0.2887 - 0.5000i   0.2887 + 0.5000i
  -0.5774 + 0.0000i  -0.5774 + 0.0000i  -0.5774 + 0.0000i
  -0.5774 + 0.0000i   0.2887 + 0.5000i   0.2887 - 0.5000i

D =

   6.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i  -1.5000 + 0.8660i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i  -1.5000 - 0.8660i
```

Verify that the results satisfy `A*V = V*D`.

```
A*V - V*D

ans =

   1.0e-14 *

  -0.2665 + 0.0000i  -0.0444 + 0.0222i  -0.0444 - 0.0222i
   0.0888 + 0.0000i   0.0111 + 0.0777i   0.0111 - 0.0777i
  -0.0444 + 0.0000i  -0.0111 + 0.0833i  -0.0111 - 0.0833i
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since `eig` performs the decomposition using floating-point computations, then `A*V` can, at best, approach `V*D`. In other words, `A*V - V*D` is close to, but not exactly, `0`.

Eigenvalues of a Matrix Whose Elements Differ Dramatically in Scale

```
A = [ 3.0     -2.0      -0.9     2*eps;
     -2.0      4.0       1.0    -eps;
     -eps/4    eps/2    -1.0     0;
     -0.5     -0.5       0.1     1.0];
```

Calculate the eigenvalues and right eigenvectors using the default (balancing) behavior.

```
[VB,DB] = eig(A)

VB =

    0.6153   -0.4176   -0.0000   -0.1437
   -0.7881   -0.3261   -0.0000    0.1264
   -0.0000   -0.0000   -0.0000   -0.9196
    0.0189    0.8481    1.0000    0.3432
```

```
DB =

    5.5616         0         0         0
         0    1.4384         0         0
         0         0    1.0000         0
         0         0         0   -1.0000
```

Verify that the results satisfy `A*VB = VB*DB`.

```
A*VB - VB*DB

ans =

    0.0000    0.0000   -0.0000    0.0000
         0   -0.0000    0.0000   -0.0000
    0.0000   -0.0000    0.0000    0.0000
         0    0.0000    0.0000    0.6031
```

This result does not satisfy `A*VB = VB*DB`. Ideally, the eigenvalue decomposition satisfies this relationship. Since `eig` performs the decomposition using floating-point computations, then `A*V` can, at best, approach `V*D`. In other words, `A*V - V*D` is close to, but not exactly, `0`.

Now, try calculating the eigenvalues and right eigenvectors without the balancing step.

```
[VN,DN] = eig(A,'nobalance')

VN =

    0.6153   -0.4176   -0.0000   -0.1528
   -0.7881   -0.3261         0    0.1345
   -0.0000   -0.0000   -0.0000   -0.9781
    0.0189    0.8481   -1.0000    0.0443


DN =

    5.5616         0         0         0
         0    1.4384         0         0
         0         0    1.0000         0
         0         0         0   -1.0000
```

Verify that the results satisfy `A*VN = VN*DN`.

```
A*VN - VN*DN
```

```
ans =

   1.0e-14 *

   -0.1776   -0.0111   -0.0559   -0.0167
    0.3553    0.1055    0.0336   -0.0194
    0.0017    0.0002    0.0007         0
    0.0264   -0.0222    0.0222    0.0097
```

`A*VN - VN*DN` is much closer to `0`, so the `'nobalance'` option produces more accurate results in this case.

Left Eigenvectors

Create a 3-by-3 matrix.

```
A = [1 7 3; 2 9 12; 5 22 7];
```

Calculate the right eigenvectors, `V`, the eigenvalues, `D`, and the left eigenvectors, `W`.

```
[V,D,W] = eig(A)
```

```
V =

   -0.2610   -0.9734    0.1891
   -0.5870    0.2281   -0.5816
   -0.7663   -0.0198    0.7912


D =

   25.5548         0         0
         0   -0.5789         0
         0         0   -7.9759


W =

   -0.1791   -0.9587   -0.1881
   -0.8127    0.0649   -0.7477
   -0.5545    0.2768    0.6368
```

Verify that the results satisfy `W'*A = D*W'`.

```
W'*A - D*W'
```

```
ans =

   1.0e-13 *

   -0.0444   -0.1066   -0.0888
   -0.0011    0.0442    0.0333
         0    0.0266    0.0178
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since `eig` performs the decomposition using floating-point computations, then `W'*A` can, at best, approach `D*W'`. In other words, `W'*A - D*W'` is close to, but not exactly, `0`.

Eigenvalues of Nondiagonalizable (Defective) Matrix

Create a 3-by-3 matrix.

```
A = [3 1 0; 0 3 1; 0 0 3];
```

Calculate the eigenvalues and right eigenvectors of `A`.

```
[V,D] = eig(A)
```

```
V =

    1.0000   -1.0000    1.0000
         0    0.0000   -0.0000
         0         0    0.0000

D =

     3     0     0
     0     3     0
     0     0     3
```

`A` has repeated eigenvalues and the eigenvectors are not independent. This means that `A` is not diagonalizable and is, therefore, defective.

Verify that `V` and `D` satisfy the equation, `A*V = V*D`, even though `A` is defective.

```
A*V - V*D
```

```
ans =

   1.0e-15 *
```

```
         0    0.8882   -0.8882
         0         0    0.0000
         0         0         0
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since `eig` performs the decomposition using floating-point computations, then `A*V` can, at best, approach `V*D`. In other words, `A*V - V*D` is close to, but not exactly, `0`.

Generalized Eigenvalues

Create two matrices, `A` and `B`, then solve the generalized eigenvalue problem for the eigenvalues and right eigenvectors of the pair `(A,B)`.

```
A = [1/sqrt(2) 0; 0 1];
B = [0 1; -1/sqrt(2) 0];
[V,D]=eig(A,B)
```

```
V =

   1.0000 + 0.0000i   1.0000 + 0.0000i
   0.0000 - 0.7071i   0.0000 + 0.7071i

D =

   0.0000 + 1.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 - 1.0000i
```

Verify that the results satisfy `A*V = B*V*D`.

```
A*V - B*V*D
```

```
ans =

     0     0
     0     0
```

The residual error `A*V - B*V*D` is exactly zero.

Generalized Eigenvalues Using QZ Algorithm for Badly Conditioned Matrices

Create a badly conditioned symmetric matrix containing values close to machine precision.

```
format long e
```

```
A = diag([10^-16, 10^-15])

A =

     1.000000000000000e-16                         0
                         0     1.000000000000000e-15
```

Calculate the generalized eigenvalues and a set of right eigenvectors using the default algorithm. In this case, the default algorithm is `'chol'`.

```
[V1,D1] = eig(A,A)

V1 =

     1.000000000000000e+08                         0
                         0     3.162277660168380e+07


D1 =

     9.999999999999999e-01                         0
                         0     1.000000000000000e+00
```

Now, calculate the generalized eigenvalues and a set of right eigenvectors using the `'qz'` algorithm.

```
[V2,D2] = eig(A,A,'qz')

V2 =

     1     0
     0     1


D2 =

     1     0
     0     1
```

Check how well the `'chol'` result satisfies `A*V1 = A*V1*D1`.

```
format short
A*V1 - A*V1*D1

ans =
```

```
   1.0e-23 *

    0.1654         0
         0   -0.6617
```

Now, check how well the `'qz'` result satisfies `A*V2 = A*V2*D2`.

```
A*V2 - A*V2*D2

ans =

     0     0
     0     0
```

When both matrices are symmetric, `eig` uses the `'chol'` algorithm by default. In this case, the QZ algorithm returns more accurate results.

Generalized Eigenvalues Where One Matrix is Singular

Create a 2-by-2 identity matrix, `A`, and a singular matrix, `B`.

```
A = eye(2);
B = [3 6; 4 8];
```

Try to calculate the generalized eigenvalues of the matrix, $B^{-1}A$.

```
[V,D] = eig(B\A)

Warning: Matrix is singular to working precision.
Error using eig
Input to EIG must not contain NaN or Inf.
```

Now calculate the generalized eigenvalues and right eigenvectors by passing both matrices to the `eig` function.

```
[V,D] = eig(A,B)

V =

   -0.7500   -1.0000
   -1.0000    0.5000

D =
```

```
    0.0909         0
         0       Inf
```

It is better to pass both matrices separately, and let `eig` choose the best algorithm to solve the problem. In this case, `eig(A,B)` returned a set of eigenvectors and at least one real eigenvalue, even though `B` is not invertible.

Verify $Av = \lambda Bv$ for the first eigenvalue and the first eigenvector.

```
eigval = D(1,1);
eigvec = V(:,1);
A*eigvec - eigval*B*eigvec
```

```
ans =

   1.0e-15 *

    0.1110
    0.2220
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since the decomposition is performed using floating-point computations, then `A*eigvec` can, at best, approach `eigval*B*eigvec`, as it does in this case.

Input Arguments

A — Input matrix

square matrix

Input matrix, specified as a real or complex square matrix.

Data Types: `double` | `single`
Complex Number Support: Yes

B — Generalized eigenvalue problem input matrix

square matrix

Generalized eigenvalue problem input matrix, specified as a square matrix of real or complex values. `B` must be the same size as `A`.

Data Types: `double` | `single`

Complex Number Support: Yes

`balanceOption` — Balance option

`'balance'` (default) | `'nobalance'`

Balance option, specified as one of two strings: `'balance'`, which enables a preliminary balancing step, or `'nobalance'` which disables it. In most cases, the balancing step improves the conditioning of `A` to produce more accurate results. However, there are cases in which balancing produces incorrect results. Specify `'nobalance'` when `A` contains values whose scale differs dramatically. For example, if `A` contains nonzero integers, as well as very small (near zero) values, then the balancing step might scale the small values to make them as significant as the integers and produce inaccurate results.

`'balance'` is the default behavior. For more information about balancing, see `balance`.

Data Types: `char`

`algorithm` — Generalized eigenvalue algorithm

`'chol'` | `'qz'`

Generalized eigenvalue algorithm, specified as `'chol'` or `'qz'`, which selects the algorithm to use for calculating the generalized eigenvalues of a pair.

algorithm	Description
`'chol'`	Computes the generalized eigenvalues of `A` and `B` using the Cholesky factorization of `B`.
`'qz'`	Uses the QZ algorithm, also known as the generalized Schur decomposition. This algorithm ignores the symmetry of `A` and `B`.

In general, the two algorithms return the same result. The QZ algorithm can be more stable for certain problems, such as those involving badly conditioned matrices.

When you omit the `algorithm` argument, the `eig` function selects an algorithm based on the properties of `A` and `B`. It uses the `'chol'` algorithm for symmetric (Hermitian) `A` and symmetric (Hermitian) positive definite `B`. Otherwise, it uses the `'qz'` algorithm.

Regardless of the algorithm you specify, the `eig` function always uses the QZ algorithm when `A` or `B` are not symmetric.

`eigvalOption` — Eigenvalue option

`'vector' | 'matrix'`

Eigenvalue option, specified as `'vector'` or `'matrix'`. This option allows you to specify whether the eigenvalues are returned in a column vector or a diagonal matrix. The default behavior varies according to the number of outputs specified:

- If you specify one output, such as `e = eig(A)`, then the eigenvalues are returned as a column vector by default.
- If you specify two or three outputs, such as `[V,D] = eig(A)`, then the eigenvalues are returned as a diagonal matrix, `D`, by default.

Example: `D = eig(A,'matrix')` returns a diagonal matrix of eigenvalues with the one output syntax.

Data Types: `char`

Output Arguments

`e` — Eigenvalues (returned as vector)

column vector

Eigenvalues, returned as a column vector containing the eigenvalues (or generalized eigenvalues of a pair) with multiplicity.

- When `A` is real and symmetric or complex Hermitian, the values of `e` that satisfy $Av = \lambda v$ are real.
- When `A` is real and skew-symmetric or skew-Hermitian, the values of `e` that satisfy $Av = \lambda v$ are purely imaginary or zero.

`V` — Right eigenvectors

square matrix

Right eigenvectors, returned as a square matrix whose columns are the right eigenvectors of `A` or generalized right eigenvectors of the pair, `(A,B)`. The form and normalization of `V` depends on the combination of input arguments:

- `[V,D] = eig(A)` returns matrix `V`, whose columns are the right eigenvectors of `A` such that `A*V = V*D`. The eigenvectors in `V` are normalized so that the 2-norm of each is 1.

If A is real symmetric, then the right eigenvectors, V, are orthonormal.

- `[V,D] = eig(A,'nobalance')` also returns matrix V. However, the 2-norm of each eigenvector is not necessarily 1.
- `[V,D] = eig(A,B)` and `[V,D] = eig(A,B,algorithm)` returns V as a matrix whose columns are the generalized right eigenvectors that satisfy `A*V = B*V*D`. The 2-norm of each eigenvector is not necessarily 1. In this case, D contains the generalized eigenvalues of the pair, `(A,B)`, along the main diagonal.

  If A is symmetric and B is symmetric positive definite, then the eigenvectors in V are normalized so that the B-norm of each is 1.

D — Eigenvalues (returned as matrix)

diagonal matrix

Eigenvalues, returned as a diagonal matrix with the eigenvalues of A on the main diagonal or the eigenvalues of the pair, `(A,B)`, with multiplicity, on the main diagonal.

- When A is real and symmetric or complex Hermitian, the values of D that satisfy $Av = \lambda v$ are real.
- When A is real and skew-symmetric or skew-Hermitian, the values of D that satisfy $Av = \lambda v$ are purely imaginary or zero.

W — Left eigenvectors

square matrix

Left eigenvectors, returned as a square matrix whose columns are the left eigenvectors of A or generalized left eigenvectors of the pair, `(A,B)`. The form and normalization of W depends on the combination of input arguments:

- `[V,D,W] = eig(A)` returns matrix W, whose columns are the left eigenvectors of A such that `W'*A = D*W'`. The eigenvectors in W are normalized so that the 2-norm of each is 1. If A is symmetric, then W is the same as V.
- `[V,D,W] = eig(A,'nobalance')` also returns matrix W. However, the 2-norm of each eigenvector is not necessarily 1.
- `[V,D,W] = eig(A,B)` and `[V,D,W] = eig(A,B,algorithm)` returns W as a matrix whose columns are the generalized left eigenvectors that satisfy `W'*A = D*W'*B`. The 2-norm of each eigenvector is not necessarily 1. In this case, D contains the generalized eigenvalues of the pair, `(A,B)`, along the main diagonal.

If A and B are symmetric, then W is the same as V.

More About

Tips

- The eig function can calculate the eigenvalues of sparse matrices that are real and symmetric. To calculate the eigenvectors of a sparse matrix, or to calculate the eigenvalues of a sparse matrix that is not real and symmetric, use the eigs function.

See Also

balance | condeig | eigs | hess | qz | schur

Introduced before R2006a

eigs

Largest eigenvalues and eigenvectors of matrix

Syntax

```
d = eigs(A)
[V,D] = eigs(A)
[V,D,flag] = eigs(A)
eigs(A,B)
eigs(A,k)
eigs(A,B,k)
eigs(A,k,sigma)
eigs(A,B,k,sigma)
eigs(A,K,sigma,opts)
eigs(A,B,k,sigma,opts)
eigs(Afun,n,...)
```

Description

`d = eigs(A)` returns a vector of `A`'s six largest magnitude eigenvalues. `A` must be a square matrix. `A` should be large and sparse, though `eigs` will work on full matrices as well. See "Tips" on page 1-2291 below.

`[V,D] = eigs(A)` returns a diagonal matrix `D` of `A`'s six largest magnitude eigenvalues and a matrix `V` whose columns are the corresponding eigenvectors.

`[V,D,flag] = eigs(A)` also returns a convergence flag. If `flag` is `0` then all the eigenvalues converged; otherwise not all converged.

`eigs(A,B)` solves the generalized eigenvalue problem `A*V == B*V*D`. `B` must be the same size as `A`. `eigs(A,[],...)` indicates the standard eigenvalue problem `A*V == V*D`.

`eigs(A,k)` and `eigs(A,B,k)` return the `k` largest magnitude eigenvalues.

`eigs(A,k,sigma)` and `eigs(A,B,k,sigma)` return `k` eigenvalues based on *`sigma`*, which can take any of the following values:

scalar (real or complex, including 0)	The eigenvalues closest to *sigma*. If A is a function, Afun must return Y = (A-sigma*B)\x (i.e., Y = A\x when *sigma* = 0).
'lm'	Largest magnitude (default).
'sm'	Smallest magnitude. Same as *sigma* = 0. If A is a function, Afun must return Y = A\x.

For real symmetric A and symmetric positive-definite B, the following are also options:

'la'	Largest algebraic
'sa'	Smallest algebraic
'be'	Both ends (one more from high end if k is odd)

For nonsymmetric and complex problems, the following are also options:

'lr'	Largest real part
'sr'	Smallest real part
'li'	Largest imaginary part
'si'	Smallest imaginary part

Note The syntax eigs(A,k,...) is not valid when A is scalar. To pass a value for k, you must specify B as the second argument and k as the third (eigs(A,B,k,...)). If necessary, you can set B equal to [], the default.

eigs(A,K,*sigma*,opts) and eigs(A,B,k,*sigma*,opts) specify an options structure. Default values are shown in brackets ({}).

Parameter	Description	Values
opts.issym	1 if A or A-*sigma**B represented by Afun is symmetric, 0 otherwise.	[{0} | 1]
opts.isreal	1 if A or A-*sigma**B represented by Afun is real, 0 otherwise.	[0 | {1}]
opts.tol	Convergence: Ritz estimate residual <= tol*norm(A).	[scalar | {eps}]
opts.maxit	Maximum number of iterations.	[integer | {300}]
opts.p	Number of Lanczos basis vectors.	[integer | {2*k}]

Parameter	Description	Values
	`p >= 2k` (`p >= 2k+1` real nonsymmetric) advised. `p` must satisfy `k < p <= n` for real symmetric, `k+1 < p <= n` otherwise. Note: If you do not specify a `p` value, the default algorithm uses at least 20 Lanczos vectors.	
`opts.v0`	Starting vector.	[`n`-by-1 vector | {randomly generated by `rand`}]
`opts.disp`	Diagnostic information display level.	`[{0} | 1 | 2]`
`opts.cholB`	`1` if `B` is really its Cholesky factor `chol(B)`, `0` otherwise.	`[{0} | 1]`
`opts.permB`	Permutation vector `permB` if sparse `B` is really `chol(B(permB,permB))`.	`[permB | {1:n}]`

`eigs(Afun,n,...)` accepts a function handle, `Afun`, instead of the matrix `A`.

`y = Afun(x)` should return:

`A*x`	if *`sigma`* is not specified, or is a string other than `'sm'`
`A\x`	if *`sigma`* is `0` or `'sm'`
`(A-sigma*I)\x`	if *`sigma`* is a nonzero scalar (standard eigenvalue problem). `I` is an identity matrix of the same size as `A`.
`(A-sigma*B)\x`	if *`sigma`* is a nonzero scalar (generalized eigenvalue problem)

"Parameterizing Functions" explains how to provide additional parameters to the function `Afun`, if necessary.

The matrix `A`, `A-`*`sigma`*`*I` or `A-`*`sigma`*`*B` represented by `Afun` is assumed to be real and nonsymmetric unless specified otherwise by `opts.isreal` and `opts.issym`. In all the `eigs` syntaxes, `eigs(A,...)` can be replaced by `eigs(Afun,n,...)`.

Examples

Smallest Eigenvalues of Sparse Matrix

```
A = delsq(numgrid('C',15));
d1 = eigs(A,5,'sm')
```

returns

```
d1 =
    0.5520
    0.4787
    0.3469
    0.2676
    0.1334
```

Smallest Eigenvalues of Function-Generated Sparse Matrix

This example replaces the matrix `A` in example 1 with a handle to a function `dnRk`. The example is contained in file `run_eigs` that

- Calls `eigs` with the function handle `@dnRk` as its first argument.
- Contains `dnRk` as a nested function, so that all variables in `run_eigs` are available to `dnRk`.

The following shows the code for `run_eigs`:

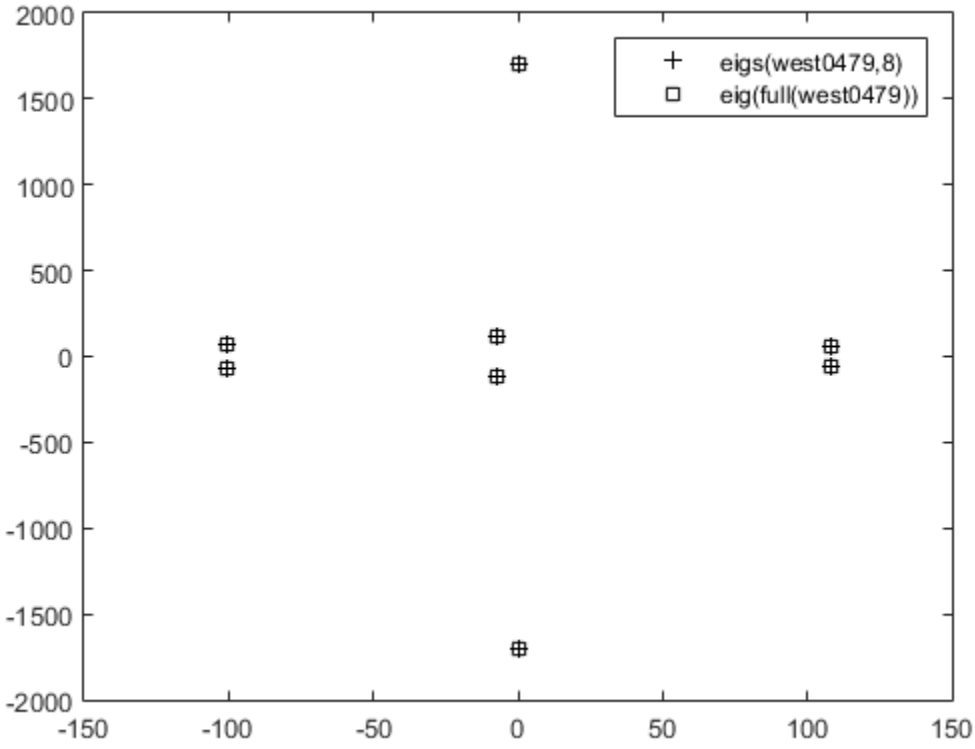
```
function d2 = run_eigs
n = 139;
opts.issym = 1;
R = 'C';
k = 15;
d2 = eigs(@dnRk,n,5,'sm',opts);

    function y = dnRk(x)
        y = (delsq(numgrid(R,k))) \ x;
    end
end
```

Largest Eigenvalue Pairs of Sparse Matrix

west0479 is a real-valued 479-by-479 sparse matrix with both real and complex pairs of conjugate eigenvalues. `eig` computes all 479 eigenvalues. `eigs` easily picks out the largest magnitude eigenvalues.

This plot shows the 8 largest magnitude eigenvalues of west0479 as computed by `eig` and `eigs`.

```
load west0479
d = eig(full(west0479));
dlm = eigs(west0479,8);
[dum,ind] = sort(abs(d));
plot(dlm,'k+')
hold on
plot(d(ind(end-7:end)),'ks')
hold off
legend('eigs(west0479,8)','eig(full(west0479))')
```



Repeated Eigenvalues of Symmetric Positive Definite Sparse Matrix

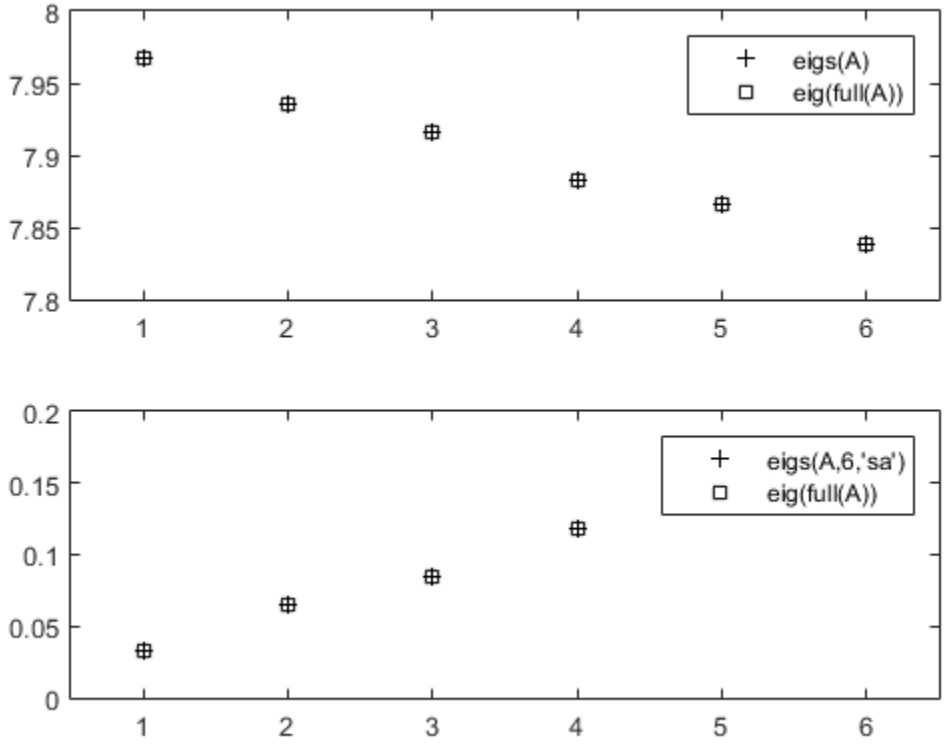
`A = delsq(numgrid('C',30))` is a symmetric positive definite matrix of size 632 with eigenvalues reasonably well-distributed in the interval (0 8), but with 18 eigenvalues repeated at 4.

Use the `eig` function to compute all 632 eigenvalues, and the `eigs` function to compute the six largest and smallest magnitude eigenvalues.

```
A = delsq(numgrid('C',30));
d = sort(eig(full(A)));
dlm = eigs(A);
dsm = eigs(A,6,'sa');
```

Plot the results from `eig` and `eigs` for the six largest and smallest magnitude eigenvalues.

```
subplot(2,1,1)
plot(dlm,'k+')
hold on
plot(d(end:-1:end-5),'ks')
hold off
legend('eigs(A)','eig(full(A))',3)
xlim([0.5 6.5])

subplot(2,1,2)
plot(dsm,'k+')
hold on
plot(d(1:6),'ks')
hold off
legend('eigs(A,6,''sa'')','eig(full(A))',2)
xlim([0.5 6.5])
```



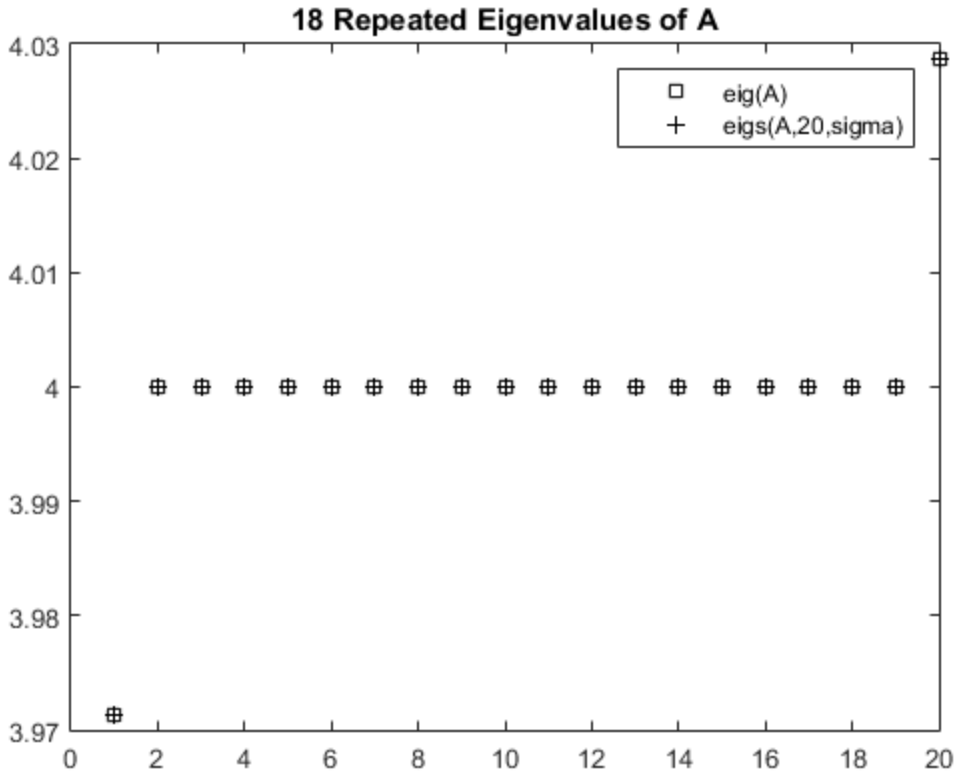
The repeated eigenvalue at 4 must be handled more carefully. The call `eigs(A,20,4.0)` to compute 20 eigenvalues near 4.0 tries to find eigenvalues of `A - 4.0*I`. This involves divisions of the form `1/(lambda - 4.0)`, where lambda is an estimate of an eigenvalue of `A`. As lambda gets closer to 4.0, `eigs` fails. We must use `sigma` near but not equal to 4 to find those eigenvalues.

```
sigma = 4 - 1e-6;
D = sort(eigs(A,20,sigma));
```

The plot below shows the 20 eigenvalues closest to 4 that were computed by `eig`, along with the 20 eigenvalues closest to 4 - 1e-6 that were computed by `eigs`.

```
figure(2)
plot(d(307:326),'ks')
```

```
hold on
plot(D,'k+')
hold off
legend('eig(A)','eigs(A,20,sigma)')
title('18 Repeated Eigenvalues of A')
```



More About

Tips

- `d = eigs(A,k)` is not a substitute for

  `d = eig(full(A))`

```
d = sort(d)
d = d(end-k+1:end)
```

but is most appropriate for large sparse matrices. If the problem fits into memory, it may be quicker to use `eig(full(A))`.

- Unless you provide a start vector with `opts.v0`, the default start vector is generated by `rand`, possibly leading to different iterations each run, and perhaps even different convergence behavior. In order to control this, specify your start vector via `opts.v0`.
- `eigs` does not always return sorted eigenvalues and eigenvectors. Use `sort` to explicitly sort the output eigenvalues and eigenvectors in cases where their order is important.

References

[1] Lehoucq, R.B. and D.C. Sorensen, "Deflation Techniques for an Implicitly Re-Started Arnoldi Iteration," *SIAM J. Matrix Analysis and Applications*, Vol. 17, 1996, pp. 789–821.

[2] Sorensen, D.C., "Implicit Application of Polynomial Filters in a k-Step Arnoldi Method," *SIAM J. Matrix Analysis and Applications*, Vol. 13, 1992, pp.357–385.

See Also

`eig` | `svds` | `function_handle`

Introduced before R2006a

ellipj

Jacobi elliptic functions

Syntax

```
[SN,CN,DN] = ellipj(U,M)
[SN,CN,DN] = ellipj(U,M,tol)
```

Description

`[SN,CN,DN] = ellipj(U,M)` returns the Jacobi elliptic functions SN, CN, and DN evaluated for corresponding elements of argument U and parameter M. Inputs U and M must be the same size, or either U or M must be scalar.

`[SN,CN,DN] = ellipj(U,M,tol)` computes the Jacobi elliptic functions to accuracy `tol`. The default value of tol is `eps`. Increase tol for a less accurate but more quickly computed answer.

Examples

Find the Jacobi Elliptic Functions

Find the Jacobi elliptic functions for `U = 0.5` and `M = 0.25`.

```
[s,c,d] = ellipj(0.5,0.25)
```

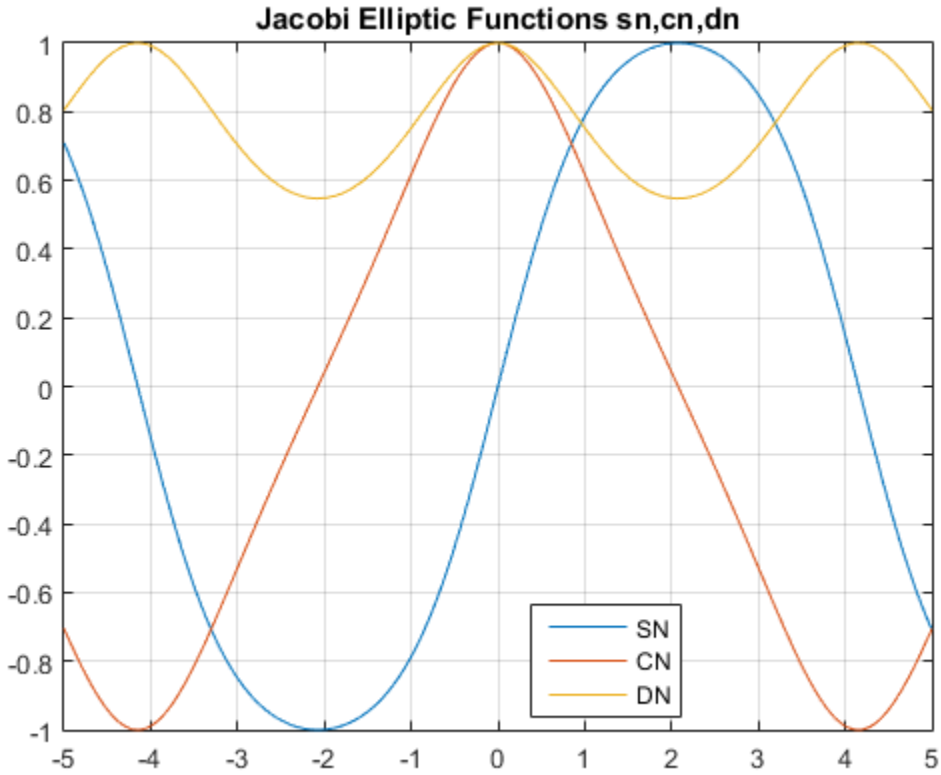
```
s =
    0.4751
c =
    0.8799
d =
    0.9714
```

Plot the Jacobi Elliptic Functions

Plot the Jacobi elliptic functions for `-5≤U≤5` and `M = 0.7`.

```
M = 0.7;
```

```
U = -5:0.01:5;
[S,C,D] = ellipj(U,M);
plot(U,S,U,C,U,D);
legend('SN','CN','DN','Location','best')
grid on
title('Jacobi Elliptic Functions sn,cn,dn')
```

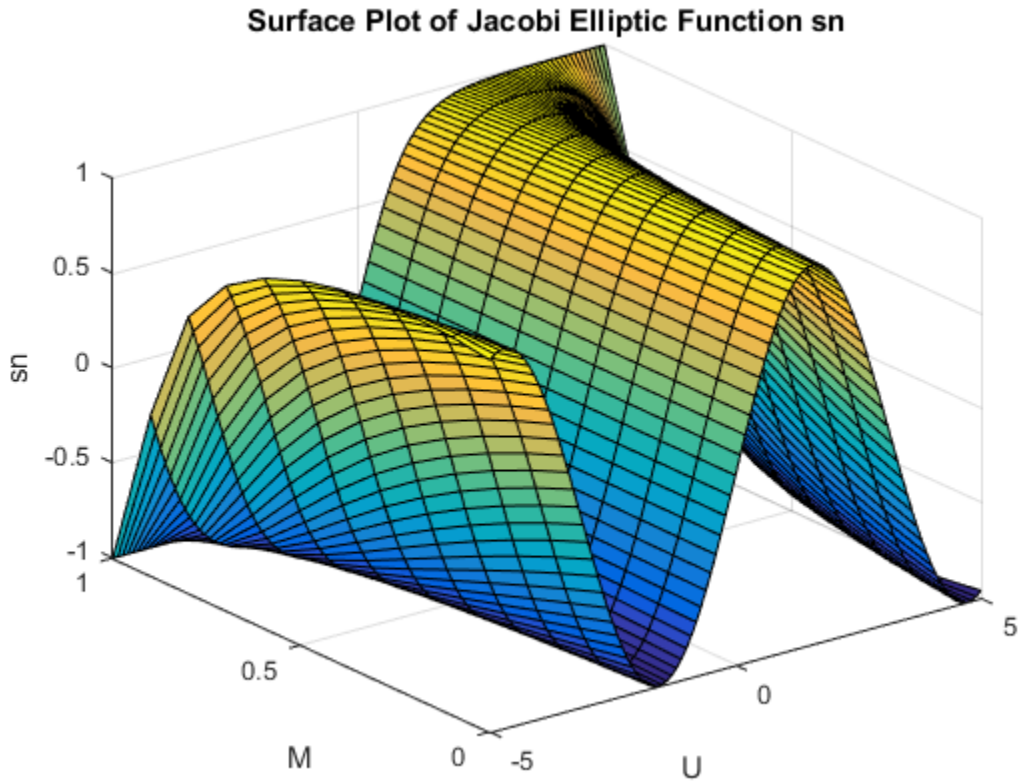


Generate a Surface Plot of the Jacobi Elliptic sn Function

Generate a surface plot of the Jacobi elliptic sn function for the allowed range of `M` and `-5≤U≤5`.

```
[M,U] = meshgrid(0:0.1:1,-5:0.1:5);
S = ellipj(U,M);
```

```
surf(U,M,S)
xlabel('U')
ylabel('M')
zlabel('sn')
title('Surface Plot of Jacobi Elliptic Function sn')
```



Faster Calculations of Jacobi Elliptic Integrals by Changing Tolerance

The default value of `tol` is `eps`. Find the run time with the default value for arbitrary `M` using `tic` and `toc`. Increase `tol` by a factor of 1000 and find the run time. Compare the run times.

```
tic
ellipj(0.253,0.937)
```

```
toc
tic
ellipj(0.253,0.937,eps*1000)
toc


ans =

    0.2479

Elapsed time is 0.048734 seconds.

ans =

    0.2479

Elapsed time is 0.007390 seconds.
```

`ellipj` runs significantly faster when tolerance is significantly increased.

Input Arguments

`U` — Floating-point input

number | vector | matrix | multidimensional array

Floating-point input, specified as a floating-point number, vector, matrix, or multidimensional array. `U` is limited to real values. If U is nonscalar, M must be a scalar or a nonscalar of the same size as U.

Data Types: `single` | `double`

`M` — Floating-point input

number | vector | matrix | multidimensional array

Floating-point input, specified as a floating-point number, vector, matrix, or multidimensional array. M can take values $0 \le m \le 1$. If M is a nonscalar, U must be a scalar or a nonscalar of the same size as M. Map other values of `M` into this range using the transformations described in [1], equations 16.10 and 16.11.

Data Types: `single` | `double`

`tol` — Accuracy of result

eps (default) | nonnegative real number

Accuracy of result, specified as a nonnegative real number. The default value is `eps`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `function_handle`

Output Arguments

SN — Jacobi elliptic function sn

floating-point number | floating-point vector | floating-point matrix | floating-point multidimensional array

Jacobi elliptic function sn, returned as a floating-point number, vector, matrix, or multidimensional array.

CN — Jacobi elliptic function cn

floating-point number | floating-point vector | floating-point matrix | floating-point multidimensional array

Jacobi elliptic function cn, returned as a floating-point number, vector, matrix, or multidimensional array.

DN — Jacobi elliptic function dn

floating-point number | floating-point vector | floating-point matrix | floating-point multidimensional array

Jacobi elliptic function dn, returned as a floating-point number, vector, matrix, or multidimensional array.

More About

Jacobi Elliptic Functions

The Jacobi elliptic functions are defined in terms of the integral

$$u = \int_0^{\phi} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}.$$

Then

$$sn(u) = \sin\phi,\ cn(u) = \cos\phi,\ dn(u) = \sqrt{1 - m\sin^2\phi}.$$

Some definitions of the elliptic functions use the elliptical modulus k or modular angle a instead of the parameter m. They are related by

$$k^2 = m = \sin^2 a.$$

The Jacobi elliptic functions obey many mathematical identities. For a good sample, see [1].

Algorithms

`ellipj` computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean of [1]. It starts with the triplet of numbers

$$a_0 = 1,\ b_0 = \sqrt{1-m},\ c_0 = \sqrt{m}.$$

`ellipj` computes successive iterations using

$$a_i = \frac{1}{2}(a_{i-1} + b_{i-1})$$
$$b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}}$$
$$c_i = \frac{1}{2}(a_{i-1} - b_{i-1}).$$

Next, it calculates the amplitudes in radians using

$$\sin(2\phi_{n-1} - \phi_n) = \frac{c_n}{a_n}\sin(\phi_n),$$

being careful to unwrap the phases correctly. The Jacobian elliptic functions are then simply

$$sn(u) = \sin\phi_0$$
$$cn(u) = \cos\phi_0$$
$$dn(u) = \sqrt{1 - m\cdot sn(u)^2}.$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

See Also

`ellipke`

Introduced before R2006a

ellipke

Complete elliptic integrals of first and second kind

Syntax

```
K = ellipke(M)
[K,E] = ellipke(M)
[K,E] = ellipke(M,tol)
```

Description

`K = ellipke(M)` returns the complete elliptic integral of the first kind for each element in M.

`[K,E] = ellipke(M)` returns the complete elliptic integral of the first and second kind.

`[K,E] = ellipke(M,tol)` computes the complete elliptic integral to accuracy `tol`. The default value of tol is `eps`. Increase tol for a less accurate but more quickly computed answer.

Examples

Find Complete Elliptic Integrals of First and Second Kind

Find the complete elliptic integrals of the first and second kind for `M = 0.5`.

```
M = 0.5;
[K,E] = ellipke(M)
```

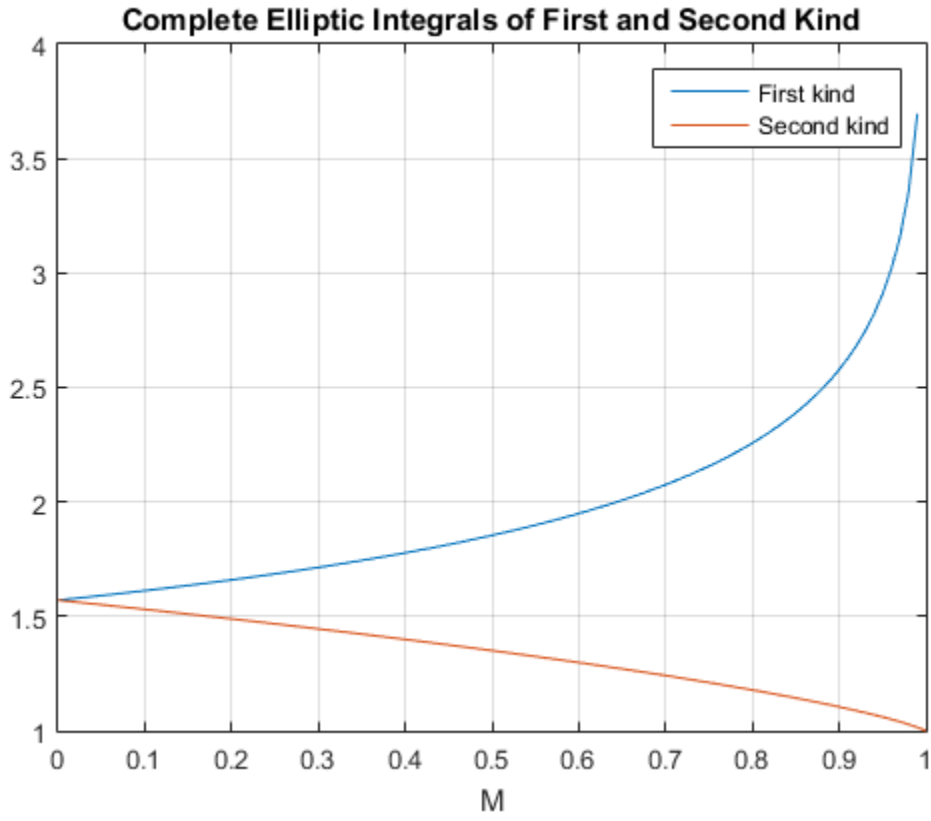
```
K =

    1.8541
```

```
E =

    1.3506
```

Plot Complete Elliptic Integrals of First and Second Kind

Plot the complete elliptic integrals of the first and second kind for the allowed range of `M`.

```
M = 0:0.01:1;
[K,E] = ellipke(M);
plot(M,K,M,E)
grid on
xlabel('M')
title('Complete Elliptic Integrals of First and Second Kind')
legend('First kind','Second kind')
```



Faster Calculations of the Complete Elliptic Integrals by Changing the Tolerance

The default value of `tol` is `eps`. Find the runtime with the default value for arbitrary `M` using `tic` and `toc`. Increase `tol` by a factor of thousand and find the runtime. Compare the runtimes.

```
tic
ellipke(0.904561)
toc
tic
ellipke(0.904561,eps*1000)
toc
```

```
ans =

    2.6001

Elapsed time is 0.060971 seconds.

ans =

    2.6001

Elapsed time is 0.003489 seconds.
```

`ellipke` runs significantly faster when tolerance is significantly increased.

Input Arguments

`M` — Floating-point input

number | vector | matrix | multidimensional array

Floating-point input, specified as a floating-point number, vector, matrix, or multidimensional array. M is limited to values $0 \le m \le 1$.

Data Types: `single` | `double`

`tol` — Accuracy of result

eps (default) | nonnegative real number

Accuracy of result, specified as a nonnegative real number. The default value is `eps`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Output Arguments

`K` — Complete elliptic integral of first kind

floating-point number | floating-point vector | floating-point matrix | floating-point multidimensional array

Complete elliptic integral of the first kind, specified as a floating-point number, vector, matrix, or multidimensional array.

E — Complete elliptic integral of second kind

floating-point number | floating-point vector | floating-point matrix | floating-point multidimensional array

Complete elliptic integral of the second kind, specified as a floating-point number, vector, matrix, or multidimensional array.

More About

Complete Elliptic Integrals of the First and Second Kind

The complete elliptic integral of the first kind is

$$[K(m)] = \int_0^1 [(1-t^2)(1-mt^2)]^{-\frac{1}{2}} dt.$$

where m is the first argument of `ellipke`.

The complete elliptic integral of the second kind is

$$E(m) = \int_0^1 (1-t^2)^{-\frac{1}{2}} (1-mt^2)^{\frac{1}{2}} dt.$$

Some definitions of the elliptic functions use the elliptical modulus k or modular angle α instead of the parameter m. They are related by

$$k^2 = m = \sin^2 \alpha.$$

References

[1] Abramowitz, M., and I. A. Stegun. *Handbook of Mathematical Functions*. Dover Publications, 1965.

See Also

`ellipj`

Introduced before R2006a

ellipsoid

Generate ellipsoid

Syntax

```
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)
ellipsoid(axes_handle,...)
ellipsoid(...)
```

Description

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)` generates a surface mesh described by three `n+1`-by-`n+1` matrices, enabling `surf(x,y,z)` to plot an ellipsoid with center `(xc,yc,zc)` and semi-axis lengths `(xr,yr,zr)`.

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)` uses `n = 20`.

`ellipsoid(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

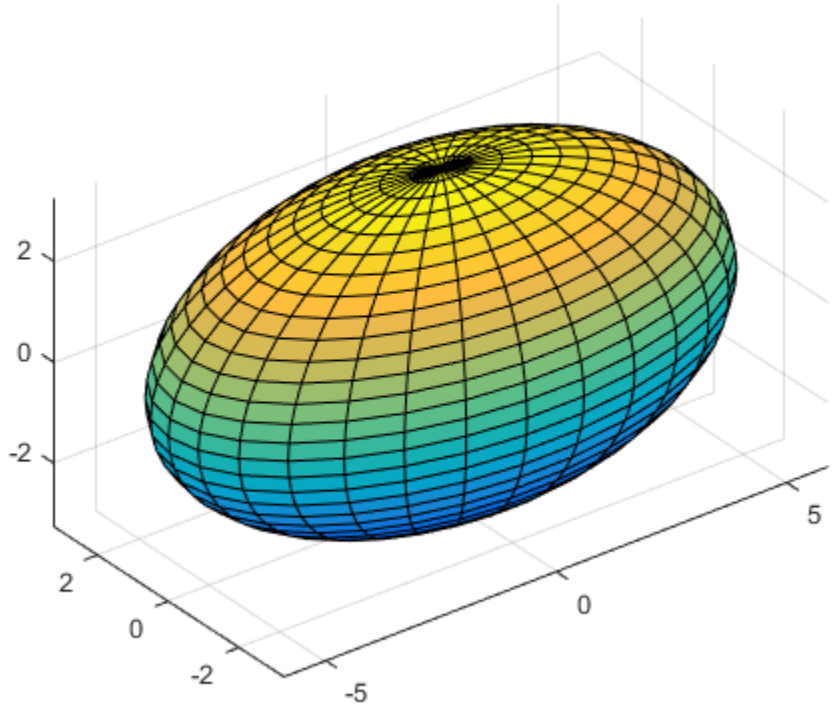
`ellipsoid(...)` with no output arguments plots the ellipsoid as a surface.

Examples

Surface Plot of Ellipsoid

Generate data for an ellipsoid with a center at (0,0,0) and semi-axis lengths (5.9,3.25,3.25). Use `surf` to plot the ellipsoid.

```
[x, y, z] = ellipsoid(0,0,0,5.9,3.25,3.25,30);
figure
surf(x, y, z)
axis equal
```



More About

Algorithms

`ellipsoid` generates the data using the following equation:

$$\frac{(x-xc)^2}{xr^2}+\frac{(y-yc)^2}{yr^2}+\frac{(z-zc)^2}{zr^2}=1$$

Note that `ellipsoid(0,0,0,.5,.5,.5)` is equivalent to a unit sphere.

See Also

`cylinder` | `sphere` | `surf`

Introduced before R2006a

empty

Create empty array

Syntax

```
A = ClassName.empty
A = ClassName.empty(n,m,p,...)
A = ClassName.empty([n,m,p,...])
```

Description

Use `empty` to create empty arrays of the specified class, *`ClassName`*. Specify at least one dimension of the array as 0. MATLAB treats negative values as 0.

`A = ClassName.empty` returns an empty 0-by-0 array of the class of *`ClassName`*.

`A = ClassName.empty(n,m,p,...)` returns an empty rectangular array with the specified dimensions. At least one of the dimensions must be 0.

`A = ClassName.empty([n,m,p,...])` returns an empty rectangular array with the specified dimensions. At least one of the dimensions must be 0. This syntax is useful when using the values returned by the `size` function to define an empty array that is the same size as an existing empty array:

```
A = ClassName.empty(size(otherEmptyArray));
```

Input Arguments

`n,m,p,...`

Dimensions of the empty array. At least one of the specified dimensions must be 0.

Output Arguments

A

An empty array of the specified dimensions and of the class used in the method invocation.

Attributes

`empty` is a hidden, public, static method of all nonabstract MATLAB classes.

`Access`	`Public`
`Hidden`	`true`
`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Use `empty` to create a rectangular empty array of class `int16`:

```
A = int16.empty(5,0);
whos
  Name      Size            Bytes  Class    Attributes

  A         5x0                 0  int16
```

Using the `empty` method of the `int16` class to produce an empty array in which some dimensions are not zero is simpler than using conversion and reshape operations:

```
A = int16([]);
A = reshape(A,5,0);
whos
  Name      Size            Bytes  Class    Attributes

  A         5x0                 0  int16
```

Given the following definition for a class,

```
classdef ExEmpty
   properties
      Color = [1,0,0];
   end
   methods
      function obj = ExEmpty(c)
         if nargin > 0
            obj.Color = c;
         end
      end
   end
end
```

Create an empty array of class `ExEmpty`:

```
A = ExEmpty.empty;
 whos
  Name        Size            Bytes  Class     Attributes

  A           0x0               104  ExEmpty
```

One dimension of an empty array must be zero:

```
A5 = ExEmpty.empty(0,5);
 whos
  Name        Size            Bytes  Class     Attributes

  A5           0x5               104  ExEmpty
```

Empty object arrays follow array concatenation behavior:

```
B = [A,A5]
B =

  0x5 ExEmpty array with properties:

    Color
```

You cannot index into an empty array:

```
A5(1)
Index exceeds matrix dimensions.
```

You can use the `isempty`, `size`, and `length` functions to identify empty object arrays:

```
isempty(A5)
```

```
ans =

     1
size(A5)

ans =

     0     5
length(A5)

ans =

     0
```

Class of Empty Object Array

The `empty` method enables you to initialize arrays of a specific class:

```
C = char.empty(0,7)

C =

   Empty string: 0-by-7

class(C)
ans =

char
```

Initializing an array with empty brackets (`[ ]`):

```
a = [];
```

produces an array of class `double`:

```
class(a)
ans =

double
```

See Also

`| isempty | size | length`

enableNETfromNetworkDrive

Enable access to .NET commands from network drive

Syntax

```
enableNETfromNetworkDrive
```

Description

`enableNETfromNetworkDrive` adds an entry for the MATLAB interface to .NET module to the security policy on your machine. You must have administrative privileges to make changes to your configuration.

Compatibility

Use `enableNETfromNetworkDrive` for MATLAB releases R2012b or earlier, which support installed versions 2.0, 3.0, and 3.5 of the Microsoft .NET Framework.

Related Examples

- "Troubleshooting Security Policy Settings From Network Drives"

Introduced in R2009b

enableservice

Enable, disable, or report status of MATLAB Automation server

Syntax

```
state = enableservice('AutomationServer',enable)
state = enableservice('AutomationServer')
```

Description

`state = enableservice('AutomationServer',enable)` enables or disables the MATLAB Automation server. If `enable` is `true` (logical `1`), `enableservice` converts an existing MATLAB session into an Automation server. If `enable` is `false` (logical `0`), `enableservice` disables the MATLAB Automation server. `state` indicates the previous state of the Automation server. If `state` = `1`, MATLAB was an Automation server. If `state` = `0`, MATLAB was not an Automation server.

`state = enableservice('AutomationServer')` returns the current state of the Automation server. If `state` is logical `1` (`true`), MATLAB is an Automation server.

COM functions are available on Microsoft Windows systems only.

Examples

Enable the Automation server in the current MATLAB session:

```
state = enableservice('AutomationServer',true);
```

Show the current state of the MATLAB session. MATLAB displays `true`:

```
state = enableservice('AutomationServer')
```

Enable the Automation server and show the previous state. MATLAB displays `true`. The previous state can be the same as the current state:

```
state = enableservice('AutomationServer',true)
```

To enable the Automation server every time you run MATLAB, see “Manually Create Automation Server”.

More About

- “MATLAB COM Automation Server Interface”

See Also

`actxserver`

Introduced before R2006a

end

Terminate block of code, or indicate last array index

Syntax

```
end
```

Description

`end` terminates `for`, `while`, `switch`, `try`, `if`, and `parfor` statements. Without an `end` statement, `for`, `while`, `switch`, `try`, `if`, and `parfor` wait for further input. Each `end` is paired with the closest previous unpaired `for`, `while`, `switch`, `try`, `if`, or `parfor` and serves to delimit its scope.

`end` also marks the termination of a function, although in many cases it is optional. If your function contains one or more nested functions, then you must terminate every function in the file, whether nested or not, with `end`. This includes primary, nested, private, and local functions.

The `end` function also serves as the last index in an indexing expression. In that context, `end = (size(x,k))` when used as part of the `k`th index. Examples of this use are `X(3:end)` and `X(1,1:2:end-1)`. When using `end` to grow an array, as in `X(end+1)=5`, make sure `X` exists first.

You can overload the `end` statement for a user object by defining an `end` method for the object. The `end` method should have the calling sequence `end(obj,k,n)`, where `obj` is the user object, `k` is the index in the expression where the `end` syntax is used, and `n` is the total number of indices in the expression. For example, consider the expression

```
A(end-1,:)
```

The MATLAB software calls the `end` method defined for `A` using the syntax

```
end(A,1,2)
```

Examples

This example shows `end` used with the `for` and `if` statements.

```
for k = 1:n
    if a(k) == 0
        a(k) = a(k) + 2;
    end
end
```

In this example, `end` is used in an indexing expression.

```
A = magic(5)

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

B = A(end,2:end)

B =

    18    25     2     9
```

See Also

`break` | `for` | `if` | `parfor` | `return` | `switch` | `try` | `while`

Introduced before R2006a

EndInvoke

Retrieve result of asynchronous call initiated by .NET System.Delegate BeginInvoke method

Syntax

```
result = EndInvoke(asyncResult)
[res0,...,resN] = EndInvoke(res0,...,resN,asyncResult)
```

Description

`result = EndInvoke(asyncResult)` retrieves result of asynchronous call initiated by `BeginInvoke` method.

`[res0,...,resN] = EndInvoke(res0,...,resN,asyncResult)` for methods with `out` and/or `ref` parameters.

Input Arguments

`asyncResult`

.NET System.IAsyncResult object returned by `BeginInvoke`.

`res0,...,resN`

For methods with `out` and/or `ref` parameters, results of the asynchronous call. The number of arguments is the sum of:

- Number of return values (`0` or `1`).
- Number of `out` and `ref` arguments.

Output Arguments

`result`

Results of the asynchronous call.

`res0,...,resN`

For methods with `out` and/or `ref` parameters, results of the asynchronous call,

Examples

The following examples show how to call delegates with various input and output arguments. Each example contains:

1. The C# delegate signature. In order to execute the MATLAB code, build the delegate code into an assembly named `SignatureExamples` and load it into MATLAB. For information, see "Build a .NET Application for MATLAB Examples".
2. An example MATLAB function to use with the delegate, which must exist on your path.
3. The `BeginInvoke` and `EndInvoke` signatures MATLAB creates. To display the signatures, create a delegate instance, `myDel`, and call the `methodsview` function.
4. Simple MATLAB example.

This example shows how to use a delegate that has no return value.

1. C# delegate:

```
public delegate void delint(Int32 arg);
```

2. MATLAB function to call:

```
%Display input argument
function dispfnc(A)
%A = number
['Input is ' num2str(A)]
end
```

3. MATLAB creates the following signatures. For `BeginInvoke`:

```
System.IAsyncResult RetVal
    BeginInvoke (
        SignatureExamples.delint this,
        int32 scalar arg,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
    EndInvoke (
        SignatureExamples.delint this,
        System.IAsyncResult result)
```

4 Call `dispfnc`:

```
myDel = SignatureExamples.delint(@dispfnc);
asyncRes = myDel.BeginInvoke(6, [], []);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
myDel.EndInvoke(asyncRes)

Input is 6
```

This example shows how to use a delegate with a return value. The delegate does not have `out` or `ref` parameters.

1 C# delegate:

```
public delegate Int32 del2int(Int32 arg1, Int32 arg2);
```

2 MATLAB function to call:

```
%Add input arguments
function res = addfnc(A, B)
%A and B are numbers
res = A + B;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`:

```
System.IAsyncResult RetVal
    BeginInvoke (
        SignatureExamples.del2int this,
        int32 scalar arg1,
        int32 scalar arg2,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
int32 scalar RetVal
    EndInvoke (
        SignatureExamples.del2int this,
        System.IAsyncResult result)
```

4 Call `addfnc`:

```
myDel = SignatureExamples.del2int(@addfnc);
asyncRes = myDel.BeginInvoke(6,8,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
result = myDel.EndInvoke(asyncRes)
```

```
result =
          14
```

This example shows how to use a delegate with a `ref` parameter, `refArg`, and no return value.

1 C# delegate:

```
public delegate void delrefvoid(ref Double refArg);
```

2 MATLAB maps the `ref` argument as both RHS and LHS arguments. MATLAB function to call:

```
%Increment input argument
function res = incfnc(A)
%A = number
res = A + 1;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`:

```
[System.IAsyncResult RetVal,
double scalar refArg]
    BeginInvoke (
        SignatureExamples.delrefvoid this,
        double scalar refArg,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
double scalar refArg
    EndInvoke (
        SignatureExamples.delrefvoid this,
        double scalar refArg,
        System.IAsyncResult result)
```

4 Call `incfnc`:

```
x = 6;
myDel = SignatureExamples.delrefvoid(@incfnc);
asyncRes = myDel.BeginInvoke(x,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
myRef = 0;
result = myDel.EndInvoke(myRef,asyncRes);
disp(['Increment of ' num2str(x) ' = ' num2str(result)]);

Increment of 6 = 7
```

This example shows how to use a delegate with an `out` parameter, `argOut`, and one return value.

1. C# delegate:

   ```
   public delegate Single deloutsingle(Single argIn, out Single argOut);
   ```

2. MATLAB maps the `out` argument as a return value for a total of two return values. MATLAB function to call:

   ```
   %Double input argument
   function [res1 res2] = times2fnc(A)
   res1 = A*2;
   res2 = res1;
   end
   ```

3. MATLAB creates the following signatures. For `BeginInvoke`:

   ```
   [System.IAsyncResult RetVal,
   single scalar argOut]
       BeginInvoke (
           SignatureExamples.deloutsingle this,
           single scalar argIn,
           System.AsyncCallback callback,
           System.Object object)
   ```

   The `EndInvoke` signature:

   ```
   [single scalar RetVal,
   single scalar argOut]
       EndInvoke (
           SignatureExamples.deloutsingle this,
           System.IAsyncResult result)
   ```

4 Call `times2fnc`:

```
myDel = SignatureExamples.deloutsingle(@times2fnc);
asyncRes = myDel.BeginInvoke(6,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
[a1 a2] = myDel.EndInvoke(asyncRes);
a1

a1 =
    12
```

More About

Tips

- If the delegate contains `out` or `ref` parameters, the signature for the `EndInvoke` method follows the MATLAB mapping rules. For information, see "Call Delegates With out and ref Type Arguments".

- "Calling .NET Methods Asynchronously"
- MSDN Calling Synchronous Methods Asynchronously

See Also

`BeginInvoke`

Introduced in R2011a

eomday

Last day of month

Syntax

```
E = eomday(Y,M)
```

Description

`E = eomday(Y,M)` returns the last day of the year and month given by corresponding elements of the numeric arrays `Y` and `M`.

Examples

Show the end of month for January through September for the year 1900:

```
eomday(1900, 1:9)
ans =
    31    28    31    30    31    30    31    31    30
```

Find the number of days during that period:

```
sum(eomday(1900, 1:9))
ans =
   273
```

Because 1996 is a leap year, the statement `eomday(1996,2)` returns `29`. To show all the leap years in the twentieth century, try:

```
y = 1900:1999;
E = eomday(y, 2);
y(find(E == 29))

ans =
  Columns 1 through 6
     1904     1908     1912     1916     1920     1924
```

```
Columns 7 through 12
   1928     1932     1936     1940     1944     1948

Columns 13 through 18
   1952     1956     1960     1964     1968     1972

Columns 19 through 24
   1976     1980     1984     1988     1992     1996
```

See Also

`datenum` | `datevec` | `weekday`

Introduced before R2006a

enumeration

Display class enumeration members and names

Syntax

```
enumeration ClassName
enumeration(obj)
m = enumeration(ClassName)
m = enumeration(obj)
[m,s] = enumeration(ClassName)
[m,s] = enumeration(obj)
```

Description

`enumeration ClassName` displays the names of the enumeration members for the MATLAB class with the name ClassName.

`enumeration(obj)` displays the names of the enumeration members for the class of obj.

`m = enumeration(ClassName)` returns the enumeration members for the class in the column vector m of objects.

`m = enumeration(obj)` returns the enumeration members for the class of object, obj, in the column vector m of objects.

`[m,s] = enumeration(ClassName)` returns the names of the enumeration members in the cell array of strings s. The names in s correspond element-wise to the enumeration members in m.

`[m,s] = enumeration(obj)` returns the enumeration members for the class of object, obj, in the column vector m of objects.

Input Arguments

`ClassName`

The name of the enumeration class, in single quotes.

obj

A instance of an enumeration class.

Output Arguments

m

Column vector of enumeration members.

s

Cell array of strings containing the enumeration names.

Examples

All examples use the following enumeration class.

```
classdef Boolean < logical
   enumeration
      No  (0)
      Yes (1)
      Off (0)
      On  (1)
   end
end
```

Display the names of the enumeration members for class `Boolean`:

```
enumeration Boolean
Enumerations for class Boolean:

    No
    Yes
```

Get the enumeration members for class `Boolean` in a column vector of objects:

```
members = enumeration('Boolean')
members =

    No
```

```
    Yes
```

Get all available enumeration members and their names:

```
[members, names] = enumeration('Boolean')
members =

    No
    Yes
    No
    Yes


names =

    'No'
    'Yes'
    'Off'
    'On'
```

More About

Tips

- An enumeration class that derives from a built-in class can specify more than one name for a given enumeration member.
- When you call the `enumeration` function with no output arguments, MATLAB displays only the first name for each enumeration member (as specified in the class definition). To see all available enumeration members and their names, use the two output arguments (for example, `[m, s] = enumeration(obj);`).

See Also

`classdef`

eps

Floating-point relative accuracy

Syntax

```
d = eps
d = eps(x)
d = eps(datatype)
```

Description

`d = eps` returns the distance from `1.0` to the next largest double-precision number, that is, `eps = 2^-52`.

`d = eps(x)`, where `x` has data type `single` or `double`, returns the positive distance from `abs(x)` to the next largest floating-point number of the same precision as `x`. The command `eps(1.0)` is equivalent to `eps`.

`d = eps(datatype)` returns `eps` according to the data type specified by `datatype`, which can be either `'double'` or `'single'`. The syntax `eps('double')` (default) is equivalent to `eps`, and `eps('single')` is equivalent to `eps(single(1.0))`.

Examples

Accuracy in Double Precision

Display the distance from `1.0` to the next largest double-precision number.

```
d = eps
```

```
d =

   2.2204e-16
```

`eps` is equivalent to `eps(1.0)` and `eps('double')`.

Compute `log2(eps)`.

```
d = log2(eps)
```

```
d =

   -52
```

In base `2`, `eps` is equal to `2^-52`.

Find the distance from `10.0` to the next largest double-precision number.

```
d = eps(10.0)
```

```
d =

   1.7764e-15
```

Accuracy in Single Precision

Display the distance from `1.0` to the next largest single-precision number.

```
d = eps('single')
```

```
d =

  1.1921e-07
```

`eps('single')` is equivalent to `eps(single(1.0))`.

Compute `log2(eps('single'))`.

```
d = log2(eps('single'))
```

```
d =

   -23
```

In base 2, single-precision `eps` is equal to `2^-23`.

Find the distance from the single-precision representation of 10.0 to the next largest single-precision number.

```
d = eps(single(10.0))
```

```
d =

  9.5367e-07
```

Input Arguments

x — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `d` is the same size as `x`. For all `x`, `eps(x) = eps(-x) = eps(abs(x))`. If `x` is complex, `d` is the distance to the next largest floating-point number in magnitude. If `x` is `Inf` or `NaN`, then `eps(x)` returns `NaN`.

Data Types: `single` | `double`
Complex Number Support: Yes

datatype — Output data type

`'double'` (default) | `'single'`

Output data type, specified as `'double'` or `'single'`.

- `eps('double')` is equivalent to `eps` and `eps(1.0)`.
- `eps('single')` is equivalent to `eps(single(1.0))` and `single(2^-23)`.

Data Types: `char`

More About

- Floating-Point Numbers

See Also

`double` | `intmax` | `realmax` | `realmin` | `single`

Introduced before R2006a

eq, ==

Determine equality

Syntax

```
A == B
eq(A,B)
```

Description

`A == B` returns a logical array with elements set to logical `1` (`true`) where arrays `A` and `B` are equal; otherwise, it returns logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays. `eq` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`eq(A,B)` is an alternative way to execute `A == B`, but is rarely used. It enables operator overloading for classes.

Examples

Equality of Two Vectors

Create two vectors containing both real and imaginary numbers.

```
A = [1+i 3 2 4+i];
B = [1 3+i 2 4+i];
```

Compare the two vectors for equality.

```
A == B
```

```
ans =

     0     0     1     1
```

The `eq` function tests both real and imaginary parts for equality, and returns logical `1` (`true`) only where both parts are equal.

Find Characters in String

Create a string of characters.

```
M = 'masterpiece';
```

Test the string for the presence of a specific character using `==`.

```
M == 'e'
```

```
ans =

     0     0     0     0     1     0     0     0     1     0     1
```

The value of logical `1` (`true`) in the vector indicates the presence of the character `'e'` in the string.

Find Values in Categorical Array

Create a categorical array.

```
A = categorical({'heads' 'heads' 'tails'; 'tails' 'heads' 'tails'})
```

```
A =

     heads      heads      tails
     tails      heads      tails
```

The array has two categories: `'heads'` and `'tails'`.

Find all values in the `'heads'` category.

```
A == 'heads'
```

```
ans =

     1     1     0
     0     1     0
```

A value of logical `1` (`true`) indicates a value in the category.

Compare the rows of A for equality.

```
A(1,:) == A(2,:)
```

```
ans =

     0     1     1
```

The function returns logical 1 (true) where the rows have equal category values.

Compare Floating-Point Numbers

Some floating-point numbers cannot be represented exactly in binary form. This leads to small differences in results that the == operator reflects.

Perform a few subtraction operations on a floating-point number and store the result in C.

```
C = 0.5-0.4-0.1
```

```
C =

  -2.7756e-17
```

Intuitively, C should be equal to *exactly* 0. Its small value is due to the nature of floating-point arithmetic.

Compare C to zero for equality.

```
C == 0
```

```
ans =

     0
```

The result is logical 0 (false).

Compare floating-point numbers using a tolerance, tol, instead of ==.

```
tol = eps;
abs(C-0) < tol
```

```
ans =

     1
```

The two numbers, C and 0, are closer to one another than two consecutive floating-point numbers. They are essentially equal.

Compare Datetime Values

Compare the elements of two `datetime` arrays.

Create two `datetime` arrays in different time zones.

```
t1 = [2014,04,14,9,0,0;2014,04,14,10,0,0];
A = datetime(t1,'TimeZone','America/Los_Angeles');
A.Format = 'd-MMM-y HH:mm:ss Z'
```

```
A =

   14-Apr-2014 09:00:00 -0700
   14-Apr-2014 10:00:00 -0700
```

```
t2 = [2014,04,14,12,0,0;2014,04,14,12,30,0];
B = datetime(t2,'TimeZone','America/New_York');
B.Format = 'd-MMM-y HH:mm:ss Z'
```

```
B =

   14-Apr-2014 12:00:00 -0400
   14-Apr-2014 12:30:00 -0400
```

Check where elements in A and B are equal.

```
A==B
```

```
ans =

     1
     0
```

Input Arguments

A — Left array

numeric array | logical array | character array | categorical array | datetime array | duration array

Left array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is a categorical array, the other input can be a categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. If both inputs are categorical arrays that are not ordinal, they can have different sets of categories. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

Complex Number Support: Yes

B — Right array

numeric array | logical array | character array | categorical array | datetime array | duration array

Right array, specified as a numeric array, logical array, character array, or categorical array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is a categorical array, the other input can be a categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. If both inputs are categorical arrays that are not ordinal, they can have different sets of categories. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.
Complex Number Support: Yes

More About

Tips

- When comparing handle objects, use == to test whether objects have the same handle. Use `isequal` to determine if objects with different handles have equal property values.

See Also

`ge` | `gt` | `le` | `lt` | `ne`

Introduced before R2006a

erf

Error function

Syntax

```
erf(x)
```

Description

`erf(x)` returns the "Error Function" on page 1-2343 evaluated for each element of x.

Examples

Find Error Function

Find the error function of a value.

```
erf(0.76)
```

```
ans =

    0.7175
```

Find the error function of the elements of a vector.

```
V = [-0.5 0 1 0.72];
erf(V)
```

```
ans =

   -0.5205         0    0.8427    0.6914
```

Find the error function of the elements of a matrix.

```
M = [0.29 -0.11; 3.1 -2.9];
erf(M)
```

```
ans =

    0.3183   -0.1236
    1.0000   -1.0000
```

Find Cumulative Distribution Function of Normal Distribution

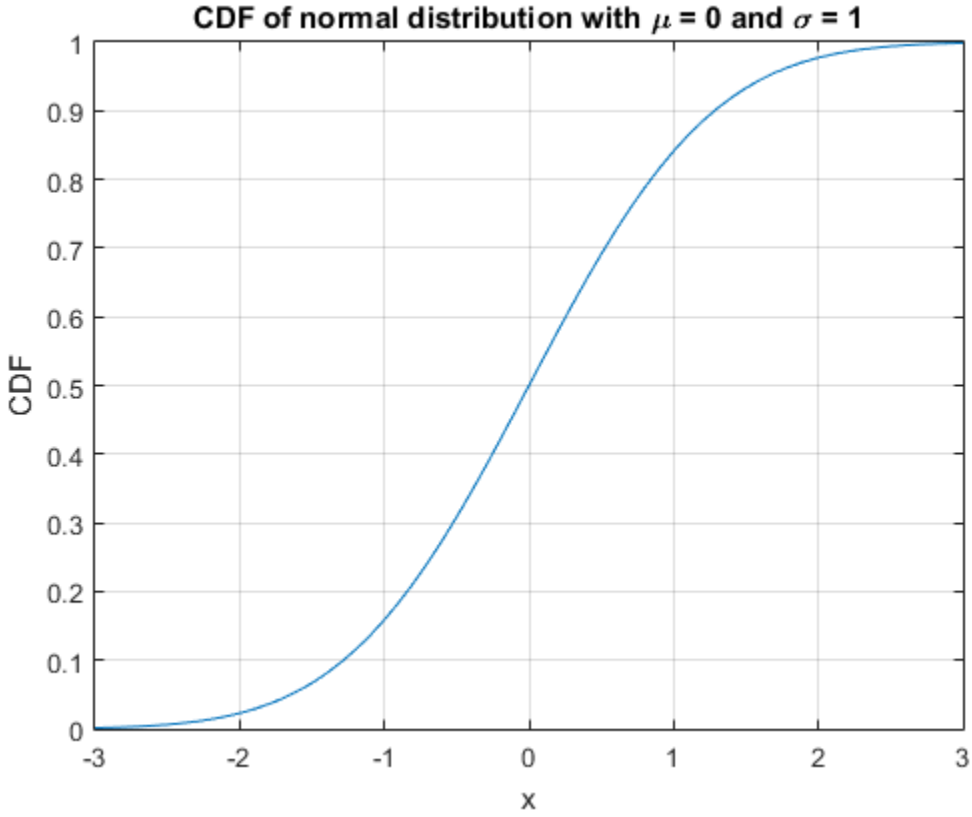
The cumulative distribution function (CDF) of the normal, or Gaussian, distribution with standard deviation σ and mean μ is

$$\phi(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x - \mu}{\sigma\sqrt{2}}\right)\right).$$

Note that for increased computational accuracy, you can rewrite the formula in terms of `erfc` . For details, see “Tips”.

Plot the CDF of the normal distribution with $\mu = 0$ and $\sigma = 1$.

```
x = -3:0.1:3;
y = (1/2)*(1+erf(x/sqrt(2)));
plot(x,y)
grid on
title('CDF of normal distribution with \mu = 0 and \sigma = 1')
xlabel('x')
ylabel('CDF')
```



Calculate Solution of Heat Equation with Initial Condition

Where $u(x,t)$ represents the temperature at position x and time t, the heat equation is

$$\frac{\partial u}{\partial t} = c\frac{\partial^2 u}{\partial x^2},$$

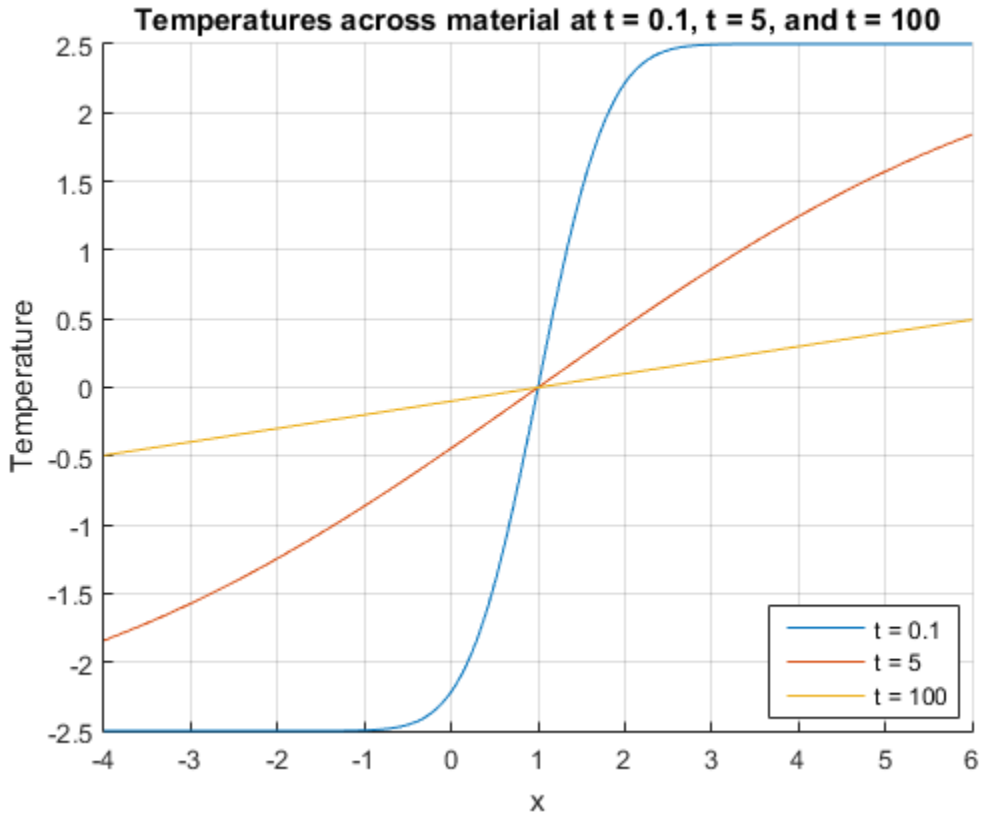
where c is a constant.

For a material with heat coefficient k, and for the initial condition $u(x,0) = a$ for $x > b$ and $u(x,0) = 0$ elsewhere, the solution to the heat equation is

$$u(x,t) = \frac{a}{2}\left(\text{erf}\left(\frac{\text{x} - \text{b}}{\sqrt{4\text{kt}}}\right)\right).$$

For `k = 2`, `a = 5`, and `b = 1`, plot the solution of the heat equation at times `t = 0.1`, `5`, and `100`.

```
x = -4:0.01:6;
t = [0.1 5 100];
a = 5;
k = 2;
b = 1;
figure(1)
hold on
for i = 1:3
    u(i,:) = (a/2)*(erf((x-b)/sqrt(4*k*t(i))));
    plot(x,u(i,:))
end
grid on
xlabel('x')
ylabel('Temperature')
legend('t = 0.1','t = 5','t = 100','Location','best')
title('Temperatures across material at t = 0.1, t = 5, and t = 100')
```



Input Arguments

x — Input

real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. x cannot be sparse.

Data Types: `single` | `double`

More About

Error Function

The error function erf of x is

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Tips

- You can also find the standard normal probability distribution using the Statistics and Machine Learning Toolbox™ function `normcdf`. The relationship between the error function `erf` and `normcdf` is

$$\mathrm{normcdf}(x) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{-x}{\sqrt{2}}\right)\right).$$

- For expressions of the form `1 - erf(x)`, use the complementary error function `erfc` instead. This substitution maintains accuracy. When `erf(x)` is close to `1`, then `1 - erf(x)` is a small number and might be rounded down to `0`. Instead, replace `1 - erf(x)` with `erfc(x)`.

See Also

`erfc` | `erfcinv` | `erfcx` | `erfinv`

Introduced before R2006a

erfc

Complementary error function

Syntax

```
erfc(x)
```

Description

`erfc(x)` returns the "Complementary Error Function" on page 1-2347 evaluated for each element of x. Use the `erfc` function to replace `1 - erf(x)` for greater accuracy when `erf(x)` is close to `1`.

Examples

Find Complementary Error Function

Find the complementary error function of a value.

```
erfc(0.35)
```

```
ans =

    0.6206
```

Find the complementary error function of the elements of a vector.

```
V = [-0.5 0 1 0.72];
erfc(V)
```

```
ans =
```

```
    1.5205    1.0000    0.1573    0.3086
```

Find the complementary error function of the elements of a matrix.

```
M = [0.29 -0.11; 3.1 -2.9];
erfc(M)
```

```
ans =

    0.6817    1.1236
    0.0000    2.0000
```

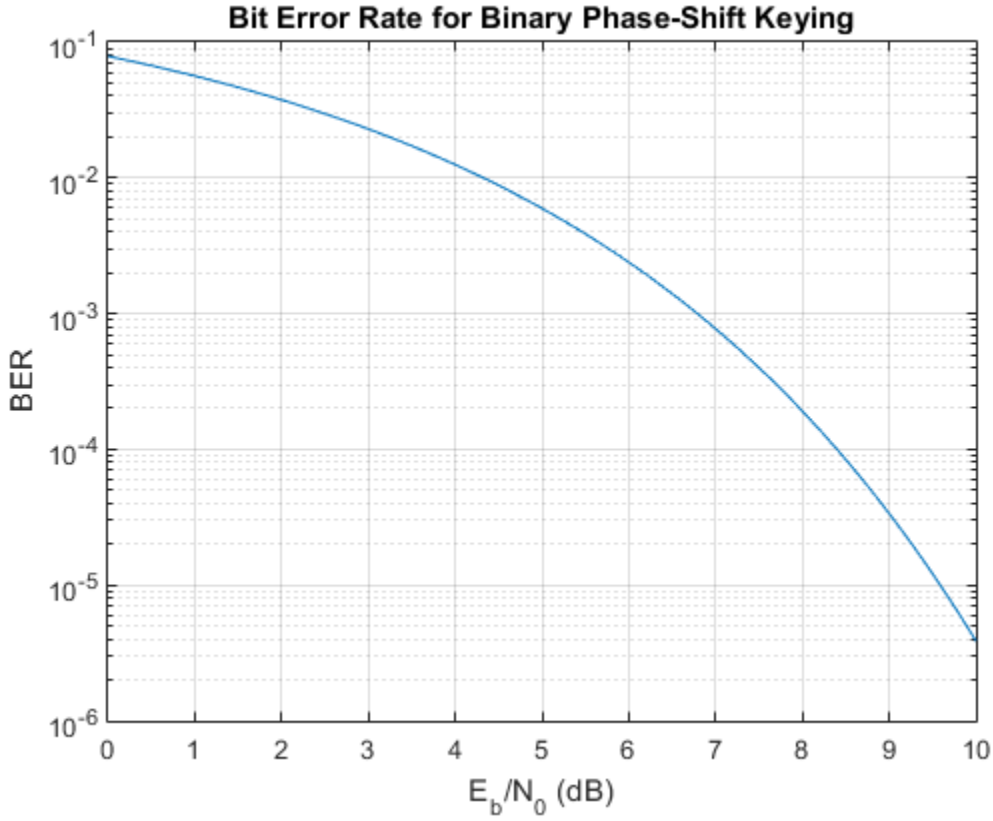
Find Bit Error Rate of Binary Phase-Shift Keying

The bit error rate (BER) of binary phase-shift keying (BPSK), assuming additive white gaussian noise (AWGN), is

$$P_b = \frac{1}{2}\mathrm{erfc}\left(\sqrt{\frac{\mathrm{E_b}}{\mathrm{N_0}}}\right).$$

Plot the BER for BPSK for values of E_b/N_0 from `0dB` to `10dB`.

```
EbNO_dB = 0:0.1:10;
EbNO = 10.^(EbNO_dB/10);
BER = 1/2.*erfc(sqrt(EbNO));
semilogy(EbNO_dB,BER)
grid on
ylabel('BER')
xlabel('E_b/N_O (dB)')
title('Bit Error Rate for Binary Phase-Shift Keying')
```



Avoid Roundoff Errors Using Complementary Error Function

You can use the complementary error function `erfc` in place of `1 - erf(x)` to avoid roundoff errors when `erf(x)` is close to `1`.

Show how to avoid roundoff errors by calculating `1 - erf(10)` using `erfc(10)`. The original calculation returns `0` while `erfc(10)` returns the correct result.

```
1 - erf(10)
erfc(10)
```

```
ans =
```

```
     0

ans =

   2.0885e-45
```

Input Arguments

x — Input

real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. x cannot be sparse.

Data Types: `single` | `double`

More About

Complementary Error Function

The complementary error function of x is defined as

$$\begin{aligned}\mathrm{erfc}(x) &= \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt \\ &= 1-\mathrm{erf}(x).\end{aligned}$$

It is related to the error function as

$$\mathrm{erfc}(x) = 1-\mathrm{erf}(x).$$

Tips

- You can also find the standard normal probability distribution using the Statistics and Machine Learning Toolbox function `normcdf`. The relationship between the error function `erfc` and `normcdf` is

$$\text{normcdf}(x) = \left(\frac{1}{2}\right) \times \text{erfc}\left(\frac{-x}{\sqrt{2}}\right)$$

- For expressions of the form `1 - erfc(x)`, use the error function `erf` instead. This substitution maintains accuracy. When `erfc(x)` is close to `1`, then `1 - erfc(x)` is a small number and might be rounded down to `0`. Instead, replace `1 - erfc(x)` with `erf(x)`.
- For expressions of the form `exp(x^2)*erfc(x)`, use the scaled complementary error function `erfcx` instead. This substitution maintains accuracy by avoiding roundoff errors for large values of `x`.

See Also

`erf` | `erfcinv` | `erfcx` | `erfinv`

Introduced before R2006a

erfcinv

Inverse complementary error function

Syntax

```
erfcinv(x)
```

Description

`erfcinv(x)` returns the value of the "Inverse Complementary Error Function" on page 1-2351 for each element of x. For inputs outside the interval `[0 2]`, `erfcinv` returns `NaN`. Use the `erfcinv` function to replace expressions containing `erfinv(1-x)` for greater accuracy when `x` is close to `1`.

Examples

Find Inverse Complementary Error Function

```
erfcinv(0.3)
```

```
ans =

    0.7329
```

Find the inverse complementary error function of the elements of a vector.

```
V = [-10 0 0.5 1.3 2 Inf];
erfcinv(V)
```

```
ans =

       NaN       Inf    0.4769   -0.2725      -Inf       NaN
```

Find the inverse complementary error function of the elements of a matrix.

```
M = [0.1 1.2; 1 0.9];
erfcinv(M)
```

```
ans =

    1.1631   -0.1791
         0    0.0889
```

Avoid Roundoff Errors Using Inverse Complementary Error Function

You can use the inverse complementary error function `erfcinv` in place of `erfinv(1-x)` to avoid roundoff errors when `x` is close to `0`.

Show how to avoid roundoff by calculating `erfinv(1-x)` using `erfcinv(x)` for `x = 1e-100`. The original calculation returns `Inf` while `erfcinv(x)` returns the correct result.

```
x = 1e-100;
erfinv(1-x)
erfcinv(x)
```

```
ans =

   Inf


ans =

   15.0656
```

Input Arguments

x — Input

real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. x cannot be sparse.

Data Types: `single` | `double`

More About

Inverse Complementary Error Function

The inverse complementary error function `erfcinv(x)` is defined as $\text{erfcinv}(\text{erfc}(x)) = x$.

Tips

- You can also find the inverse standard normal probability distribution using the Statistics and Machine Learning Toolbox function `norminv`. The relationship between the inverse complementary error function `erfcinv` and `norminv` is

  $$\text{norminv}(p) = \left(-\sqrt{2}\right) \times \text{erfcinv}(2p).$$

- For expressions of the form `erfcinv(1-x)`, use the inverse error function `erfinv` instead. This substitution maintains accuracy. When `x` is close to `1`, then `1 - x` is a small number and might be rounded down to `0`. Instead, replace `erfcinv(1-x)` with `erfinv(x)`.

See Also

`erf` | `erfc` | `erfcx` | `erfinv`

Introduced before R2006a

erfcx

Scaled complementary error function

Syntax

```
erfcx(x)
```

Description

`erfcx(x)` returns the value of the "Scaled Complementary Error Function" on page 1-2354 for each element of x. Use the `erfcx` function to replace expressions containing `exp(x^2)*erfc(x)` to avoid underflow or overflow errors.

Examples

Find Scaled Complementary Error Function

```
erfcx(5)
```

```
ans =

    0.1107
```

Find the scaled complementary error function of the elements of a vector.

```
V = [-Inf -1 0 1 10 Inf];
erfcx(V)
```

```
ans =

       Inf    5.0090    1.0000    0.4276    0.0561         0
```

Find the scaled complementary error function of the elements of a matrix.

```
M = [-0.5 15; 3.2 1];
erfcx(M)
```

```
ans =

    1.9524    0.0375
    0.1687    0.4276
```

Avoid Roundoff Errors Using Scaled Complementary Error Function

You can use the scaled complementary error function `erfcx` in place of `exp(x^2)*erfc(x)` to avoid underflow or overflow errors.

Show how to avoid roundoff errors by calculating `exp(35^2)*erfc(35)` using `erfcx(35)`. The original calculation returns `NaN` while `erfcx(35)` returns the correct result.

```
x = 35;
exp(x^2)*erfc(x)
erfcx(x)
```

```
ans =

   NaN


ans =

    0.0161
```

Input Arguments

x — Input

real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. x cannot be sparse.

Data Types: `single` | `double`

More About

Scaled Complementary Error Function

The scaled complementary error function `erfcx(x)` is defined as

$$\mathrm{erfcx}(x) = e^{x^2}\,\mathrm{erfc}(x).$$

For large `X`, `erfcx(X)` is approximately $\left(\frac{1}{\sqrt{\pi}}\right)\frac{1}{x}$.

Tips

- For expressions of the form `exp(-x^2)*erfcx(x)`, use the complementary error function `erfc` instead. This substitution maintains accuracy by avoiding roundoff errors for large values of `x`.

See Also

`erf` | `erfc` | `erfcinv` | `erfinv`

Introduced before R2006a

erfinv

Inverse error function

Syntax

```
erfinv(x)
```

Description

`erfinv(x)` returns the "Inverse Error Function" on page 1-2360 evaluated for each element of x. For inputs outside the interval `[-1 1]`, `erfinv` returns `NaN`.

Examples

Find Inverse Error Function of Value

```
erfinv(0.25)
```

```
ans =

    0.2253
```

For inputs outside `[-1,1]`, `erfinv` returns `NaN`. For `-1` and `1`, `erfinv` returns `-Inf` and `Inf`, respectively.

```
erfinv([-2 -1 1 2])
```

```
ans =

   NaN  -Inf   Inf   NaN
```

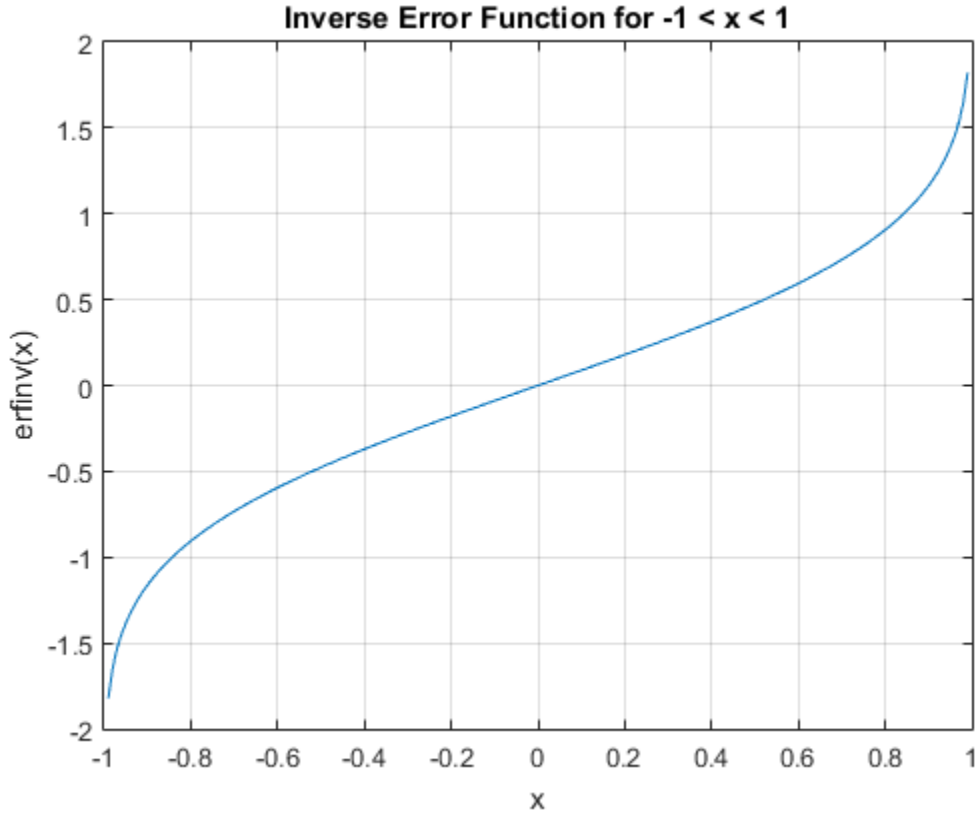
Find the inverse error function of the elements of a matrix.

```
M = [0 -0.5; 0.9 -0.2];
erfinv(M)


ans =

         0   -0.4769
    1.1631   -0.1791
```

Plot the Inverse Error Function

Plot the inverse error function for -1 < x < 1.

```
x = -1:0.01:1;
y = erfinv(x);
plot(x,y)
grid on
xlabel('x')
ylabel('erfinv(x)')
title('Inverse Error Function for -1 < x < 1')
```



Generate Gaussian Distributed Random Numbers

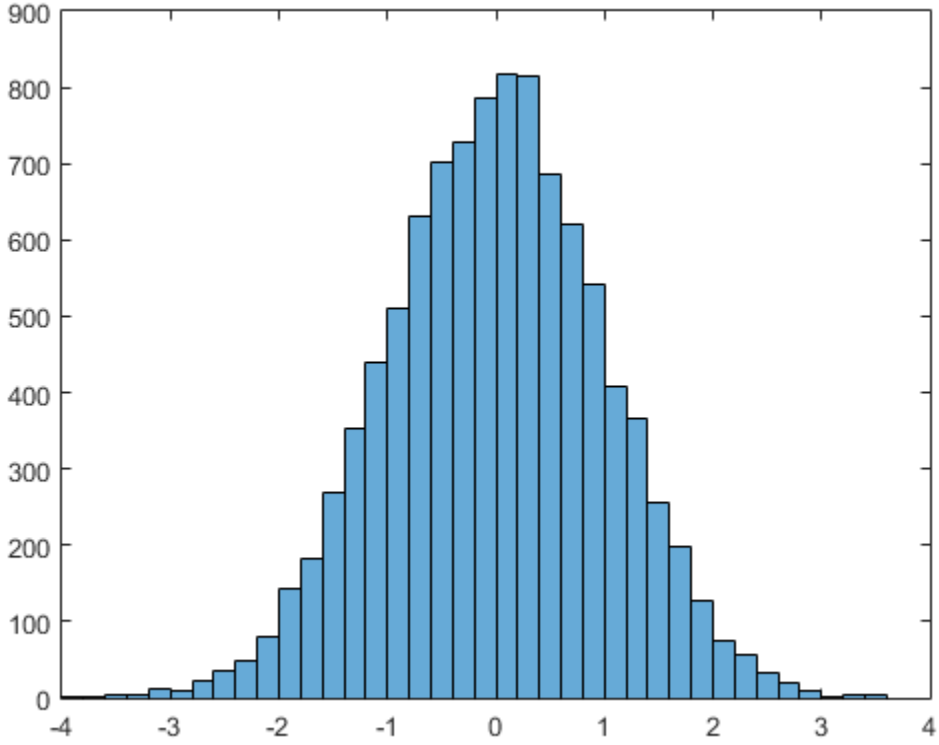
Generate Gaussian distributed random numbers using uniformly distributed random numbers. To convert a uniformly distributed random number x to a Gaussian distributed random number y, use the transform

$$y = \sqrt{2}\mathrm{erf}^{-1}(\mathrm{x}).$$

Note that because `x` has the form `-1 + 2*rand(1,10000)`, you can improve accuracy by using `erfcinv` instead of `erfinv`. For details, see “Tips”.

Generate 10,000 uniformly distributed random numbers on the interval `[-1,1]`. Transform them into Gaussian distributed random numbers. Show that the numbers follow the form of the Gaussian distribution using a histogram plot.

```
rng('default')
x = -1 + 2*rand(1,10000);
y = sqrt(2)*erfinv(x);
h = histogram(y);
```



Input Arguments

x — Input

real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. x cannot be sparse.

Data Types: `single` | `double`

More About

Inverse Error Function

The inverse error function erfinv is defined as the inverse of the error function, such that

$$\mathrm{erfinv}(\mathrm{erf}(x)) = x.$$

Tips

- For expressions of the form `erfinv(1-x)`, use the complementary inverse error function `erfcinv` instead. This substitution maintains accuracy. When `x` is close to `1`, then `1 - x` is a small number and may be rounded down to `0`. Instead, replace `erfinv(1-x)` with `erfcinv(x)`.

See Also

`erf` | `erfc` | `erfcinv` | `erfcx`

Introduced before R2006a

error

Throw error and display message

Syntax

```
error(msg)
error(msg,A1,...,An)
error(msgID,___)

error(errorStruct)
```

Description

`error(msg)` throws an error and displays an error message.

`error(msg,A1,...,An)` displays an error message that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `msg` is converted to one of the values `A1,...,An`.

`error(msgID,___)` includes an error identifier on the exception. The identifier enables you to distinguish errors and to control what happens when MATLAB encounters the errors. You can include any of the input arguments in the previous syntaxes.

`error(errorStruct)` throws an error using the fields in a scalar structure.

Examples

Throw Error

```
msg = 'Error occurred.';
error(msg)
```

```
Error occurred.
```

Throw Error with Formatted Message

Throw a formatted error message with a line break. You must specify more than one input argument with `error` if you want MATLAB to convert special characters (such as

\n) in the error message string. Include information about the class of variable n in the error message.

```
n = 7;
if ~ischar(n)
   error('Error. \nInput must be a char, not a %s.',class(n))
end
```

```
Error.
Input must be a char, not a double.
```

If you only use one input argument with error, then MATLAB does not convert \n to a line break.

```
if ~ischar(n)
   error('Error. \nInput must be a char.')
end
```

```
Error. \nInput must be a char.
```

Throw an error with an identifier.

```
if ~ischar(n)
   error('MyComponent:incorrectType',...
       'Error. \nInput must be a char, not a %s.',class(n))
end
```

```
Error.
Input must be a char, not a double.
```

Use the MException.last to view the last uncaught exception.

```
exception = MException.last
```

```
exception =

  MException with properties:

    identifier: 'MyComponent:incorrectType'
       message: 'Error.
Input must be a char, not a double.'
         cause: {0x1 cell}
```

```
         stack: [0x1 struct]
```

Throw Error Using Structure

Create structure with message and identifier fields. To keep the example simple, do not use the stack field.

```
errorStruct.message = 'Data file not found.';
errorStruct.identifier = 'MyFunction:fileNotFound';
```

```
errorStruct =

       message: 'Data file not found.'
    identifier: 'MyFunction:fileNotFound'
```

Throw the error.

```
error(errorStruct)
```

```
Data file not found.
```

- " Capture Information About Exceptions"

Input Arguments

`msg` — Information about error

string

Information about the error, specified as a string. This message displays as the error message. To format the string, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see "Formatting Strings".

Note: You must specify more than one input argument with `error` if you want MATLAB to convert special characters (such as `\t`, `\n`, `%s`, and `%d`) in the error message string.

Example: `'File not found.'`

`msgID` — Identifier for error

string

Identifier for the error, specified as a string. Use the error identifier to help identify the source of the error or to control a selected subset of the errors in your program.

The error identifier includes a `component` and `mneumonic`. The identifier must always contain a colon and follows this simple format: `component:mnemonic`. The `component` and `mneumonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No whitespace characters can appear anywhere in msgID. For more information, see "Message Identifiers".

Example: `'MATLAB:singularMatrix'`

Example: `'MATLAB:narginchk:notEnoughInputs'`

`A1,...,An` — Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array. This input argument provides the values that correspond to and replace the conversion specifiers in `msg`.

`errorStruct` — Error reporting information

scalar structure

Error reporting information, specified as a scalar structure. The structure must contain at least one of these fields.

`message`	Error message string. For more information, see msg.
`identifier`	Error message identifier. For more information, see msgID.
`stack`	Stack field for the error. When `errorStruct` includes a `stack` field, `error` uses it to set the stack field of the error. When you specify `stack`, use the absolute file name and the entire sequence of functions that nests the function in the stack frame. This string is the same as the string returned by `dbstack('-completenames')`.

More About

Tips

- When you throw an error, MATLAB captures information about it and stores it in a data structure that is an object of the `MException` class. You can access information

in the exception object by using `try/catch`. Or, if your program terminates because of an exception and returns control to the Command Prompt, you can use `MException.last`.

- MATLAB does not cease execution of a program if an error occurs within a `try` block. In this case, MATLAB passes control to the `catch` block.
- If all inputs to `error` are empty, MATLAB does not throw an error.

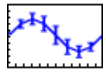
See Also

`MException` | `MException.last` | `assert` | `dbstack` | `errordlg` | `try` | `warning`

Introduced before R2006a

errorbar

Plot error bars along curve



Syntax

```
errorbar(Y,E)
errorbar(X,Y,E)
errorbar(X,Y,L,U)
errorbar(...,LineSpec)
errorbar(ax,...)
h = errorbar(...)
```

Description

Error bars show the confidence intervals of data or the deviation along a curve.

`errorbar(Y,E)` plots `Y` and draws an error bar at each element of `Y`. The error bar is a distance of `E(i)` above and below the curve so that each bar is symmetric and `2*E(i)` long.

`errorbar(X,Y,E)` plots `Y` versus `X` with symmetric error bars `2*E(i)` long. `X`, `Y`, `E` must be the same size. When they are vectors, each error bar is a distance of `E(i)` above and below the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `E(i,j)` above and below the point defined by `(X(i,j),Y(i,j))`.

`errorbar(X,Y,L,U)` plots `X` versus `Y` with error bars `L(i)+U(i)` long specifying the lower and upper error bars. `X`, `Y`, `L`, and `U` must be the same size. When they are vectors, each error bar is a distance of `L(i)` below and `U(i)` above the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `L(i,j)` below and `U(i,j)` above the point defined by `(X(i,j),Y(i,j))`.

`errorbar(...,LineSpec)` uses the color and line style specified by the string `'LineSpec'`. The color is applied to the data line and error bars. The linestyle and marker are applied to the data line only. See `linespec` for examples of styles.

`errorbar(ax,...)` plots into the axes specified by `ax` instead of the current axes.

`h = errorbar(...)` returns handles to the errorbarseries objects created. `errorbar` creates one object for vector input arguments and one object per column for matrix input arguments. See Errorbar Series Properties for more information.

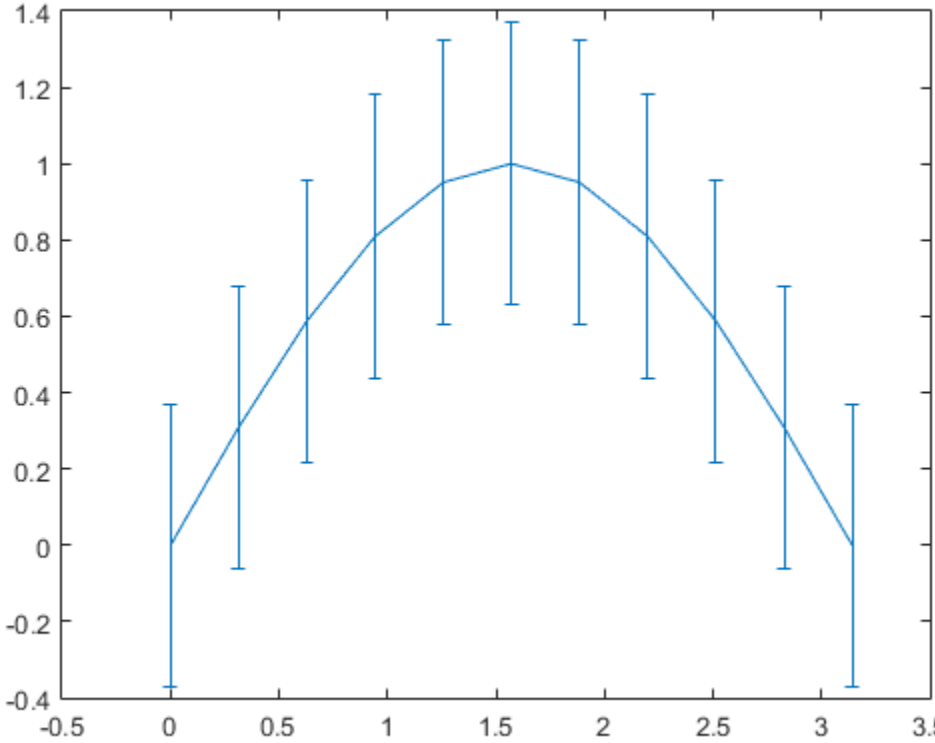
When the arguments are all matrices, `errorbar` draws one line per matrix column. If `X` and `Y` are vectors, they specify one curve.

Examples

Symmetric Error Bars

Draw symmetric error bars that are two standard deviation units in length.

```
x = 0:pi/10:pi;
y = sin(x);
e = std(y)*ones(size(x));

figure
errorbar(x,y,e)
```



Change Error Bar Marker and Color

Load the `count` data set to get the three-column matrix `count` that contains traffic volume for three street locations over the course of a day. Compute the mean of `count` for each row.

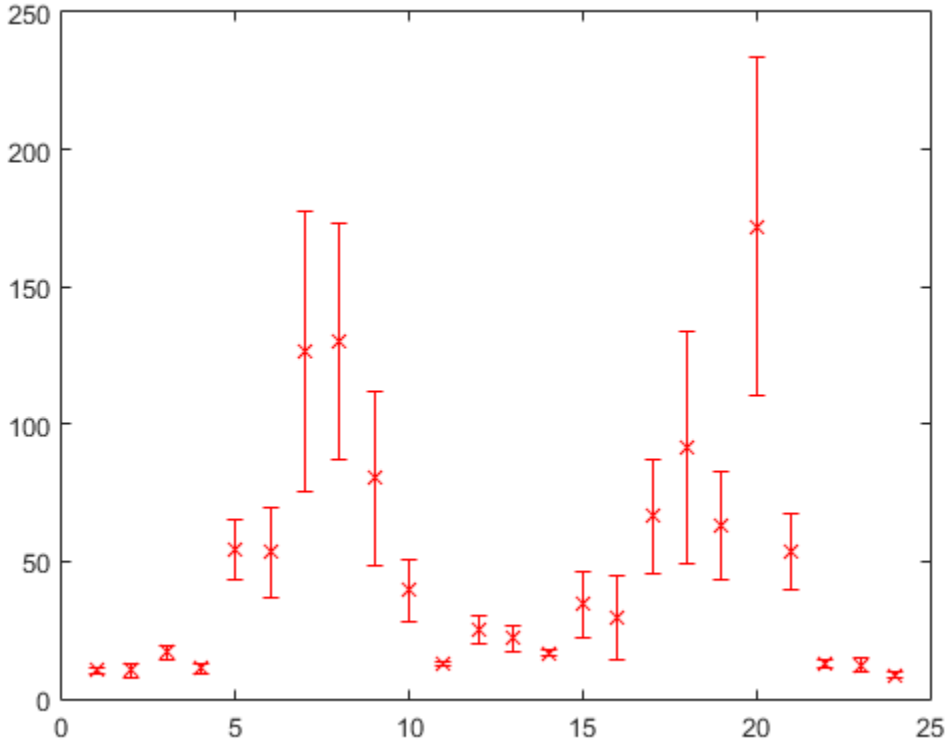
```
load count.dat;
y = mean(count,2);
```

Compute the standard deviation of `count` for each row and normalize by the number of elements in the sample by setting the second input argument to 1.

```
e = std(count,1,2);
```

Plot the computed average traffic volume, `y`, and the computed standard deviations, `e`, for the three street locations. Set the `LineSpec` to specify a red color, cross markers, and no line.

```
figure
errorbar(y,e,'rx')
```



More About

- ConfidenceBounds

See Also

Functions
`corrcoef` | `linespec` | `plot` | `std`

Properties
Errorbar Series Properties

Introduced before R2006a

Errorbar Series Properties

Control errorbar series appearance and behavior

Errorbar series properties control the appearance and behavior of errorbar series object. By changing property values, you can modify certain aspects of the errorbar series.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = errorbar(...);
s = h.LineStyle;
h.LineStyle = ':';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Lines

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- - - - -
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	_._._._.-
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the `Color` as `'none'`, then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines

to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the errorbar series object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)

String	Marker Symbol
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Data

`XData` — x values

`[ ]` (default) | vector

x values, specified as a vector. The input argument `X` to the `errorbar` function sets the *x* values. If you do not specify `X`, then `errorbar` uses the indices of `YData` as the *x* values. `XData` and `YData` must have equal lengths.

Example: `1:10`

`YData` — y values

`[ ]` (default) | vector

y values, specified as a vector. The input argument `Y` to the `errorbar` function sets the *y* values. `XData` and `YData` must have equal lengths.

`LData` — Errorbar lengths below data points

`[ ]` (default) | vector

Errorbar lengths below the data points, specified as a vector with length equal to `XData` and `YData`. Specify the values in data units.

Example: `1:10`

`UData` — Errorbar lengths above data points

`[ ]` (default) | vector

Errorbar lengths above the data points, specified as a vector with length equal to `XData` and `YData`. Specify the values in data units.

Example: `1:10`

`XDataSource` — Variable linked to `XData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`LDataSource` — Variable linked to `LData`

`''` (default) | string containing MATLAB workspace variable

Variable linked to `LData`, specified as a string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `LData`.

By default, there is no linked variable so the value is an empty string, `''`. If you change the variable for this property, then MATLAB does not update the `LData` values. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`UDataSource` — Variable linked to `UData`

`''` (default) | string containing MATLAB workspace variable

Variable linked to `UData`, specified as a string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `UData`.

By default, there is no linked variable so the value is an empty string, `''`. If you change the variable for this property, then MATLAB does not update the `UData` values. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`XDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData`.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

Visibility

`Visible` — Visibility of errorbar series

`'on'` (default) | `'off'`

Visibility of errorbar series, specified as one of these values:

- `'on'` — Display the errorbar series.
- `'off'` — Hide the errorbar series without deleting it. You still can access the properties of an invisible errorbar series object.

`Clipping` — Clipping of errorbar series to axes limits

`'on'` (default) | `'off'`

Clipping of errorbar series to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the errorbar series that are outside the axes limits.
- `'off'` — Display the entire errorbar series, even if parts of it appear outside the axes limits. Parts of the errorbar series might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the errorbar series that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'errorbar'`

Type of graphics object, returned as `'errorbar'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

`Tag` — Tag to associate with errorbar series

`''` (default) | string

Tag to associate with the errorbar series, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

Data Types: `char`

`UserData` — Data to associate with errorbar series

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the errorbar series object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the errorbar series.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.

- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the errorbar series object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the errorbar series from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the errorbar series object in the legend as one entry (default).
   - `'off'` — Do not include the errorbar series object in the legend.
   - `'children'` — Include only children of the errorbar series object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of errorbar series

axes object | group object | transform object

Parent of errorbar series, specified as an axes, group, or transform object.

`Children` — Children of errorbar series

empty `GraphicsPlaceholder` array

The errorbar series has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of errorbar series object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The errorbar series object handle is always visible.
- `'off'` — The errorbar series object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The errorbar series object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the errorbar series at the command-line, but allows callback functions to access it.

If the errorbar series object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the errorbar series. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The errorbar series object — You can access properties of the errorbar series object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the errorbar series. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the errorbar series when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the errorbar series.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the errorbar series that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the errorbar series responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the errorbar series passes the click to the object below it in the current view of the figure window. The HitTest property of the errorbar series has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the errorbar series. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the errorbar series that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the errorbar series object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for errorbar series

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for errorbar series, specified as one of these values:

- `'off'` — Click the errorbar series plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the errorbar series plot to select it, that is, anywhere within the rectangle that encloses the errorbar series plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the errorbar series is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.

- If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the errorbar series tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the errorbar series. Setting the `CreateFcn` property on an existing errorbar series has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during errorbar series creation. MATLAB executes the callback after creating the errorbar series and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The errorbar series object — You can access properties of the errorbar series object from within the callback function. You also can access the errorbar series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the errorbar series. MATLAB executes the callback before destroying the errorbar series so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The errorbar series object — You can access properties of the errorbar series object from within the callback function. You also can access the errorbar series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of errorbar series

`'off'` (default) | `'on'`

Deletion status of errorbar series, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the errorbar series begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the errorbar series no longer exists.

Check the value of the BeingDeleted property to verify that the errorbar series is not about to be deleted before querying or modifying it.

See Also

`errorbar`

More About

- "Access Property Values"
- "Graphics Object Properties"

errordlg

Create error dialog box

Syntax

```
h = errordlg
h = errordlg(errorstring)
h = errordlg(errorstring,dlgname)
h = errordlg(errorstring,dlgname,createmode)
```

Description

`h = errordlg` creates and displays a dialog box with title `Error Dialog` that contains the string, `This is the default error string`. The `errordlg` function returns the handle of the dialog box in `h`.

`h = errordlg(errorstring)` displays a dialog box with title `Error Dialog` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname)` displays a dialog box with title`dlgname` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname,createmode)` specifies whether the error dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `errorstring` and `dlgname`. The *`createmode`* argument can be a string or a structure.

If *`createmode`* is a string, it must be one of the values shown in the following table.

createmode Value	Description
`modal`	Replaces the error dialog box having the specified `Title`, that was last created or clicked on, with a modal error dialog box as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.
`nonmodal` (default)	Creates a new nonmodal error dialog box with the specified parameters. Existing error dialog boxes with the same title are not deleted.

createmode Value	Description
`replace`	Replaces the error dialog box having the specified `Title`, that was last created or clicked on, with a nonmodal error dialog box as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

Note: A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

Modal dialogs (created using `errordlg`, `msgbox`, or `warndlg`) replace any existing dialogs created with these functions that also have the same name.

For more information about modal dialog boxes, see WindowStyle in Figure Properties.

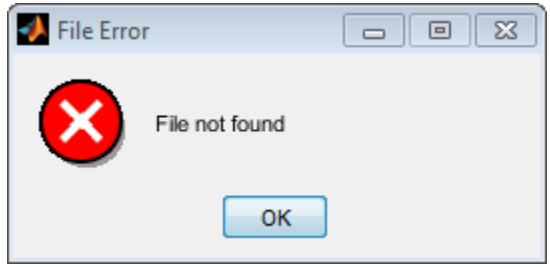
If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings `'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

Examples

This code,

```
errordlg('File not found','File Error');
```

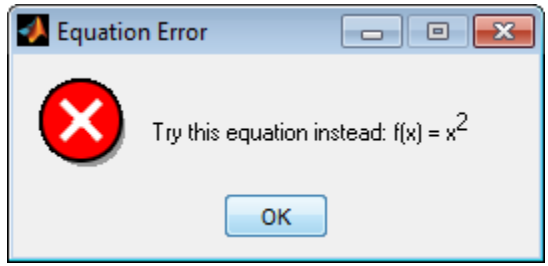
displays this dialog box:



This code,

```
mode = struct('WindowStyle','nonmodal',...
       'Interpreter','tex');
h = errordlg('Try this equation instead: f(x) = x^2',...
    'Equation Error', mode);
```

displays this dialog box:



More About

Tips

MATLAB sizes the dialog box to fit the string `'errorstring'`. The error dialog box has an **OK** push button and remains on the screen until you press the **OK** button or the **Return** key. After pressing the button, the error dialog box disappears.

The appearance of the dialog box depends on the platform you use.

See Also

`dialog` | `helpdlg` | `inputdlg` | `listdlg` | `msgbox` | `questdlg` | `warndlg` | `figure` | `uiwait` | `uiresume`

Introduced before R2006a

etime

Time elapsed between date vectors

Syntax

```
e = etime(t2,t1)
```

Description

`e = etime(t2,t1)` returns the number of seconds between two date vectors or matrices of date vectors, `t1` and `t2`.

Examples

Compute Elapsed Time

Compute the time elapsed between a specific time and the current time, to 0.01-second accuracy.

Define the initial date and time and convert to date vector form.

```
format shortg
str = 'March 28, 2012 11:51:00';
t1 = datevec(str,'mmmm dd, yyyy HH:MM:SS')
```

```
t1 =

        2012           3          28          11          51           0
```

Determine the current date and time.

```
t2 = clock
```

```
t2 =
```

```
        2015           2          23          10          10      12.594
```

The `clock` function returns the current date and time as a date vector.

Use `etime` to compute the number of seconds between `t1` and `t2`.

```
e = etime(t2,t1)
```

```
e =

   9.1751e+07
```

Input Arguments

`t2,t1` — Date vectors

1-by-6 vector | `m`-by-6 matrix

Date vectors, specified as 1-by-6 vectors or `m`-by-6 matrices containing `m` full date vectors in the format:`[Year Month Day Hour Minute Second]`.

Example: `[2012 03 27 11 50 01]`

Data Types: `double`

More About

Tips

- To time the duration of an event, use the `timeit` or `tic` and `toc` functions instead of `clock` and `etime`. The `clock` function is based on the system time, which can be adjusted periodically by the operating system, and thus might not be reliable in time comparison operations.

Algorithms

`etime` does not account for the following:

- Leap seconds.

- Daylight savings time adjustments.
- Differences in time zones.

See Also

`clock` | `cputime` | `now` | `tic` | `timeit` | `toc`

Introduced before R2006a

etree

Elimination tree

Syntax

```
p = etree(A)
p = etree(A,'col')
p = etree(A,'sym')
[p,q] = etree(...)
```

Description

`p = etree(A)` returns an elimination tree for the square symmetric matrix whose upper triangle is that of `A`. `p(j)` is the parent of column `j` in the tree, or `0` if `j` is a root.

`p = etree(A,'col')` returns the elimination tree of `A'*A`.

`p = etree(A,'sym')` is the same as `p = etree(A)`.

`[p,q] = etree(...)` also returns a postorder permutation `q` of the tree.

See Also

`treelayout` | `treeplot` | `etreeplot`

Introduced before R2006a

etreeplot

Plot elimination tree

Syntax

```
etreeplot(A)
etreeplot(A,nodeSpec,edgeSpec)
```

Description

`etreeplot(A)` plots the elimination tree of `A` (or `A+A'`, if non-symmetric).

`etreeplot(A,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

See Also

`etree` | `treeplot` | `treelayout`

Introduced before R2006a

eval

Execute MATLAB expression in text string

Syntax

```
eval(expression)
[output1,...,outputN] = eval(expression)
```

Description

`eval(expression)` evaluates the MATLAB code in the string `expression`. If you use `eval` within an anonymous function, nested function, or function that contains a nested function, the evaluated `expression` cannot create a variable.

`[output1,...,outputN] = eval(expression)` stores output from `expression` in the specified variables.

Input Arguments

`expression`

String that contains a valid MATLAB expression.

To include a numeric value in the expression, convert it to a string with `int2str`, `num2str`, or `sprintf`.

Output Arguments

`output1,...,outputN`

Outputs from the evaluated `expression`.

Examples

Variable Name Evaluation

Select a matrix to plot at runtime.

This example requires that you have a matrix in the current workspace. For example:

```
aMatrix = magic(5);
```

Interactively request the name of a matrix to plot, and call `eval` to use its value.

```
expression = input('Enter the name of a matrix: ','s');
if (exist(expression,'var'))
    mesh(eval(expression))
end
```

If you type `aMatrix` at the input prompt, this code creates a mesh plot of `magic(5)`.

More About

Tips

- Many common uses of the `eval` function are less efficient and are more difficult to read and debug than other MATLAB functions and language constructs. For more information, see “Alternatives to the eval Function”.
- Whenever possible, do not include output arguments within the input to the `eval` function, such as `eval(['output = ',expression])`. The preferred syntax,

  ```
  output = eval(expression)
  ```

  allows the MATLAB parser to perform stricter checks on your code, preventing untrapped errors and other unexpected behavior.

- “Alternatives to the eval Function”
- “Variables in Nested and Anonymous Functions”

See Also

`assignin` | `evalc` | `evalin` | `feval` | `try`

Introduced before R2006a

evalc

Evaluate MATLAB expression with capture

Syntax

```
T = evalc(expression)
[T,output1,...,outputN] = evalc(expression)
```

Description

`T = evalc(expression)` is the same as `eval(expression)` except that anything that would normally be written to the command window, except for error messages, is captured and returned in the character array `T` (lines in `T` are separated by `\n` characters).

`[T,output1,...,outputN] = evalc(expression)` is the same as `[output1,...,outputN] = eval(expression)` except that any output is captured into `T`.

Input Arguments

`expression`

String that contains a valid MATLAB expression.

To include a numeric value in the expression, convert it to a string with `int2str`, `num2str`, or `sprintf`.

Output Arguments

`T`

Output normally written to the command window during the evaluation of `expression`, except for error messages, returned in a character array. The lines in `T` are separated by `\n` characters.

`output1,...,outputN`

Outputs from the evaluated `expression`.

More About

Tips

When you are using `evalc`, functions `diary`, `more`, and `input` are disabled.

See Also

`eval` | `evalin` | `assignin` | `more` | `feval` | `diary` | `input`

Introduced before R2006a

evalin

Execute MATLAB expression in specified workspace

Syntax

```
evalin(ws, expression)
[a1, a2, a3, ...] = evalin(ws, expression)
```

Description

`evalin(ws, expression)` executes *`expression`*, a string containing any valid MATLAB expression, in the context of the workspace `ws`. `ws` can have a value of `'base'` or `'caller'` to denote the MATLAB base workspace or the workspace of the caller function. You can construct *`expression`* by concatenating substrings and variables inside square brackets:

```
expression = [string1, int2str(var), string2,...]
```

`[a1, a2, a3, ...] = evalin(ws, expression)` executes *`expression`* and returns the results in the specified output variables. Using the `evalin` output argument list is recommended over including the output arguments in the expression string:

```
evalin(ws,'[a1, a2, a3, ...] = function(var)')
```

The above syntax avoids strict checking by the MATLAB parser and can produce untrapped errors and other unexpected behavior.

Examples

This example extracts the value of the variable `var` in the MATLAB base workspace and captures the value in the local variable `v`:

```
v = evalin('base', 'var');
```

Limitation

`evalin` cannot be used recursively to evaluate an expression. For example, a sequence of the form `evalin('caller', 'evalin(''caller'', ''x'')')` doesn't work.

More About

Tips

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the currently running function. Note that the base and caller workspaces are equivalent in the context of a function that is invoked from the MATLAB command line.

`evalin('caller', expression)` finds only *variables* in the caller's workspace; it does not find *functions* in the caller. For this reason, you cannot use `evalin` to construct a handle to a function that is defined in the caller.

If you use `evalin('caller', expression)` in the MATLAB debugger after having changed your local workspace context with `dbup` or `dbdown`, MATLAB evaluates the expression in the context of the function that is one level up in the stack from your current workspace context.

See Also

`assignin` | `eval` | `evalc` | `feval` | `try`

Introduced before R2006a

event.EventData

Base class for all data objects passed to event listeners

Description

The `event.EventData` class defines the event data objects passed to listeners. When you trigger an event using the `notify` handle class method, MATLAB assigns values to the properties of the `event.EventData` object that is passed to the listener callback function (the event handler).

If you want to provide additional information to event listeners, you can do so by subclassing `event.EventData`. See "Define Event-Specific Data" for more information.

Note: Subclasses of `event.EventData` must set the class `ConstructOnLoad` attribute.

Properties

The `event.EventData` class defines two properties and no methods:

- `EventName` — The name of the event described by this data object.
- `Source` — The source object whose class defines the event described by the data object.

More About

- "Learn to Use Events and Listeners"
- "Class with Custom Event Data"

See Also

`notify` | `event.PropertyEvent`

event.listener

Class defining listener objects

Syntax

```
lh = event.listener(Hobj,'EventName',@CallbackFunction)
```

Description

The `event.listener` class defines listener objects. Listener objects respond to the specified event and identify the callback function to invoke when the event is triggered.

`lh = event.listener(Hobj,'EventName',@CallbackFunction)` creates an `event.listener` object, `lh`, for the event named in *`EventName`*, on the specified object, `Hobj`.

If `Hobj` is an array of object handles, the listener responds to the named event on any of the objects referenced in the array.

The listener callback function must accept at least two input arguments. For example,

```
function CallbackFunction(source,eventData)
   ...
end
```

where `source` is the object that is the source of the event and `eventData` is an `event.EventData` object.

The `event.listener` class is a `handle` class.

Limiting Listener Lifecycle

Generally, you create a listener object using `addlistener`. However, you can call the `event.listener` constructor directly to create a listener. When you use the `event.listener` constructor, the listener's lifecycle is not tied to the object(s) being listened to—once the listener object goes out of scope, the listener no longer exists. See “Control Listener Lifecycle” for more information on creating listener objects.

Removing a Listener

If you call `delete(lh)` on the listener object, the listener ceases to exist, which means the event no longer causes the listener callback function to execute.

Disabling a Listener

You can enable or disable a listener by setting the value of the listener's `Enabled` property (see Properties table below).

More Information on Events and Listeners

See “Events” for more information and examples of how to use events and listeners.

Properties

Property	Purpose
`Source`	Cell array of source objects
`EventName`	Name of the event
`Callback`	Function to execute when the event is triggered and the `Enabled` property is set to `true`
`Enabled`	The callback executes when the event occurs if and only if `Enabled` is set to `true` (the default).
`Recursive`	When `false` (the default), this listener does not execute recursively. Therefore, if the callback triggers its own event, the listener does not execute again. When `true`, the listener callback can cause the same event that triggered the callback. This scheme can lead to infinite recursion, which ends when the MATLAB recursion limit eventually triggers an error.

See Also

`delete` | `addlistener` | `event.proplistener`

event.PropertyEvent

Data for property events

Description

The `event.PropertyEvent` class defines the data objects passed to listeners of the `meta.property` events:

- `PreGet`
- `PostGet`
- `PreSet`
- `PostSet`

`event.PropertyEvent` is a sealed subclass of `event.EventData` (i.e., you cannot subclass `event.PropertyEvent`).

Properties

`event.PropertyEvent` inherits the first two properties from the `event.EventData`, and defines one new property:

- `EventName` — One of the four event names listed in the Description section
- `Source` — `meta.property` object that triggers the event
- `AffectedObject` — The object whose property is affected.

More About

- "Listen for Changes to Property Values"

See Also

`meta.property` | `event.EventData`

event.proplistener

Define listener object for property events

Syntax

```
lh =
event.proplistener(Hobj,Properties,'PropEvent',@CallbackFunction)
```

Description

`lh = event.proplistener(Hobj,Properties,'PropEvent',@CallbackFunction)` creates a property listener object for one or more properties on the specified object.

- `Hobj` — handle of object whose property or properties are to be listened to. If `Hobj` is an array, the listener responds to the named event on all objects in the array.
- `Properties` — an object array or a cell array of `meta.property` object handles representing the properties to which you want to listen.
- `PropEvent` — must be one of the strings: `PreSet`, `PostSet`, `PreGet`, `PostGet`
- `@CallbackFunction` — function handle to the callback function that executes when the event occurs.

The `event.proplistener` class defines property event listener objects. It is a subclass of the `event.listener` class and adds one property to those defined by `event.listener`:

- `Object` — Cell array of objects whose property events are being listened to.

You can call the `event.proplistener` constructor instead of calling `addlistener` to create a property listener. However, when you do not use `addlistener`, the listener's lifecycle is not tied to the object(s) being listened to.

See “Listen for Changes to Property Values”.

See “Getting Information About Properties” for more information on using `meta.property` objects.

See Also

`event.listener` | `addlistener`

eventlisteners

List event handler functions associated with COM object events

Syntax

```
info = eventlisteners(h)
```

Description

`info = eventlisteners(h)` lists the events and their event handler routines registered with COM object `h`. The function returns a cell array of strings `info`, with each row containing the name of a registered event and the handler routine for that event. If the object has no registered events, `eventlisteners` returns an empty cell array. You can register events either when you create the control, using `actxcontrol`, or at any time afterward, using `registerevent`.

COM functions are available on Microsoft Windows systems only.

Examples

This example manages events for the MATLAB control, `mwsamp`.

```
f = figure('position',[100 200 200 200]);
% Create an mwsamp control and
% register the Click event
h = actxcontrol('mwsamp.mwsampctrl.2',...
    [0 0 200 200],f,{'Click' 'myclick'});
eventlisteners(h)
```

```
ans =
    'Click'    'myclick'
```

MATLAB displays the event name and its event handler, `myclick`.

Register two more events, `DblClick` and `MouseDown`:

```
registerevent(h,{'DblClick', 'my2click'; 'MouseDown' 'mymoused'});
```

```
eventlisteners(h)

ans =
    'Click'        'myclick'
    'Dblclick'     'my2click'
    'Mousedown'    'mymoused'
```

MATLAB displays all event names and handlers.

Unregister all events for the control:

```
unregisterallevents(h)
eventlisteners(h)

ans =
     {}
```

MATLAB displays an empty cell array, indicating the control has no registered events.

See Also

`events (COM)` | `registerevent` | `unregisterevent` | `unregisterallevents` | `isevent` | `actxcontrol`

Introduced before R2006a

events

Event names

Syntax

```
events('classname')
events(obj)
e = events(...)
```

Description

`events('classname')` displays the names of the public events for the MATLAB class *`classname`*, including events inherited from superclasses.

`events(obj)` `obj` is a scalar or array of objects of a MATLAB class.

`e = events(...)` returns the event names in a cell array of strings.

An event is public when the value of its `ListenAccess` attribute is `public` and its `Hidden` attribute value is `false` (default values for both attributes). See "Event Attributes" for a complete list of attributes.

`events` is also a MATLAB class-definition keyword. See `classdef` for more information on class definition keywords.

Examples

Get the names of the public events for the `handle` class:

```
events('handle')
Events for class handle:

    ObjectBeingDestroyed
```

See Also

`properties` | `methods`

Introduced before R2006a

events (COM)

List of events COM object can trigger

Syntax

```
S = events(h)
```

Description

`S = events(h)` returns structure array, `S`, containing all events, both registered and unregistered, known to the COM object, and the function prototype used when calling the event handler routine. For each array element, the structure field is the event name and the contents of that field is the function prototype for that event's handler.

COM functions are available on Microsoft Windows systems only.

Examples

List Control Events Example

Create an `mwsamp` control and list all events.

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f);
events(h)
```

```
Click = void Click()
DblClick = void DblClick()
MouseDown = void MouseDown(int16 Button, int16 Shift,
  Variant x, Variant y)
Event_Args = void Event_Args(int16 typeshort, int32 typelong,
  double typedouble, string typestring, bool typebool)
```

Assign the output to a variable and get one field of the returned structure:

```
ev = events(h);
```

```
ev.MouseDown

ans =
  void MouseDown(int16 Button, int16 Shift, Variant x, Variant y)
```

List Workbook Events Example

Open a Microsoft Excel application and list all events for a `Workbook` object.

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = Add(wbs);
events(wb)
```

The MATLAB software displays all events supported by the `Workbook` object.

See Also

`isevent` | `eventlisteners` | `registerevent` | `unregisterevent` | `unregisterallevents`

exceltime

Convert MATLAB datetime to Excel date number

Syntax

```
e = exceltime(t)
e = exceltime(t,dateType)
```

Description

`e = exceltime(t)` returns a `double` array containing Excel serial date numbers equivalent to the datetime values in `t`. Excel serial date numbers are the number of days and fractional days since 0-January-1900 00:00:00, and do not take into account time zone and leap seconds.

`e = exceltime(t,dateType)` returns the type of Excel serial date numbers specified by `dateType`. For example, you can convert datetime values to the number of days since 1-January-1904 00:00:00.

Examples

Convert Datetime Array to Excel Date Numbers

Create a `datetime` array. Then, convert the dates to the equivalent Excel® serial date numbers.

```
t = datetime('now') + calmonths(1:3)
```

```
t =

   23-Mar-2015 10:04:20   23-Apr-2015 10:04:20   23-May-2015 10:04:20
```

```
e = exceltime(t)
```

```
e =

   1.0e+04 *

    4.2086    4.2117    4.2147
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`dateType` — Type of Excel serial date numbers

`'1900'` (default) | `'1904'`

Type of Excel serial date numbers, specified as either `'1900'` or `'1904'`.

- If `dateType` is `'1900'`, then `exceltime` converts the datetime values in `t` to the equivalent the number of days and fractional days since 0-January-1900 00:00:00.
- If `dateType` is `'1904'`, then `exceltime` converts the datetime values in `t` to the equivalent the number of days and fractional days since 1-January-1904 00:00:00.

`exceltime` does not account for time zone.

Example:

Output Arguments

`e` — Excel serial date numbers

scalar | vector | matrix | multidimensional array

Excel serial date numbers, returned as a scalar, vector, matrix, or multidimensional array of type `double`. Excel serial date numbers are not defined prior to their epoch (0-January-1900 or 1-January-1904). Excel serial date numbers treat 1900 as a leap year. Therefore, dates after February 28, 1900 are offset by one day relative to MATLAB serial date numbers, and there is a discontinuity of one day between February 28, 1900 and March 1, 1900.

See Also

`datenum` | `datetime` | `juliandate` | `posixtime` | `yyyymmdd`

Introduced in R2014b

Execute

Execute MATLAB command in Automation server

Syntax

IDL Method Signature

```
BSTR Execute([in] BSTR command)
```

Microsoft Visual Basic Client

```
Execute(command As String) As String
```

MATLAB Client

```
result = Execute(h,'command')
result = invoke(h,'Execute','command')
```

Description

The `Execute` function executes the MATLAB statement specified by the string `command` in the MATLAB Automation server attached to handle `h`.

The server returns output from the command in the string, `result`. The `result` string also contains any MATLAB warning or error messages.

If you terminate the MATLAB command string with a semicolon and there are no warnings or error messages, `result` might be returned empty.

COM functions are available on Microsoft Windows systems only.

Visual Basic .NET Examples

Create a 6-by-6 matrix in the MATLAB server, remove rows 4-6, and return the results to the client.

```
Dim Matlab As Object
Dim server_version As String
Matlab = CreateObject("matlab.application")
MatLab.PutWorkspaceData("A","base",rand(6))
Matlab.Execute("A(4:6,:) = [];")
Matlab.GetWorkspaceData("A","base",B)
```

More About

Tips

If you want to display output from `Execute` in the client window, specify an output variable, `result`.

If there is an error, the `Execute` function returns the MATLAB error message with the characters `???` prepended to the text.

See Also

`Feval` | `GetFullMatrix` | `PutFullMatrix`

Introduced before R2006a

exifread

Read EXIF information from JPEG and TIFF image files

Syntax

Note: `exifread` has been removed. Use `imfinfo` instead.

```
output = exifread(filename)
```

Description

`output = exifread(filename)` reads the Exchangeable Image File Format (EXIF) data from the file specified by the string `filename`. `filename` must specify a JPEG or TIFF image file. `output` is a structure containing metadata values about the image or images in `imagefile`.

Note `exifread` returns all EXIF tags and does not process them in any way.

EXIF is a standard used by digital camera manufacturers to store information in the image file, such as, the make and model of a camera, the time the picture was taken and digitized, the resolution of the image, exposure time, and focal length. For more information about EXIF and the meaning of metadata attributes, see http://www.exif.org/.

See Also

`imfinfo` | `imread`

Introduced before R2006a

exist

Check existence of variable, function, folder, or class

Syntax

```
exist name
exist name kind
A = exist('name','kind')
```

Description

`exist name` returns the status of `name`:

0	`name` does not exist.
1	`name` is a variable in the workspace.
2	One of the following is true: • `name` exists on your MATLAB search path as a file with extension `.m`. • `name` is the name of an ordinary file on your MATLAB search path. • `name` is the full pathname to any file.
3	`name` exists as a MEX-file on your MATLAB search path.
4	`name` exists as a Simulink model or library file on your MATLAB search path.
5	`name` is a built-in MATLAB function.
6	`name` is a P-file on your MATLAB search path.
7	`name` is a folder.
8	`name` is a class. (`exist` returns 0 for Java classes if you start MATLAB with the `-nojvm` option.)

If `name` is a class, then `exist('name')` returns an `8`. However, if `name` is a class file, then `exist('name')` returns a `2`.

If a file or folder is not on the search path, then `name` must specify either a full pathname, a partial pathname relative to `MATLABPATH`, a partial pathname relative to your current folder, or the file or folder must reside in your current working folder.

If `name` specifies a filename, that filename may include an extension to preclude conflicting with other similar filenames. For example, `exist('file.ext')`.

`exist name kind` returns the status of `name` for the specified *`kind`*. If `name` of type *`kind`* does not exist, it returns `0`. The *`kind`* argument may be one of the following:

`builtin`	Checks only for built-in functions.
`class`	Checks only for classes.
`dir`	Checks only for folders.
`file`	Checks only for files or folders.
`var`	Checks only for variables.

If you do not specify a *`kind`* argument, and `name` belongs to more than one of these categories, `exist` returns one value according to the order of evaluation shown in the table below. For example, if `name` matches both a folder and a file that defines a MATLAB function, `exist` returns `7`, identifying it as a folder.

Order of Evaluation	Return Value	Type of Entity
1	1	Variable
2	5	Built-in
3	7	Folder
4	3	MEX-file
5	6	P-file
6	2	MATLAB function
7	4	SLX or MDL-file
8	8	Class

`A = exist('name','kind')` is the function form of the syntax.

Examples

Check Existence of Workspace Variable

Create a variable named |testresults, and then check its existence in the workspace.

```
testresults = magic(5);
exist testresults var
```

```
ans =

     1
```

A variable named `testresults` exists in the workspace.

Check if MATLAB Function is Built-In Function

Check whether the `plot` function is a built-in function or a file.

```
A = exist('plot')
```

```
A =

     5
```

This indicates that `plot` is a built-in MATLAB function.

More About

Tips

If `name` specifies a filename, MATLAB attempts to locate the file, examines the filename extension, and determines the value to return based on the extension alone. MATLAB does not examine the contents or internal structure of the file.

You can specify a partial path to a folder or file. A partial pathname is a pathname relative to the MATLAB path that contains only the trailing one or more components of the full pathname. For example, both of the following commands return `2`, identifying `mkdir.m` as a MATLAB function. The first uses a partial pathname:

```
exist('matlab/general/mkdir.m')
exist([matlabroot '/toolbox/matlab/general/mkdir.m'])
```

To check for the existence of more than one variable, use the `ismember` function. For example,

```
a = 5.83;
c = 'teststring';
ismember({'a','b','c'},who)

ans =

     1     0     1
```

See Also

assignin | dir | evalin | help | inmem | computer | isfield | isempty | lookfor | mfilename | what | which | who

Introduced before R2006a

exit

Terminate MATLAB program (same as `quit`)

Alternatives

As an alternative to the `exit` function, click the Close box in the MATLAB desktop.

Syntax

```
exit
exit(code)
```

Description

`exit` terminates the current session of MATLAB after running `finish.m`, if the file `finish.m` exists. It performs the same as `quit` and takes the same termination options, such as **`force`**. Call `exit` from the MATLAB command prompt. To interrupt a MATLAB command, see "Stop Execution".

`exit(code)` returns exit code when calling a MATLAB command from the system command line.

More About

- "Stop Execution"

See Also

`quit` | `finish`

Introduced before R2006a

exp

Exponential

Syntax

```
Y = exp(X)
```

Description

`Y = exp(X)` returns the exponential e^x for each element in array `X`. For complex elements $z = x + iy$, it returns the complex exponential

$$e^z = e^x (\cos y + i \sin y).$$

Use `expm` to compute a matrix exponential.

Examples

Numeric Representation of e

Calculate the exponential of 1, which is Euler's number, *e*.

```
exp(1)
```

```
ans =

    2.7183
```

Euler's Identity

Euler's identity is the equality $e^{i\pi} + 1 = 0$.

Compute the value of $e^{i\pi}$.

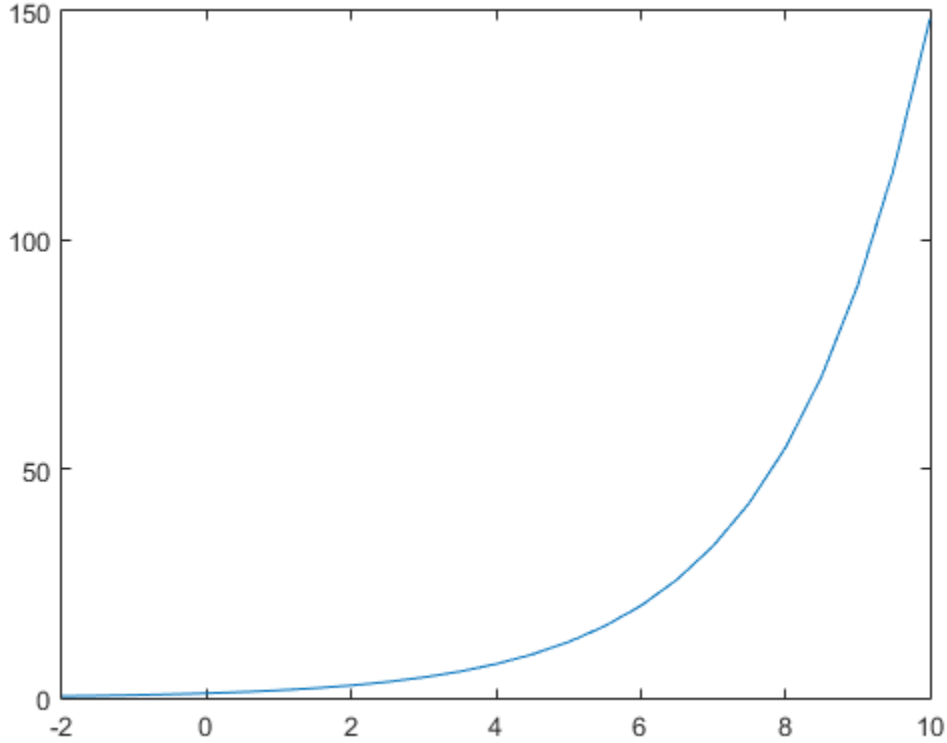
```
Y = exp(1i*pi)
```

```
Y =

  -1.0000 + 0.0000i
```

Plot Exponential Function

Plot $y = e^{x/2}$ for `x` values in the range $[-2, 10]$.

```
X = -2:0.5:10;
Y = exp(X/2);
plot(X,Y)
```



Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Exponential values

scalar | vector | matrix | multidimensional array

Exponential values, returned as a scalar, vector, matrix, or multidimensional array.

For real values of `X` in the interval (`-Inf`, `Inf`), `Y` is in the interval (`0`,`Inf`). For complex values of `X`, `Y` is complex. The data type of `Y` is the same as that of `X`.

See Also

`expint` | `expm` | `expm1` | `log` | `log10` | `mpower` | `power`

Introduced before R2006a

expint

Exponential integral

Syntax

```
Y = expint(X)
```

Definitions

The exponential integral computed by this function is defined as

$$E_1(x) = \int_x^\infty e^{-t} / t \, dt$$

Another common definition of the exponential integral function is the Cauchy principal value integral

$$\mathrm{Ei}(x) = \int_{-\infty}^x e^t / t \, dt$$

which, for real positive x, is related to `expint` as

$$E_1(-x) = -\mathrm{Ei}(x) - i\pi$$

Description

`Y = expint(X)` evaluates the exponential integral for each element of `X`.

References

[1] Abramowitz, M. and I. A. Stegun. *Handbook of Mathematical Functions*. Chapter 5, New York: Dover Publications, 1965.

Introduced before R2006a

expm

Matrix exponential

Syntax

```
Y = expm(X)
```

Description

`Y = expm(X)` computes the matrix exponential of `X`.

Although it is not computed this way, if `X` has a full set of eigenvectors `V` with corresponding eigenvalues `D`, then

```
[V,D] = EIG(X) and EXPM(X) = V*diag(exp(diag(D)))/V
```

Use `exp` for the element-by-element exponential.

Examples

This example computes and compares the matrix exponential of `A` and the exponential of `A`.

```
A = [1         1         0
     0         0         2
     0         0        -1 ];

expm(A)
ans =
    2.7183   1.7183         1.0862
    0        1.0000         1.2642
    0             0         0.3679

exp(A)
ans =
    2.7183         2.7183         1.0000
    1.0000         1.0000         7.3891
```

```
    1.0000         1.0000         0.3679
```

Notice that the diagonal elements of the two results are equal. This would be true for any triangular matrix. But the off-diagonal elements, including those below the diagonal, are different.

More About

Algorithms

`expm` uses the Padé approximation with scaling and squaring. See reference [3], below.

Note The files, expmdemo1.m, expmdemo2.m, and expmdemo3.m illustrate the use of Padé approximation, Taylor series approximation, and eigenvalues and eigenvectors, respectively, to compute the matrix exponential. References [1] and [2] describe and compare many algorithms for computing a matrix exponential.

References

[1] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, p. 384, Johns Hopkins University Press, 1983.

[2] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review 20*, 1978, pp. 801–836. Reprinted and updated as "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later," *SIAM Review 45*, 2003, pp. 3–49.

[3] Higham, N. J., "The Scaling and Squaring Method for the Matrix Exponential Revisited," *SIAM J. Matrix Anal. Appl.*, 26(4) (2005), pp. 1179–1193.

See Also

`exp` | `expm1` | `funm` | `logm` | `eig` | `sqrtm`

Introduced before R2006a

expm1

Compute exp(x)-1 accurately for small values of x

Syntax

```
y = expm1(x)
```

Description

`y = expm1(x)` computes `exp(x)-1`, compensating for the roundoff in `exp(x)`.

For small `x`, `expm1(x)` is approximately `x`, whereas `exp(x)-1` can be zero.

See Also

`exp` | `expm` | `log1p`

Introduced before R2006a

export2wsdlg

Create dialog box for exporting variables to workspace

Syntax

```
export2wsdlg(cklabels,defaultvars,xitems)
export2wsdlg(cklabels,defaultvars,xitems,title)
export2wsdlg(cklabels,defaultvars,xitems,title,selected)
export2wsdlg(cklabels,defaultvars,xitems,title,selected,helpfunction)
export2wsdlg(cklabels,defaultvars,xitems,title,selected,helpfunction,functionl
hdialog = export2wsdlg(...)
[hdialog,ok_pressed] = export2wsdlg(...)
```

Description

`export2wsdlg(cklabels,defaultvars,xitems)` creates a dialog with a series of check boxes and edit fields. `cklabels` is a cell array of labels for the check boxes. `defaultvars` is a cell array of strings that serve as a basis for variable names that appear in the edit fields. `xitems` is a cell array of the values to be stored in the variables. If there is only one item to export, `export2wsdlg` creates a text control instead of a check box.

Note: By default, the dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding.

`export2wsdlg(cklabels,defaultvars,xitems,title)` creates the dialog with `title` as its title.

`export2wsdlg(cklabels,defaultvars,xitems,title,selected)` creates the dialog allowing the user to control which check boxes are checked. `selected` is a logical array whose length is the same as `cklabels`. True indicates that the check box should initially be checked, false unchecked.

`export2wsdlg(cklabels,defaultvars,xitems,title,selected,helpfunction)` creates the dialog with a help button. `helpfunction` is a callback that displays help.

`export2wsdlg(cklabels,defaultvars,xitems,title,selected,helpfunction,functionl` creates a dialog that enables the user to pass in `functionlist`, a cell array of functions and optional arguments that calculate, then return the value to export. `functionlist` should be the same length as `cklabels`.

`hdialog = export2wsdlg(...)` returns the handle of the dialog.

`[hdialog,ok_pressed] = export2wsdlg(...)` sets `ok_pressed` to true if the OK button is pressed, or false otherwise. If two return arguments are requested, `hdialog` is `[]` and the function does not return until the dialog is closed.

The user can edit the text fields to modify the default variable names. If the same name appears in multiple edit fields, `export2wsdlg` creates a structure using that name. It then uses the `defaultvars` as field names for that structure.

The lengths of `cklabels`, `defaultvars`, `xitems` and `selected` must all be equal.

The strings in `defaultvars` must be unique.

Examples

This example creates a dialog box that enables the user to save the variables `sumA` and/or `meanA` to the workspace. The dialog box title is Save Sums to Workspace.

```
A = randn(10,1);
checkLabels = {'Save sum of A to variable named:' ...
               'Save mean of A to variable named:'};
varNames = {'sumA','meanA'};
items = {sum(A),mean(A)};
export2wsdlg(checkLabels,varNames,items,...
             'Save Sums to Workspace');
```

exportsetupdlg

Open figure Export Setup dialog box

The `exportseupdlg` function displays the Export Setup dialog box for a figure. The dialog box is the same as the one that displays when you select **File > Export Setup...** from the menu at the top of a figure.

Syntax

```
exportsetupdlg(f)
exportsetupdlg
```

Description

`exportsetupdlg(f)` displays the export settings dialog box. MATLAB applies your selections to the figure, `f`.

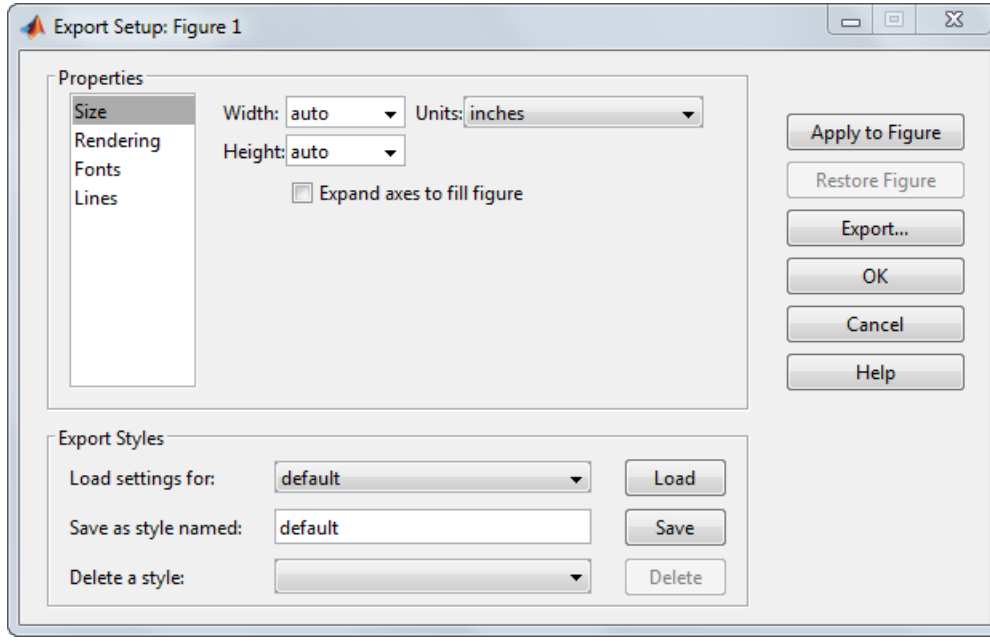
`exportsetupdlg` applies your selections to the current figure. If no figure exists, MATLAB creates a new figure.

Examples

Export Setting for a Figure

Create a figure and display the Export Setup dialog.

```
f = figure;
exportsetupdlg(f);
```



Input Arguments

`f` — Target figure

figure object

Target figure, specified as a figure object.

See Also

`printdlg` | `printpreview`

Introduced in R2006b

eye

Identity matrix

Syntax

```
I = eye
I = eye(n)
I = eye(n,m)
I = eye(sz)

I = eye(classname)
I = eye(n,classname)
I = eye(n,m,classname)
I = eye(sz,classname)

I = eye('like',p)
I = eye(n,'like',p)
I = eye(n,m,'like',p)
I = eye(sz,'like',p)
```

Description

`I = eye` returns the scalar, `1`.

`I = eye(n)` returns an `n`-by-`n` identity matrix with ones on the main diagonal and zeros elsewhere.

`I = eye(n,m)` returns an `n`-by-`m` matrix with ones on the main diagonal and zeros elsewhere.

`I = eye(sz)` returns an array with ones on the main diagonal and zeros elsewhere. The size vector, `sz`, defines `size(I)`. For example, `eye([2,3])` returns a 2-by-3 array with ones on the main diagonal and zeros elsewhere.

`I = eye(classname)` returns a scalar, `1`, where the string, `classname`, specifies the data type. For example, `eye('int8')` returns a scalar, 8-bit integer.

`I = eye(n,classname)` returns an `n`-by-`n` identity matrix of data type `classname`.

`I = eye(n,m,classname)` returns an `n`-by-`m` matrix of data type `classname` with ones on the main diagonal and zeros elsewhere.

`I = eye(sz,classname)` returns a matrix with ones on the main diagonal and zeros elsewhere. The size vector, `sz`, defines `size(I)` and `classname` defines `class(I)`.

`I = eye('like',p)` returns a scalar, `1`, with the same data type, sparsity, and complexity (real or complex) as the numeric variable, `p`.

`I = eye(n,'like',p)` returns an `n`-by-`n` identity matrix like `p`.

`I = eye(n,m,'like',p)` returns an `n`-by-`m` matrix like `p`.

`I = eye(sz,'like',p)` returns a matrix like `p` where the size vector, `sz`, defines `size(I)`.

Examples

Square Identity Matrix

Create a 4-by-4 identity matrix.

```
I = eye(4)
```

```
I =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Rectangular Matrix

Create a 2-by-3 identity matrix.

```
I = eye(2,3)
```

```
I =

     1     0     0
```

```
     0     1     0
```

Identity Vector

Create a 3-by-1 identity vector.

```
sz = [3,1];
I = eye(sz)
```

```
I =

     1
     0
     0
```

Nondefault Numeric Data Type

Create a 3-by-3 identity matrix whose elements are 32-bit unsigned integers.

```
I = eye(3,'uint32'),
class(I)
```

```
I =

           1           0           0
           0           1           0
           0           0           1


ans =

uint32
```

Complex Identity Matrix

Create a 2-by-2 identity matrix that is not real valued, but instead is complex like an existing array.

Define a complex vector.

```
p = [1+2i 3i];
```

Create an identity matrix that is complex like `p`.

```
I = eye(2,'like',p)

I =

   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i
```

Sparse Identity Matrix

Define a 5-by-5 sparse matrix.

```
p = sparse(5,5,pi);
```

Create a 5-by-5 identity matrix that is sparse like `P`.

```
I = eye(5,'like',p)

I =

   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Size and Numeric Data Type Defined by Existing Array

Define a 2-by-2 matrix of single precision.

```
p = single([1 3 ; 2 4]);
```

Create an identity matrix that is the same size and data type as `P`.

```
I = eye(size(p),'like',p),
class(I)

I =

     1     0
     0     1


ans =
```

`single`

Input Arguments

`n` — Size of first dimension of `I`

integer value

Size of first dimension of `I`, specified as an integer value.

- If `n` is the only integer input argument, then `I` is a square n-by-n identity matrix.
- If `n` is `0`, then `I` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`m` — Size of second dimension of `I`

integer value

Size of second dimension of `I`, specified as an integer value.

- If `m` is `0`, then `I` is an empty matrix.
- If `m` is negative, then it is treated as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of `I`

row vector of no more than two integer values

Size of `I`, specified as a row vector of no more than two integer values.

- If an element of `sz` is `0`, then `I` is an empty matrix.
- If an element of `sz` is negative, then the element is treated as `0`.

Example: `sz = [2,3]` defines `I` as a 2-by-3 matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`classname` — Output class

`'double'` (default) | `'single'` | `'int8'` | `'uint8'` | ...

Output class, specified as `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Data Types: `char`

`p` — Prototype

numeric variable

Prototype, specified as a numeric variable.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

More About

- "Class Support for Array-Creation Functions"

See Also

`ones` | `speye` | `zeros`

Introduced before R2006a

ezcontour

Easy-to-use contour plotter

Syntax

```
ezcontour(fun)
ezcontour(fun,domain)
ezcontour(...,n)
ezcontour(axes_handle,...)
h = ezcontour(...)
```

Description

`ezcontour(fun)` plots the contour lines of `fun(x,y)` using the `contour` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for a MATLAB file function or an anonymous function (see `function handle` and "Anonymous Functions") or a string (see Tips).

`ezcontour(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]` (where $min < x < max$, $min < y < max$).

`ezcontour(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontour(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezcontour(...)` returns the handle to a contour object in `h`.

`ezcontour` automatically adds a title and axis labels.

Examples

Create Contour Plot of Mathematical Expression

This mathematical expression defines a function of two variables, x and y.

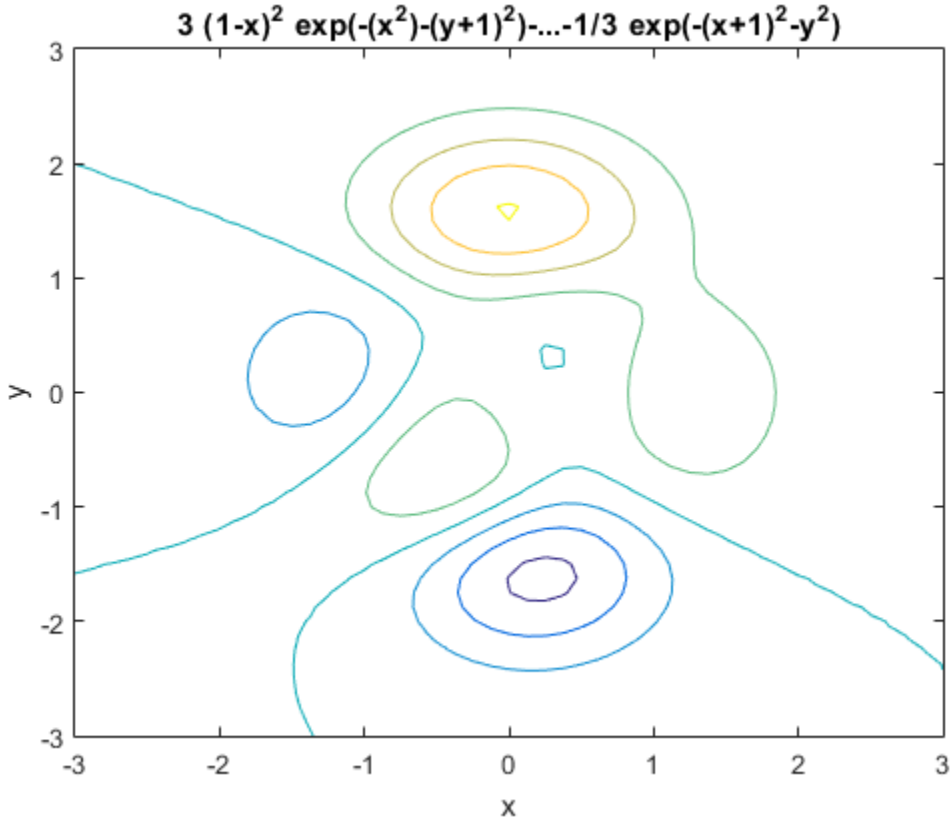
$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

The `ezcontour` function requires a function handle argument. Write this mathematical expression in MATLAB® syntax as an anonymous function with handle `f`. You can define an anonymous function in the command window without creating a separate file. For convenience, write the function on three lines.

```
f = @(x,y) 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

Pass the function handle, `f`, to `ezcontour`. Specify a domain from -3 to 3 in both the x-direction and y-direction and use a 49-by-49 computational grid.

```
ezcontour(f,[-3,3],49)
```


3 (1-x)^2 exp(-(x^2)-(y+1)^2)-...-1/3 exp(-(x+1)^2-y^2)
3
2
1
0
-1
-2
-3
y
-3
-2
-1
0
1
2
3
x

In this particular case, the title string is too long to fit at the top of the graph so MATLAB® abbreviates the string.

More About

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezcontour`. For example, the MATLAB syntax for a contour plot of the expression

```
sqrt(x.^2 + y.^2)
```

is written as

```
ezcontour('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezcontour`.

If the function to be plotted is a function of the variables u and v (rather than x and y), the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontour('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontour`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontour(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezcontour` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then use an anonymous function to specify that parameter:

```
ezcontour(@(x,y)myfun(x,y,2))
```

See Also

`contour` | `ezmesh` | `ezcontourf` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `function_handle`

Introduced before R2006a

ezcontourf

Easy-to-use filled contour plotter

Syntax

```
ezcontourf(fun)
ezcontourf(fun,domain)
ezcontourf(...,n)
ezcontourf(axes_handle,...)
h = ezcontourf(...)
```

Description

`ezcontourf(fun)` plots the contour lines of `fun(x,y)` using the `contourf` function. `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle or a string (see Tips).

`ezcontourf(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]`, where `min` < `x` < `max`, `min` < `y` < `max`).

`ezcontourf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontourf(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = ezcontourf(...)` returns the handle to a contour object in `h`.

`ezcontourf` automatically adds a title and axis labels.

Examples

Create Filled Contour Plot of Mathematical Expression

This mathematical expression defines a function of two variables, x and y.

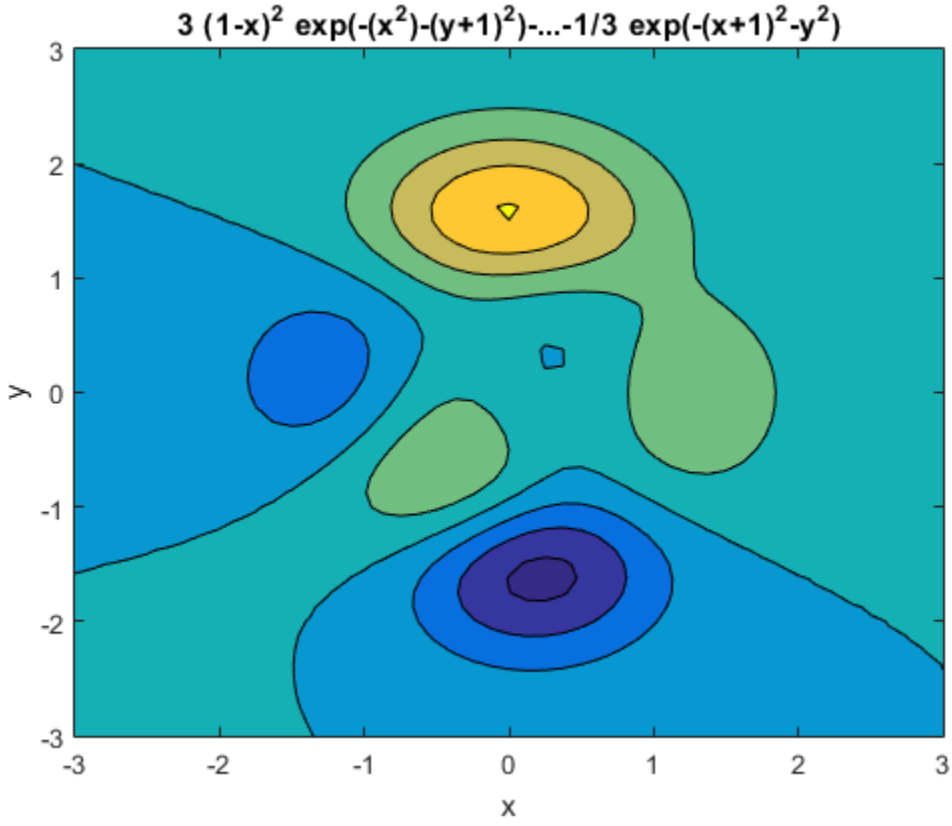
$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-x^2-y^2} - \frac{1}{3} e^{-(x+1)^2-y^2}$$

The `ezcontourf` function requires a function handle argument. Write this mathematical expression in MATLAB® syntax as an anonymous function with handle `f`. You can define an anonymous function in the command window without creating a separate file. For convenience, write the function on three lines.

```
f = @(x,y) 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

Pass the function handle, `f`, to `ezcontourf`. Specify a domain from -3 to 3 in both the x-direction and y-direction and use a 49-by-49 computational grid.

```
ezcontourf(f,[-3,3],49)
```


3 (1-x)^2 exp(-(x^2)-(y+1)^2)-...-1/3 exp(-(x+1)^2-y^2)
3
2
1
0
-1
-2
-3
y
-3
-2
-1
0
1
2
3
x

In this particular case, the title string is too long to fit at the top of the graph so MATLAB® abbreviates the string.

More About

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezcontourf`. For example, the MATLAB syntax for a filled contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezcontourf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezcontourf`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontourf('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontourf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontourf(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezcontourf` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezcontourf(@(x,y)myfun(x,y,2))
```

- Anonymous Functions

See Also

`contourf` | `ezmesh` | `ezcontour` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `function_handle`

Introduced before R2006a

ezmesh

Easy-to-use 3-D mesh plotter

Syntax

```
ezmesh(fun)
ezmesh(fun,domain)
ezmesh(funx,funy,funz)
ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])
ezmesh(funx,funy,funz,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
ezmesh(axes_handle,...)
h = ezmesh(...)
```

Description

`ezmesh(fun)` creates a graph of `fun(x,y)` using the `mesh` function. `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle or a string (see the Tips section).

`ezmesh(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezmesh(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmesh(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmesh(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmesh(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

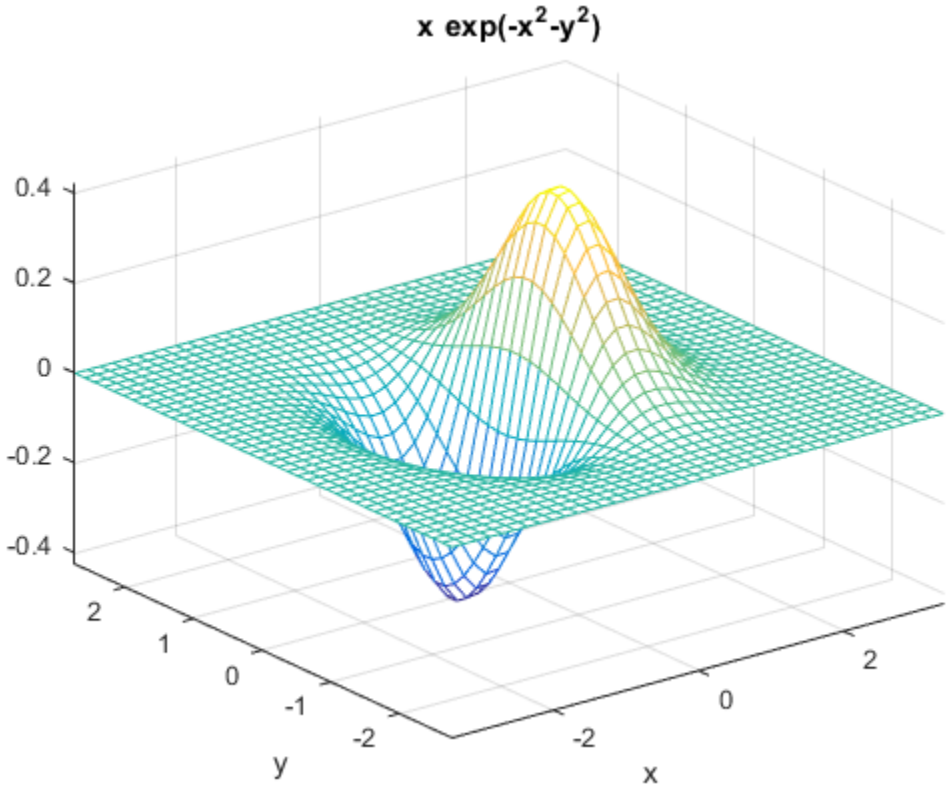
`h = ezmesh(...)` returns the handle to a surface object in `h`.

Examples

Mesh Plot of Mathematical Function

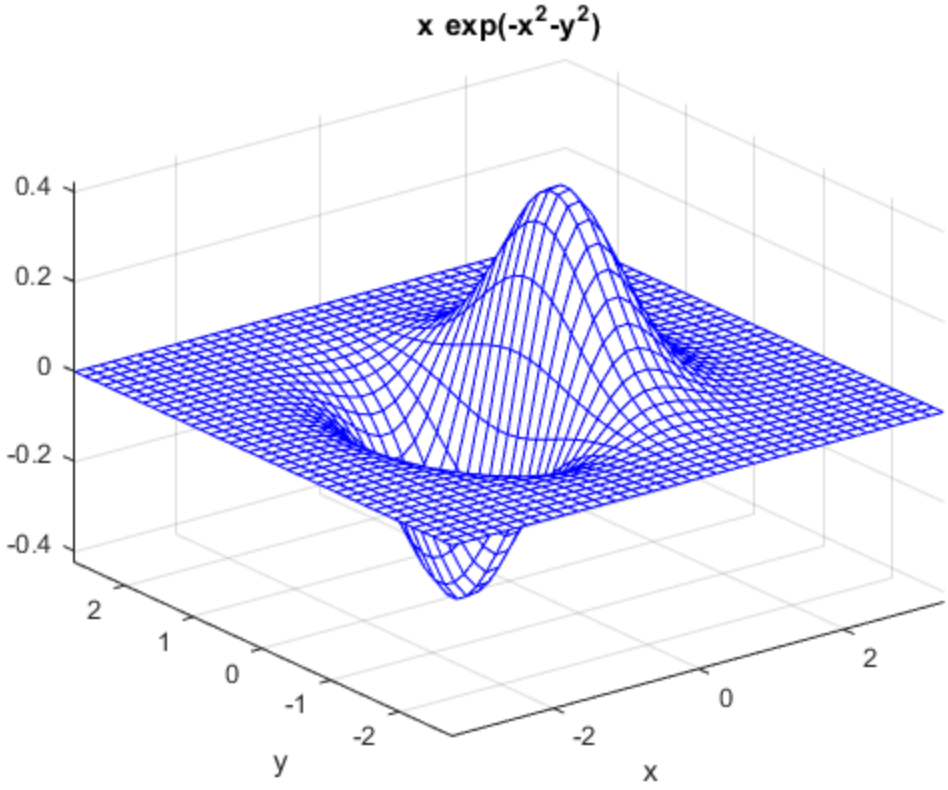
Create a mesh plot of the function $f(x,y) = xe^{-x^2-y^2}$ over a 40-by-40 grid.

```
fh = @(x,y) x.*exp(-x.^2-y.^2);
ezmesh(fh,40)
```



Set the mesh lines to a uniform blue color by setting the colormap to a single color.

```
colormap([0 0 1])
```



More About

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezmesh`. For example, the MATLAB syntax for a mesh plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmesh('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezmesh`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmesh('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmesh`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmesh(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmesh` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmesh(@(x,y)myfun(x,y,2))
```

- Anonymous Functions

See Also

`ezmeshc` | `mesh` | `function_handle`

Introduced before R2006a

ezmeshc

Easy-to-use combination mesh/contour plotter

Syntax

```
ezmeshc(fun)
ezmeshc(fun,domain)
ezmeshc(funx,funy,funz)
ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])
ezmeshc(funx,funy,funz,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
ezmesh(axes_handle,...)
h = ezmeshc(...)
```

Description

`ezmeshc(fun)` creates a graph of `fun(x,y)` using the `meshc` function. `fun` is plotted over the default domain -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle or a string (see the Tips section).

`ezmeshc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezmeshc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmeshc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmeshc(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmeshc(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

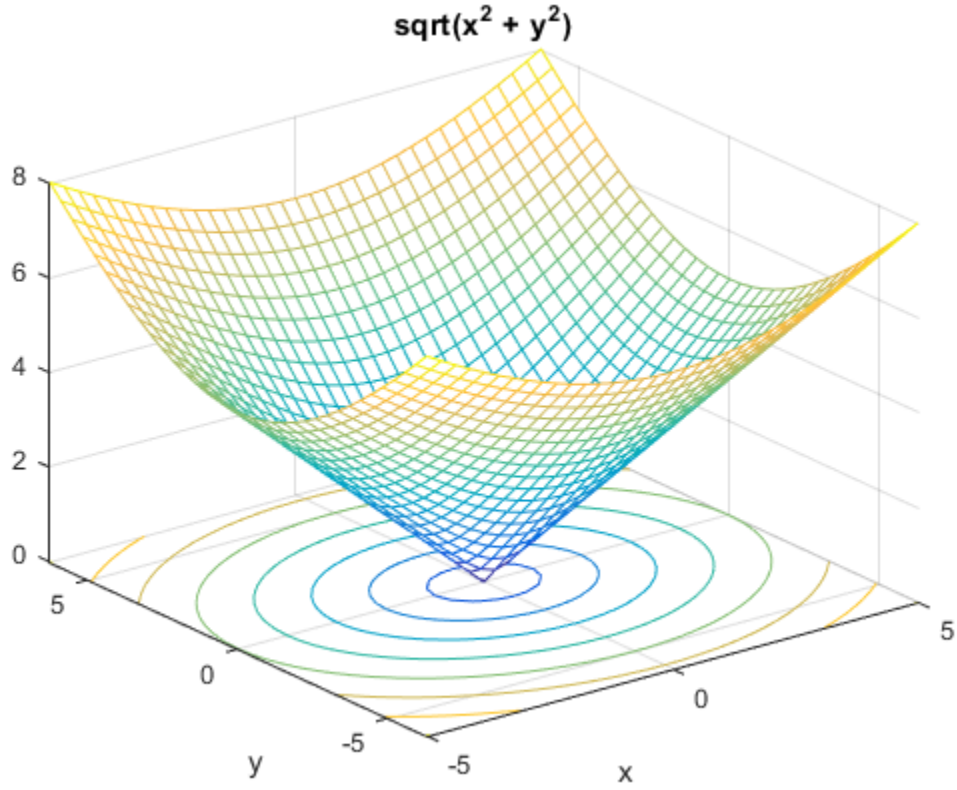
`h = ezmeshc(...)` returns the handle to a surface object in `h`.

Examples

Mesh and Contour Plot of Mathematical Function

Create a mesh/contour plot of the expression $f(x,y) = \sqrt{x^2 + y^2}$ over the domain $-5 < x < 5$ and $-2\pi < y < 2\pi$ with a computational grid size of 35-by-35.

```
ezmeshc('sqrt(x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
```



More About

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezmeshc`. For example, the MATLAB syntax for a mesh/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmeshc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezmeshc`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmeshc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmeshc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmeshc(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmeshc` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmeshc(@(x,y)myfun(x,y,2))
```

- Anonymous Functions

See Also

`ezmesh` | `ezsurfc` | `meshc` | `function_handle`

ezplot

Easy-to-use function plotter

Syntax

```
ezplot(fun)
ezplot(fun,[xmin,xmax])
ezplot(fun2)
ezplot(fun2,[xymin,xymax])
ezplot(fun2,[xmin,xmax,ymin,ymax])
ezplot(funx,funy)
ezplot(funx,funy,[tmin,tmax])
ezplot(...,figure_handle)
ezplot(axes_handle,...)
h = ezplot(...)
```

Description

`ezplot(fun)` plots the expression `fun(x)` over the default domain -2π < `x` < 2π, where `fun(x)` is an explicit function of only `x`.

`fun` can be a function handle or a string.

`ezplot(fun,[xmin,xmax])` plots `fun(x)` over the domain: `xmin` < `x` < `xmax`.

For an implicit function, `fun2(x,y)`:

`ezplot(fun2)` plots `fun2(x,y) = 0` over the default domain -2π < `x` < 2π, -2π < `y` < 2π.

`ezplot(fun2,[xymin,xymax])` plots `fun2(x,y) = 0` over `xymin` < `x` < `xymax` and `xymin` < `y` < `xymax`.

`ezplot(fun2,[xmin,xmax,ymin,ymax])` plots `fun2(x,y) = 0` over `xmin` < `x` < `xmax` and `ymin` < `y` < `ymax`.

`ezplot(funx,funy)` plots the parametrically defined planar curve `funx(t)` and `funy(t)` over the default domain $0 < t < 2\pi$.

`ezplot(funx,funy,[tmin,tmax])` plots `funx(t)` and `funy(t)` over `tmin` < `t` < `tmax`.

`ezplot(...,figure_handle)` plots the given function over the specified domain in the figure window identified by the handle `figure`.

`ezplot(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

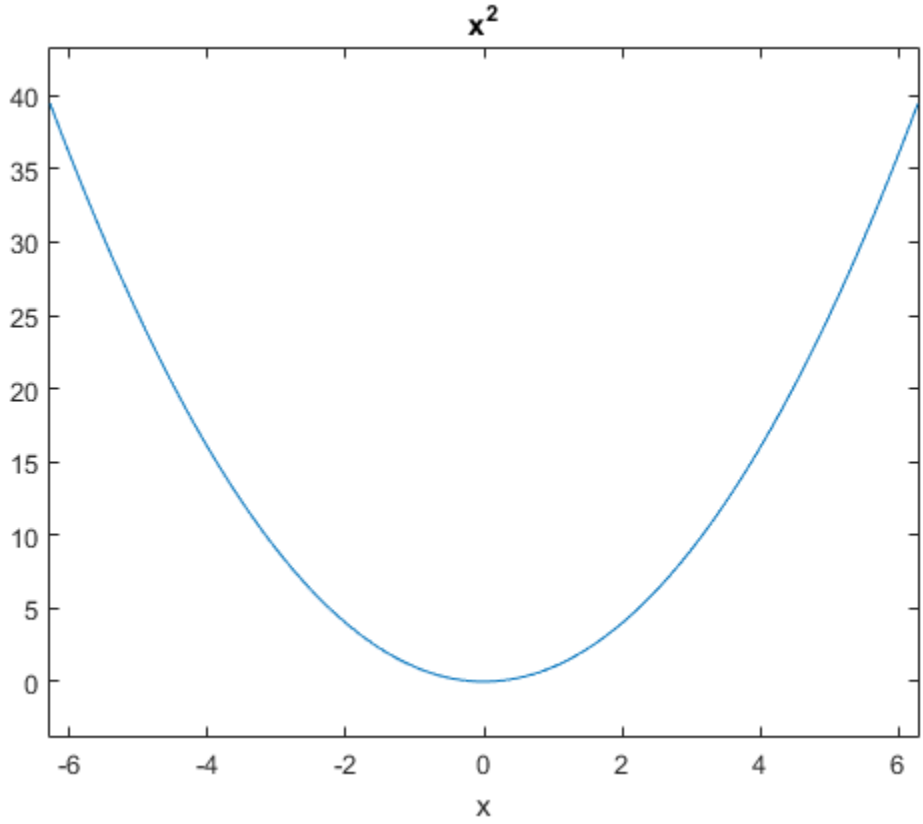
`h = ezplot(...)` returns either a chart line or contour object.

Examples

Plot an Explicit Function

Plot the explicit function x^2 over the domain $[-2\pi, 2\pi]$.

```
ezplot('x^2')
```

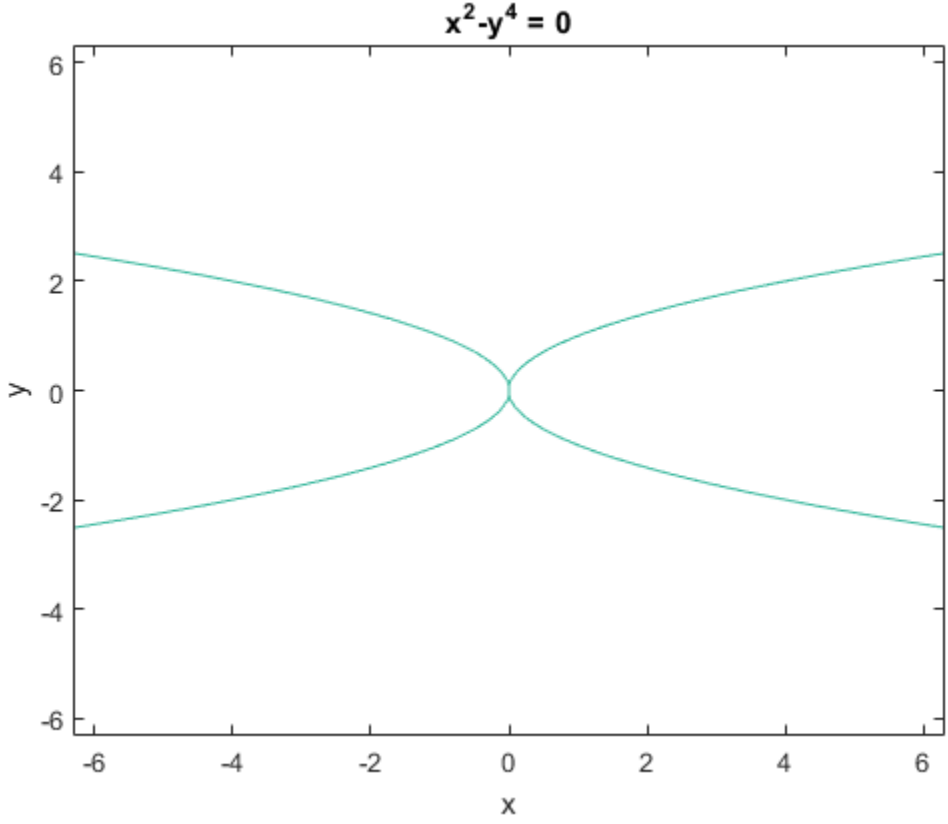


The default domain is $[-2\pi, 2\pi]$.

Plot an Implicit Function

Plot the implicitly defined function $x^2 - y^4 = 0$ over the domain $[-2\pi, 2\pi]$.

```
ezplot('x^2-y^4')
```



The default domain is $[-2\pi, 2\pi]$.

More About

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot`. For example, the MATLAB syntax for a plot of the expression

```
x.^2 - y.^2
```

which represents an implicitly defined function, is written as

```
ezplot('x^2 - y^2')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezplot`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot`.

```
fh = @(x,y) x.^2 + y.^3 - 2*y - 1;
ezplot(fh)
axis equal
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezplot(@(x,y)myfun(x,y,2))
```

- Anonymous Functions

See Also

`ezplot3` | `ezpolar` | `function_handle` | `plot`

Introduced before R2006a

ezplot3

Easy-to-use 3-D parametric curve plotter

Syntax

```
ezplot3(funx,funy,funz)
ezplot3(funx,funy,funz,[tmin,tmax])
ezplot3(...,'animate')
ezplot3(axes_handle,...)
h = ezplot3(...)
```

Description

`ezplot3(funx,funy,funz)` plots the spatial curve `funx(t)`, `funy(t)`, and `funz(t)` over the default domain $0 < t < 2\pi$.

`funx`, `funy`, and `funz` can be function handles or strings (see the Tips section).

`ezplot3(funx,funy,funz,[tmin,tmax])` plots the curve `funx(t)`, `funy(t)`, and `funz(t)` over the domain `tmin` < `t` < `tmax`.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

`ezplot3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezplot3(...)` returns the handle to the plotted objects in `h`.

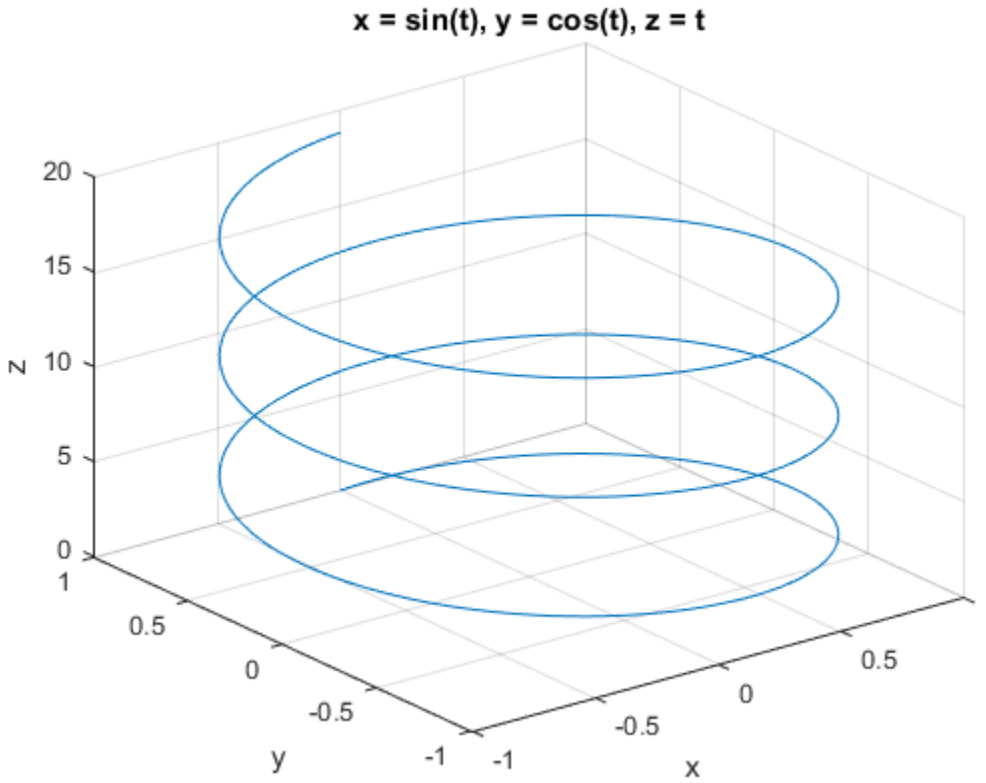
Examples

Plot a Parametric Curve

Plot this parametric curve over the domain $[0, 6\pi]$.

$$x = \sin(t), \quad y = \cos(t), \quad z = t$$

```
ezplot3('sin(t)','cos(t)','t',[0,6*pi])
```



More About

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot3`. For example, the MATLAB syntax for a plot of the expression

```
x = s./2, y = 2.*s, z = s.^2;
```

which represents a parametric function, is written as

```
ezplot3('s/2','2*s','s^2')
```

That is, `s/2` is interpreted as `s./2` in the string you pass to `ezplot3`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot3`.

```
fh1 = @(s) s./2; fh2 = @(s) 2.*s; fh3 = @(s) s.^2;
ezplot3(fh1,fh2,fh3)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot3` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfuntk`:

```
function s = myfuntk(t,k)
s = t.^k.*sin(t);
```

then you can use an anonymous function to specify that parameter:

```
ezplot3(@cos,@(t)myfuntk(t,1),@sqrt)
```

- Anonymous Functions

See Also

`ezplot` | `ezpolar` | `plot3` | `function_handle`

Introduced before R2006a

ezpolar

Easy-to-use polar coordinate plotter

Syntax

```
ezpolar(fun)
ezpolar(fun,[a,b])
ezpolar(axes_handle,...)
h = ezpolar(...)
```

Description

`ezpolar(fun)` plots the polar curve `rho = fun(theta)` over the default domain $0 <$ `theta` $< 2\pi$.

`fun` can be a function handle or a string (see the Tips section).

`ezpolar(fun,[a,b])` plots `fun` for `a` < `theta` < `b`.

`ezpolar(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

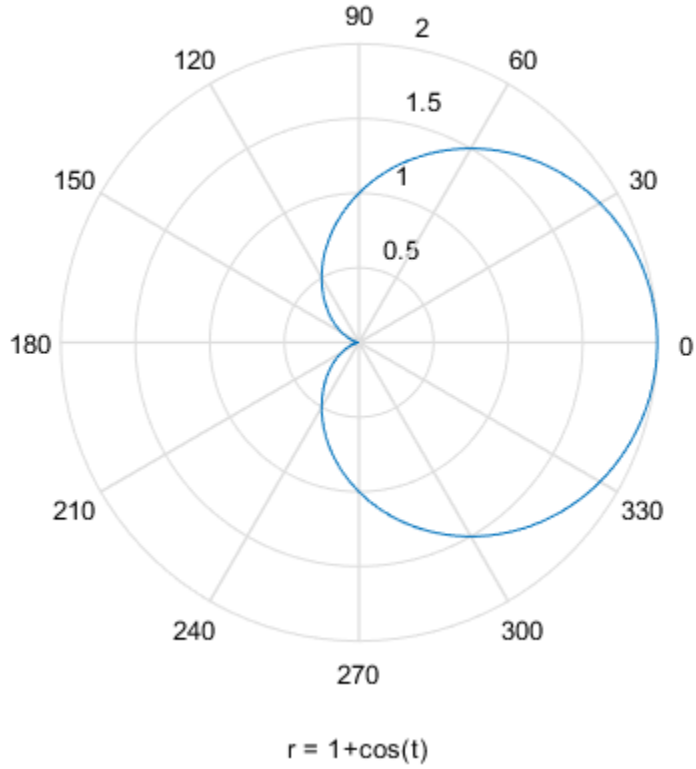
`h = ezpolar(...)` returns the handle to a line object in `h`.

Examples

Polar Plot of Mathematical Function

Plot the function $1 + \cos(t)$ over the domain $[0, 2\pi]$.

```
figure
ezpolar('1+cos(t)')
```



More About

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezpolar`. For example, the MATLAB syntax for a plot of the expression

```
t.^2.*cos(t)
```

which represents an implicitly defined function, is written as

```
ezpolar('t^2*cos(t)')
```

That is, `t^2` is interpreted as `t.^2` in the string you pass to `ezpolar`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezpolar`.

```
fh = @(t) t.^2.*cos(t);
ezpolar(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezpolar` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k1` and `k2` in `myfun`:

```
function s = myfun(t,k1,k2)
s = sin(k1*t).*cos(k2*t);
```

then you can use an anonymous function to specify the parameters:

```
ezpolar(@(t)myfun(t,2,3))
```

- Anonymous Functions

See Also

`ezplot` | `ezplot3` | `plot` | `plot3` | `polar` | `function_handle`

Introduced before R2006a

ezsurf

Easy-to-use 3-D colored surface plotter

Syntax

```
ezsurf(fun)
ezsurf(fun,domain)
ezsurf(funx,funy,funz)
ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurf(funx,funy,funz,[min,max])
ezsurf(...,n)
ezsurf(...,'circ')
ezsurf(axes_handle,...)
h = ezsurf(...)
```

Description

`ezsurf(fun)` creates a graph of `fun(x,y)` using the `surf` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle or a string (see the Tips section).

`ezsurf(fun,domain)` plots `fun` over the specified `domain`. `domain` must be a vector. See the “Algorithms” on page 1-2479 section for details on vector inputs vs axes limit outputs.

`ezsurf(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurf(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurf(...,'circ')` plots `fun` over a disk centered on the domain.

`ezsurf(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

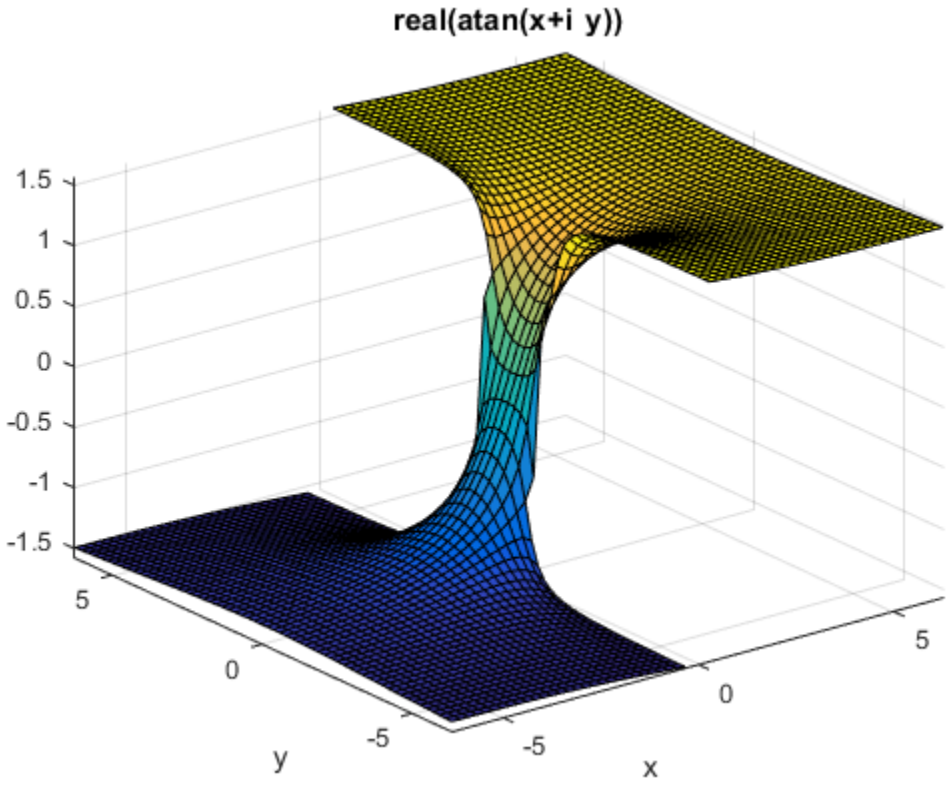
`h = ezsurf(...)` returns the handle to a surface object in `h`.

Examples

Surface Plot of Mathematical Function

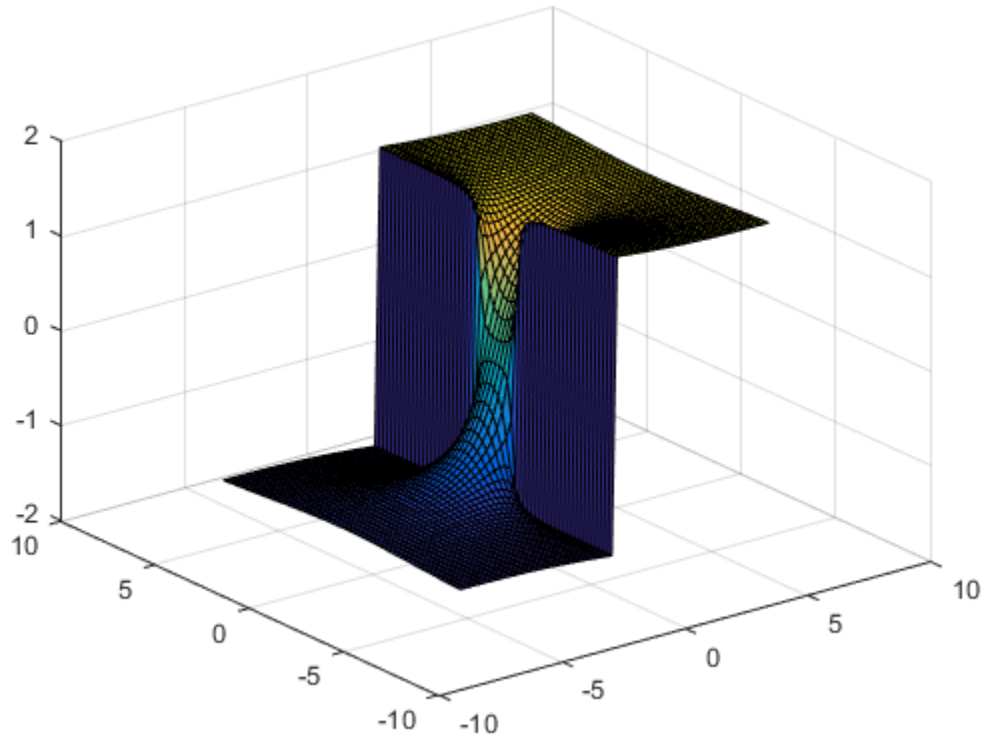
Plot the function $f(x,y) = real(a\tan(x+iy))$ over the domain $-2\pi < x < 2\pi$ and $-2\pi < y < 2\pi$. The `ezsurf` function does not plot points where the mathematical function is not defined. These points are set to `NaN` so that they do not plot.

```
figure
ezsurf('real(atan(x+i*y))')
```



Use `surf` to plot the same data without filtering discontinuities.

```
figure
[x,y] = meshgrid(linspace(-2*pi,2*pi,60));
z = real(atan(x+1i.*y));
surf(x,y,z)
```



More About

Tips

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezsurf`. For example, the MATLAB syntax for a surface plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezsurf`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurf('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurf` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurf(@(x,y)myfun(x,y,2,2,4))
```

Algorithms

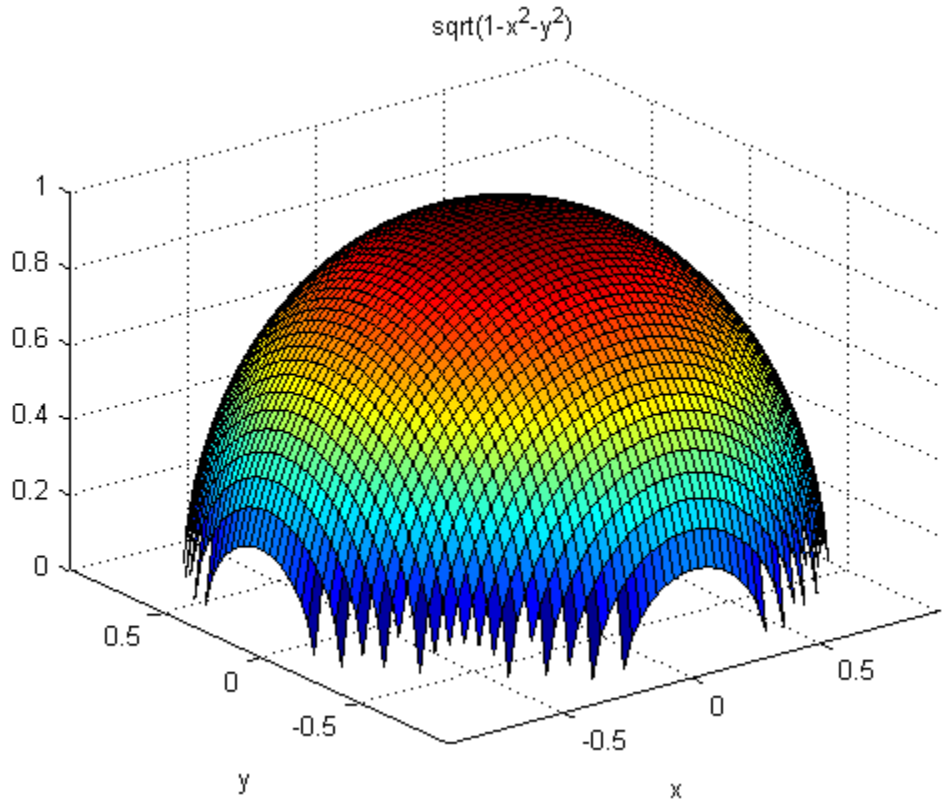
`ezsurf` determines the *x*- and *y*-axes limits in different ways depending on how you input the domain (if at all). In the following table, `R` is the vector [`xmin`, `xmax`, `ymin`, `ymax`] and `v` is the manually entered domain vector.

Number of domain values specified:	Resulting domain vector:
`v = [ ];`	`R = [-2*pi, 2*pi, -2*pi, 2*pi];`
`v = [ v(1) ];`	`R = double([-abs(v),abs(v),-abs(v),abs(v)]);`
`v = [ v(1) v(2) ];`	`R = double([v(1),v(2),v(1),v(2)]);`
`v = [ v(1) v(2) v(3) ];`	`R = double([-v(1),v(2),-abs(v(3)),abs(v(3))]);`
`v = [ v(1) v(2) v(3) v(4) ];`	`R = double(v);`
`v = [ v(1)..v(n) ]; n>4`	`R = double([-abs(v(1)), abs(v(1)), -abs(v(1)), abs(v(1))]);`

If you specify a single number in non-vector format (without square brackets, [ ]), `ezsurf` interprets it as the `n`, the number of points desired between the axes `max` and `min` values.

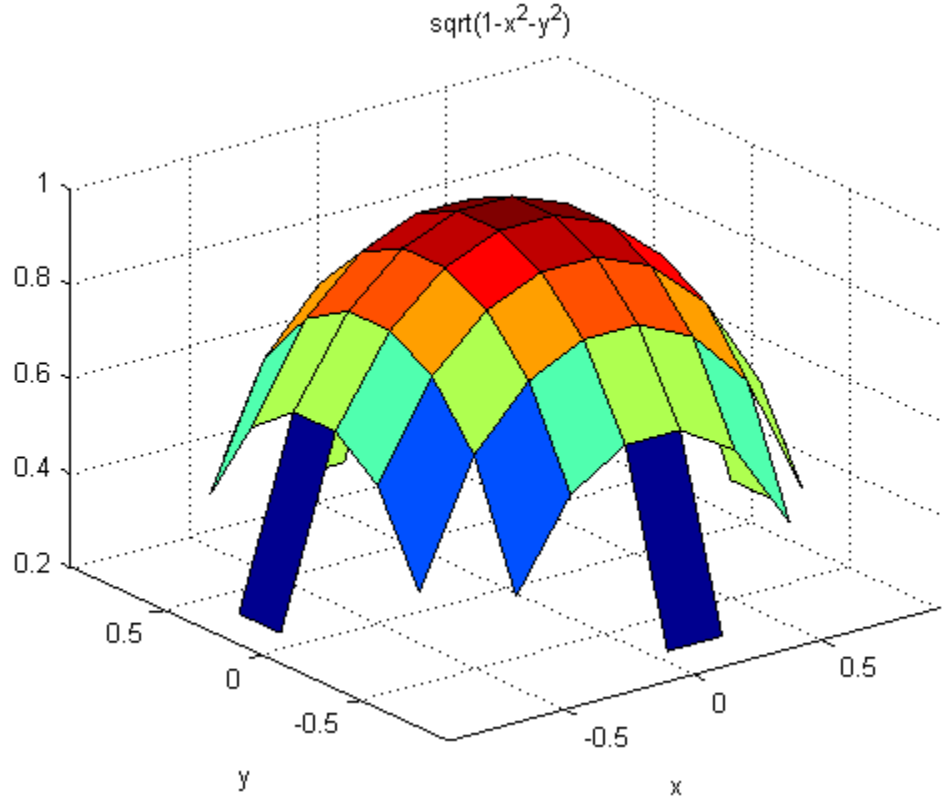
By default, `ezsurf` uses 60 points between the `max` and `min` values of an axes. When the `min` and `max` values are the default values (`R = [-2*pi, 2*pi, -2*pi, 2*pi];`), `ezsurf` ensures the 60 points fall within the non-complex range of the specified equation. For example, $\sqrt{1-x^2-y^2}$ is only real when $x^2-y^2 \le 1$. The default graph of this function looks like this:

```
ezsurf('sqrt(1-x^2-y^2)')
```



You can see that there are 60 points between the minimum and maximum values for which $\sqrt{1-x^2-y^2}$ has real values. However, when you specify the domain values to be the same as the default (`R = [-2*pi, 2*pi, -2*pi, 2*pi];`), a different result appears:

```
ezsurf('sqrt(1-x^2-y^2)',[-2*pi 2*pi])
```



In this case, the graphic limits are the same, but `ezsurf` used 60 points between the user-defined limits instead of checking to see if all those points would have real answers.

- Anonymous Functions

See Also

`ezmesh` | `ezsurfc` | `surf` | `function_handle`

Introduced before R2006a

ezsurfc

Easy-to-use combination surface/contour plotter

Syntax

```
ezsurfc(fun)
ezsurfc(fun,domain)
ezsurfc(funx,funy,funz)
ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurfc(funx,funy,funz,[min,max])
ezsurfc(...,n)
ezsurfc(...,'circ')
ezsurfc(axes_handle,...)
h = ezsurfc(...)
```

Description

`ezsurfc(fun)` creates a graph of `fun(x,y)` using the `surfc` function. The function `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle or a string (see the Tips section).

`ezsurfc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where $min < x < max$, $min < y < max$).

`ezsurfc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurfc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurfc(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurfc(...,'circ')` plots *f* over a disk centered on the domain.

`ezsurfc(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

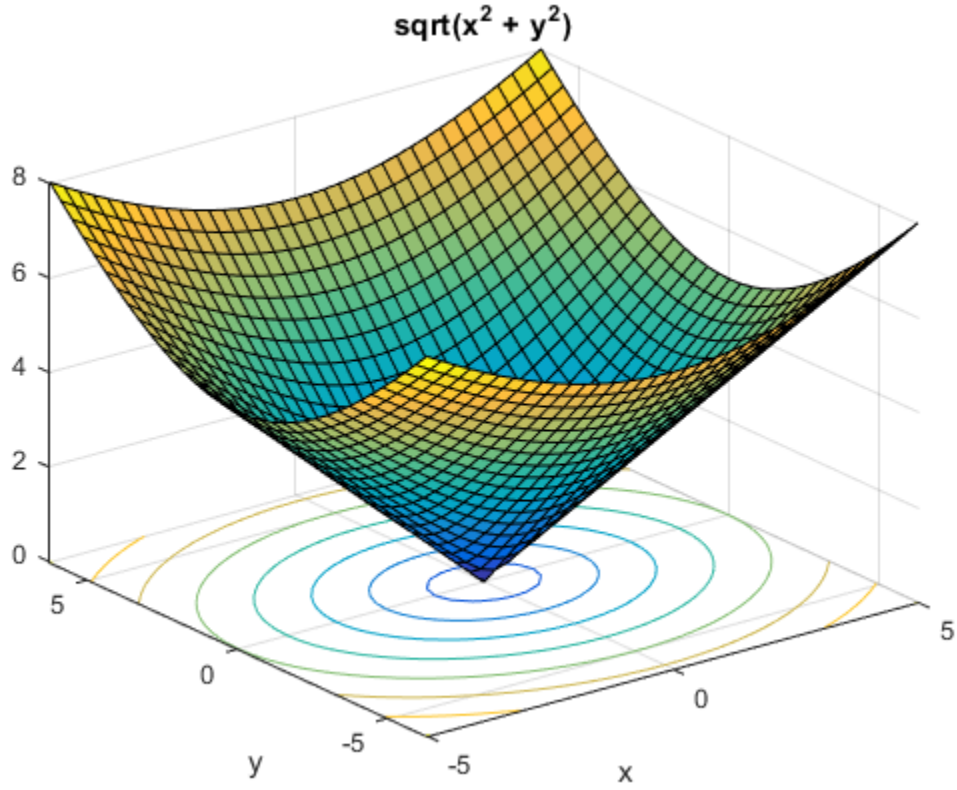
`h = ezsurfc(...)` returns the handles to the graphics objects in `h`.

Examples

Surface and Contour Plot of Mathematical Function

Create a surface/contour plot of the expression $f(x,y) = \sqrt{x^2 + y^2}$ over the domain $-5 < x < 5$ and $-2\pi < y < 2\pi$ with a computational grid size of 35-by-35.

```
ezsurfc('sqrt(x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
```



More About

Tips

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezsurfc`. For example, the MATLAB syntax for a surface/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurfc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezsurfc`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurfc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurfc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurfc` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurfc(@(x,y)myfun(x,y,2,2,4))
```

- Anonymous Functions

See Also

`ezmesh` | `ezmeshc` | `surfc` | `ezsurf` | `function_handle`

Introduced before R2006a

faceNormals

Class: TriRep

(Will be removed) Unit normals to specified triangles

Note: `faceNormals(TriRep)` will be removed in a future release. Use `faceNormal(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
FN = faceNormals(TR, TI)
```

Description

`FN = faceNormals(TR, TI)` returns the unit normal vector to each of the specified triangles TI.

Note: This query is only applicable to triangular surface meshes.

Input Arguments

`TR`	Triangulation representation.
`TI`	Column vector of indices that index into the triangulation matrix `TR.Triangulation`.

Output Arguments

`FN`	m-by-3 matrix. `m = length(TI)`, the number of triangles to be queried. Each row `FN(i,:)` represents the unit normal vector to triangle `TI(i)`.

If TI is not specified the unit normal information for the entire triangulation is returned, where the normal associated with triangle i is the `i`'th row of `FN`.

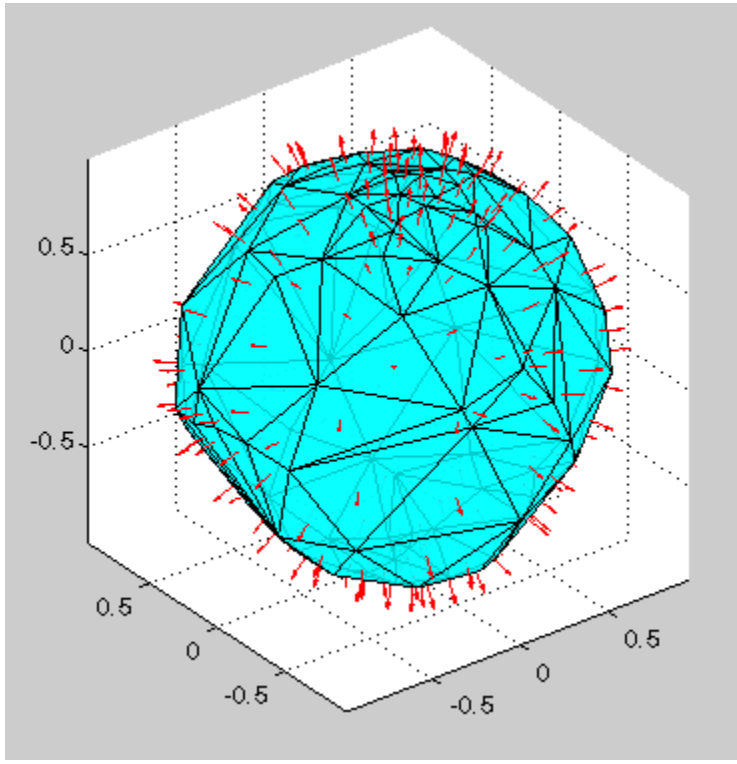
Examples

Triangulate a sample of random points on the surface of a sphere and use the `TriRep` to compute the normal to each triangle:

```
numpts = 100;
thetha = rand(numpts,1)*2*pi;
phi = rand(numpts,1)*pi;
x = cos(thetha).*sin(phi);
y = sin(thetha).*sin(phi);
z = cos(phi);
dt = DelaunayTri(x,y,z);
[tri Xb] = freeBoundary(dt);
tr = TriRep(tri, Xb);
P = incenters(tr);
fn = faceNormals(tr);
trisurf(tri,Xb(:,1),Xb(:,2),Xb(:,3), ...
     'FaceColor', 'cyan', 'faceAlpha', 0.8);
axis equal;
hold on;
```

Display the result using a quiver plot:

```
quiver3(P(:,1),P(:,2),P(:,3), ...
     fn(:,1),fn(:,2),fn(:,3),0.5, 'color','r');
hold off;
```



See Also

`delaunayTriangulation` | `freeBoundary` | `triangulation`

factor

Prime factors

Syntax

```
f = factor(n)
```

Description

`f = factor(n)` returns a row vector containing the prime factors of `n`. Vector `f` is of the same data type as `n`.

Examples

Prime Factors of Double Integer Value

```
f = factor(200)
```

```
f =

     2     2     2     5     5
```

Multiply the elements of `f` to reproduce the input value.

```
prod(f)
```

```
ans =

   200
```

Prime Factors of Unsigned Integer Value

```
n = uint16(138);
f = factor(n)
```

```
f =

       2       3      23
```

Multiply the elements of `f` to reproduce `n`.

```
prod(f)
```

```
ans =

   138
```

Input Arguments

n — Input value

real, nonnegative integer scalar

Input value, specified as a real, nonnegative integer scalar.

Example: `10`

Example: `int16(64)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`isprime` | `primes`

Introduced before R2006a

factorial

Factorial of input

Syntax

```
f = factorial(n)
```

Description

`f = factorial(n)` returns the product of all positive integers less than or equal to `n`, where `n` is a nonnegative integer value. If `n` is an array, then `f` contains the factorial of each value of `n`. The data type and size of `f` is the same as that of `n`.

Examples

10!

```
f = factorial(10)
```

```
f =

     3628800
```

22!

```
format long
f = factorial(22)
```

```
f =

     1.124000727777608e+21
```

In this case, `f` is accurate up to 15 digits, `1.12400072777760e+21`, because double-precision numbers are only accurate up to 15 digits.

Reset the output format to the default.

```
format
```

Factorial of Array Elements

```
n = [0 1 2; 3 4 5];
f = factorial(n)
```

```
f =

     1     1     2
     6    24   120
```

Factorial of Unsigned Integer Values

```
n = uint64([5 10 15 20]);
f = factorial(n)
```

```
f =

   120     3628800    1307674368000    2432902008176640000
```

Input Arguments

n — Input values

scalar, vector, or array of real, nonnegative integer values

Input values, specified as a scalar, vector, or array of real, nonnegative integers.

Example: `5`

Example: `[0 1 2 3 4]`

Example: `int16([10 15 20])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Tips

Limitations

- For double-precision inputs, the result is exact when `n` is less than or equal to `21`. Larger values of `n` produce a result that has the correct order of magnitude and is accurate for the first 15 digits. This is because double-precision numbers are only accurate up to 15 digits.
- For single-precision inputs, the result is exact when `n` is less than or equal to `13`. Larger values of `n` produce a result that has the correct order of magnitude and is accurate for the first 8 digits. This is because single-precision numbers are only accurate up to 8 digits.

Saturation

- The table below describes the saturation behavior of each data type when used with the `factorial` function. The values in the last column indicate the saturation point; that is, the first positive integer whose actual factorial is larger than the maximum representable value in the middle column. For `single` and `double`, all values larger than the maximum value are returned as `Inf`. For the integer data types, the saturation value is equal to the maximum value in the middle column.

Data type	Maximum Value	Factorial Saturation Threshold
`double`	`realmax`	`factorial(171)`
`single`	`realmax('single')`	`factorial(single(35))`
`uint64`	2^{64}-1	`factorial(uint64(21))`
`int64`	2^{63}-1	`factorial(int64(21))`
`uint32`	2^{32}-1	`factorial(uint32(13))`
`int32`	2^{31}-1	`factorial(int32(13))`
`uint16`	2^{16}-1	`factorial(uint16(9))`
`int16`	2^{15}-1	`factorial(int16(8))`
`uint8`	2^{8}-1	`factorial(uint8(6))`
`int8`	2^{7}-1	`factorial(int8(6))`

See Also

`prod`

Introduced before R2006a

false

Logical 0 (false)

Syntax

```
false
F = false(n)
F = false(sz)
F = false(sz1,...,szN)
F = false(___,'like',p)
```

Description

`false` is shorthand for `logical(0)`.

`F = false(n)` is an `n`-by-`n` array of logical zeros.

`F = false(sz)` is an array of logical zeros where the size vector, `sz`, defines `size(F)`. For example, `false([2 3])` returns a 2-by-3 array of logical zeros.

`F = false(sz1,...,szN)` is a `sz1`-by-`...`-by-`szN` array of logical zeros where `sz1,...,szN` indicates the size of each dimension. For example, `false(2,3)` returns a 2-by-3 array of logical zeros.

`F = false(___,'like',p)` returns an array of logical zeros of the same sparsity as the logical variable `p` using any of the previous size syntaxes.

Examples

Generate Square Matrix of Logical Zeros

Use `false` to generate a 3-by-3 square matrix of logical zeros.

```
A = false(3)
class(A)
```

```
A =
     0     0     0
     0     0     0
     0     0     0
ans =
logical
```

The result is of class `logical`.

Generate Array of Logical Zeros with Arbitrary Dimensions

Use `false` to generate a 3-by-2-by-2 array of logical zeros.

```
false(3,2,2)
```

```
ans(:,:,1) =
     0     0
     0     0
     0     0

ans(:,:,2) =
     0     0
     0     0
     0     0
```

Alternatively, use a size vector to specify the size of the matrix.

```
false([3 2 2])
```

```
ans(:,:,1) =
     0     0
     0     0
     0     0

ans(:,:,2) =
     0     0
     0     0
     0     0
```

Note that specifying multiple vector inputs returns an error.

Execute Logic Statement

`false` along with `true` can be used to execute logic statements.

Test the logical statement $\sim(A\ and\ B) \equiv (\sim A)\ or\ (\sim B)$ for A = logical false and B = logical true.

```
~(false & true) == (~false) | (~true)
```

```
ans =
     1
```

The result is logical 1 (true), since the logical statements on both sides of the equation are equivalent. This logical statement is an instance of De Morgan's Law.

Generate Logical Array of Selected Sparsity

Generate a logical array of the same data type and sparsity as the selected array.

```
A = logical(sparse(5,3));
whos A
F = false(4,'like',A);
whos F
```

```
  Name      Size            Bytes  Class      Attributes
  A         5x3                41  logical    sparse
  Name      Size            Bytes  Class      Attributes
  F         4x4                49  logical    sparse
```

The output array F has the same `sparse` attribute as the specified array A.

Input Arguments

n — Size of square matrix

integer

Size of square matrix, specified as an integer. `n` sets the output array size to `n`-by-`n`. For example, `false(3)` returns a 3-by-3 array of logical zeros.

- If n is `0`, then F is an empty matrix.
- If n is negative, then it is treated as `0`.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Size vector

row vector of integers

Size vector, specified as a row vector of integers. For example, `false([2 3])` returns a 2-by-3 array of logical zeros.

- If the size of any dimension is `0`, then F is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, F, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size inputs

comma-separated list of integers

Size inputs, specified by a comma-separated list of integers. For example, `false(2,3)` returns a 2-by-3 array of logical zeros.

- If the size of any dimension is `0`, then F is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, F, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`p` — Prototype

logical variable

Prototype, specified as a logical variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`F` — Output of logical zeros

scalar | vector | matrix | N-D array

Output of logical zeros, returned as a scalar, vector, matrix, or N-D array.

Data Types: `logical`

More About

Tips

- `false(n)` is much faster and more memory efficient than `logical(zeros(n))`.
- "Class Support for Array-Creation Functions"

See Also

`logical` | `true`

Introduced before R2006a

fclose

Close one or all open files

Syntax

```
fclose(fileID)
fclose('all')
status = fclose(...)
```

Description

`fclose(fileID)` closes an open file. *`fileID`* is an integer file identifier obtained from `fopen`.

`fclose('all')` closes all open files.

`status` `= fclose(...)` returns a *`status`* of `0` when the close operation is successful. Otherwise, it returns `-1`.

See Also

`ferror` | `fopen` | `frewind` | `fread` | `fwrite` | `fseek` | `ftell` | `feof` | `fscanf` | `fprintf`

Introduced before R2006a

fclose (serial)

Disconnect serial port object from device

Syntax

```
fclose(obj)
```

Description

`fclose(obj)` disconnects `obj` from the device, where `obj` is a serial port object or an array of serial port objects.

Examples

This example creates the serial port object `s` on a Windows platform, connects `s` to the device, writes and reads text data, and then disconnects `s` from the device using `fclose`.

```
s = serial('COM1');
fopen(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fclose(s)
```

At this point, the device is available to be connected to a serial port object. If you no longer need `s`, you should remove from memory with the `delete` function, and remove it from the workspace with the `clear` command.

More About

Tips

If `obj` was successfully disconnected, then the `Status` property is configured to `closed` and the `RecordStatus` property is configured to `off`. You can reconnect `obj` to the device using the `fopen` function.

An error is returned if you issue `fclose` while data is being written asynchronously. In this case, you should abort the write operation with the `stopasync` function, or wait for the write operation to complete.

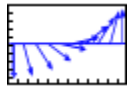
See Also

`clear` | `fopen` | `stopasync` | `delete` | `RecordStatus` | `Status`

Introduced before R2006a

feather

Plot velocity vectors



Syntax

```
feather(U,V)
feather(Z)
feather(...,LineSpec)
feather(axes_handle,...)
h = feather(...)
```

Description

A feather plot displays vectors emanating from equally spaced points along a horizontal axis. You express the vector components relative to the origin of the respective vector.

`feather(U,V)` displays the vectors specified by `U` and `V`, where `U` contains the x components as relative coordinates, and `V` contains the y components as relative coordinates.

`feather(Z)` displays the vectors specified by the complex numbers in `Z`. This is equivalent to `feather(real(Z),imag(Z))`.

`feather(...,LineSpec)` draws a feather plot using the line type, marker symbol, and color specified by `LineSpec`.

`feather(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = feather(...)` returns the handles to line objects in `h`.

Examples

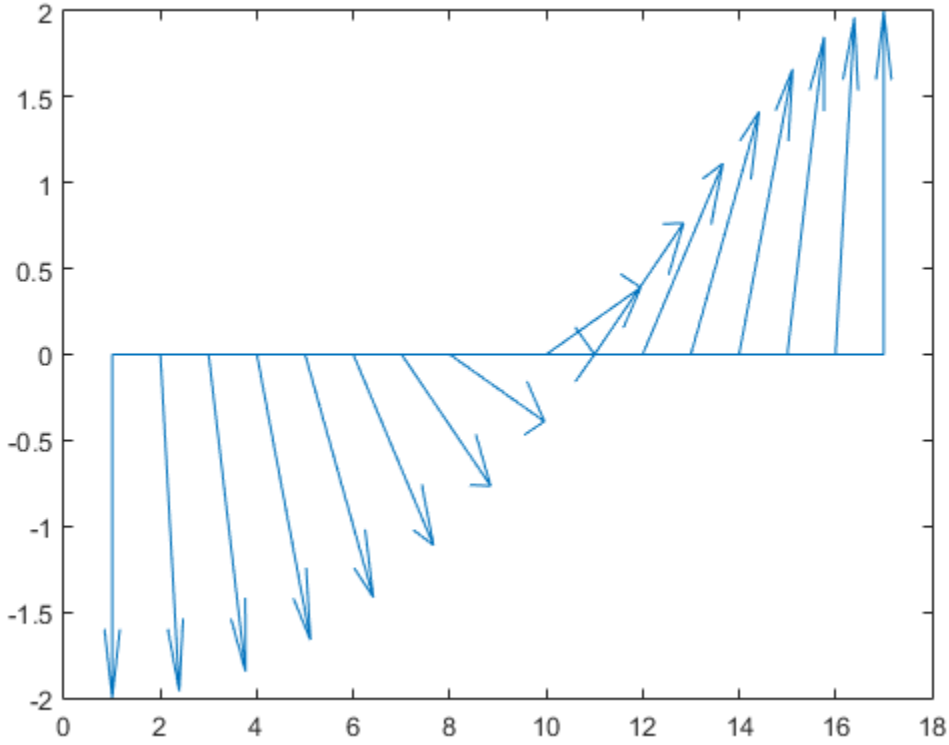
Create Feather Plot

Define `theta` as values between -2π and 2π. Define `r` as a vector the same size as `theta`.

```
theta = -pi/2:pi/16:pi/2;
r = 2*ones(size(theta));
```

Create a feather plot showing the direction of `theta`. Since `feather` uses Cartesian coordinates, convert `theta` and `r` to Cartesian coordinates using `pol2cart`.

```
[u,v] = pol2cart(theta,r);
feather(u,v)
```



See Also

`compass` | `rose` | `LineSpec`

Introduced before R2006a

featureEdges

Class: TriRep

(Will be removed) Sharp edges of surface triangulation

Note: `featureEdges(TriRep)` will be removed in a future release. Use `featureEdges(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
FE = featureEdges(TR, filterangle)
```

Description

`FE = featureEdges(TR, filterangle)` returns an edge matrix `FE`. This method is typically used to extract the sharp edges in the surface mesh for the purpose of display. Edges that are shared by only one triangle and edges that are shared by more than two triangles are considered to be feature edges by default.

Note: This query is only applicable to triangular surface meshes.

Input Arguments

`TR`	Triangulation representation.
`filterangle`	The threshold angle in radians. Must be in the range $(0, \pi)$. `featureEdges` will return adjacent triangles that have a dihedral angle that deviates from π by an angle greater than `filterangle`.

Output Arguments

FE	Edges of the triangulation. FE is of size m-by-2 where m is the number of computed feature edges in the mesh. The vertices of the edges index into the array of points representing the vertex coordinates, TR.X.

Examples

Create a surface triangulation:

```
x = [0 0 0 0 0 3 3 3 3 3 3 6 6 6 6 6 9 9 9 9 9 9]';
y = [0 2 4 6 8 0 1 3 5 7 8 0 2 4 6 8 0 1 3 5 7 8]';
dt = DelaunayTri(x,y);
tri = dt(:,:);
```

Elevate the 2-D mesh to create a surface:

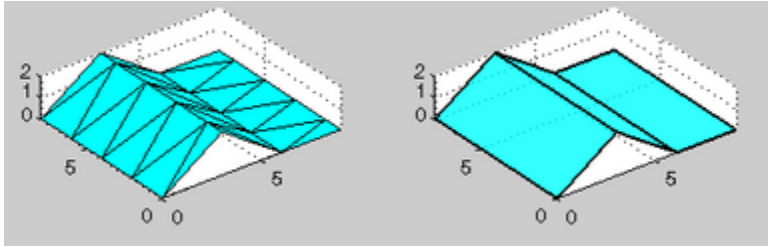
```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0]';
subplot(1,2,1);
trisurf(tri,x,y,z, 'FaceColor', 'cyan');
axis equal;
% TRISURF display of surface mesh
% showing mesh edges
```

Compute the feature edges using a filter angle of pi/6:

```
tr = TriRep(tri, x,y,z);
fe = featureEdges(tr,pi/6)';
subplot(1,2,2);
trisurf(tr, 'FaceColor', 'cyan', 'EdgeColor','none', ...
     'FaceAlpha', 0.8); axis equal;
```

Add the feature edges to the plot:

```
hold on;
plot3(x(fe), y(fe), z(fe), 'k', 'LineWidth',1.5);
hold off;
% TRISURF display of surface mesh
% suppressing mesh edges
% and showing feature edges
```



See Also

`delaunayTriangulation` | `edges` | `triangulation`

feof

Test for end-of-file

Syntax

```
status = feof(fileID)
```

Description

`status = feof(fileID)` returns `1` if a previous operation set the end-of-file indicator for the specified file. Otherwise, `feof` returns `0`. *`fileID`* is an integer file identifier obtained from `fopen`.

Opening an empty file does *not* set the end-of-file indicator. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Examples

Read `bench.dat`, which contains MATLAB benchmark data, one character at a time:

```
fid = fopen('bench.dat');

k = 0;
while ~feof(fid)
    curr = fscanf(fid,'%c',1);
    if ~isempty(curr)
       k = k+1;
       benchstr(k) = curr;
    end
end

fclose(fid);
```

More About

- "Testing for End of File (EOF)"

See Also

`fclose` | `ferror` | `fopen` | `frewind` | `fseek` | `ftell`

Introduced before R2006a

ferror

Information about file I/O errors

Syntax

```
message = ferror(fileID)
[message, errnum] = ferror(fileID)
[...] = ferror(fileID, 'clear')
```

Description

`message = ferror(fileID)` returns the error message for the most recent file I/O operation on the specified file. If the operation was successful, `message` is an empty string. `fileID` is an integer file identifier obtained from `fopen`, or an identifier reserved for standard input (`0`), standard output (`1`), or standard error (`2`).

`[message, errnum] = ferror(fileID)` returns the error number. If the most recent file I/O operation was successful, `errnum` is `0`. Negative error numbers correspond to MATLAB error messages. Positive error numbers correspond to C library error messages for your system.

`[...] = ferror(fileID, 'clear')` clears the error indicator for the specified file.

See Also

`fclose` | `fopen` | `fseek` | `ftell` | `feof` | `fscanf` | `fprintf` | `fread` | `fwrite`

Introduced before R2006a

feval

Evaluate function

Syntax

```
[y1, y2, ...] = feval(fhandle, x1, ..., xn)
[y1, y2, ...] = feval(fname, x1, ..., xn)
```

Description

`[y1, y2, ...] = feval(fhandle, x1, ..., xn)` evaluates the function handle, `fhandle`, using arguments `x1` through `xn`. If the function handle is bound to more than one built-in or `.m` function, (that is, it represents a set of overloaded functions), then the data type of the arguments `x1` through `xn` determines which function is dispatched to.

Note It is not necessary to use `feval` to call a function by means of a function handle. This is explained in "Calling a Function Using Its Handle" in the MATLAB Programming Fundamentals documentation.

`[y1, y2, ...] = feval(fname, x1, ..., xn)`. If `fname` is a quoted string containing the name of a function (usually defined within file having a .m file extension), then `feval(fname, x1, ..., xn)` evaluates that function at the given arguments. The `fname` parameter must be a simple function name; it cannot contain path information.

Examples

The following example passes a function handle, `fhandle`, in a call to `fminbnd`. The `fhandle` argument is a handle to the `humps` function.

```
fhandle = @humps;
x = fminbnd(fhandle, 0.3, 1);
```

The `fminbnd` function uses `feval` to evaluate the function handle that was passed in.

```
function [xf, fval, exitflag, output] = ...
    fminbnd(funfcn, ax, bx, options, varargin)
             .
             .
             .
fx = feval(funfcn, x, varargin{:});
```

More About

Tips

The following two statements are equivalent.

```
[V,D] = eig(A)
[V,D] = feval(@eig, A)
```

Nested functions are not accessible to `feval`. To call a nested function, you must either call it directly by name, or construct a function handle for it using the `@` operator.

See Also

`assignin` | `function_handle` | `functions` | `builtin` | `eval` | `evalin`

Introduced before R2006a

fewerbins

Decrease number of histogram bins

Syntax

```
N = fewerbins(h)
```

Description

`N = fewerbins(h)` decreases the number of bins in histogram `h` by 10% (rounded down to the nearest integer) and returns the new number of bins.

Examples

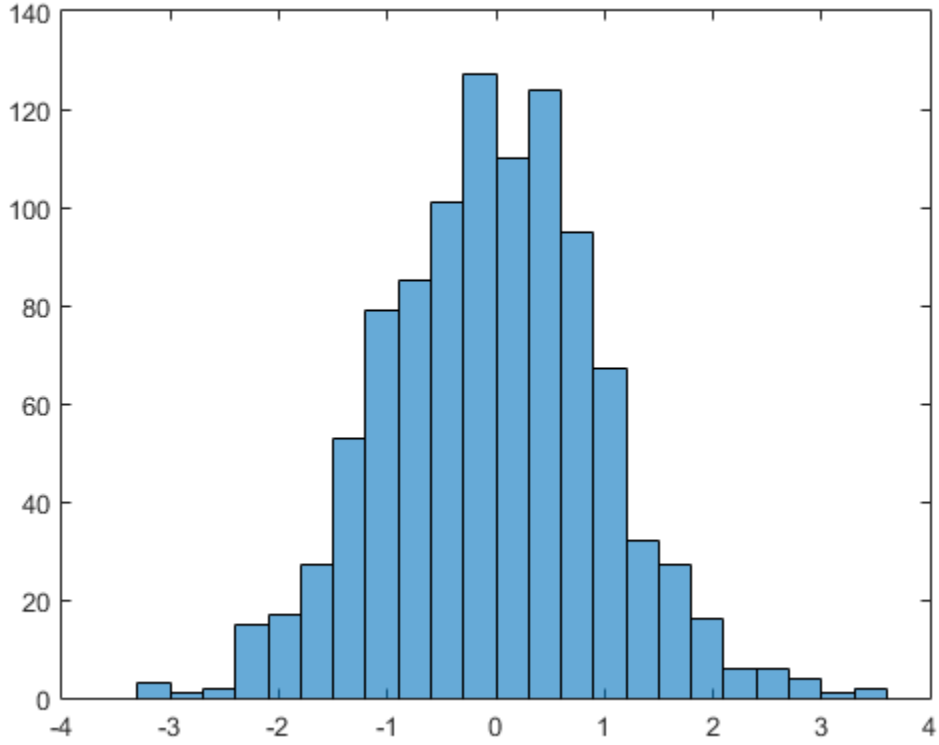
Decrease Number of Histogram Bins

Plot a histogram of 1,000 random numbers and return a handle to the histogram object.

```
x = randn(1000,1);
h = histogram(x)
```

```
h =

  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]
```

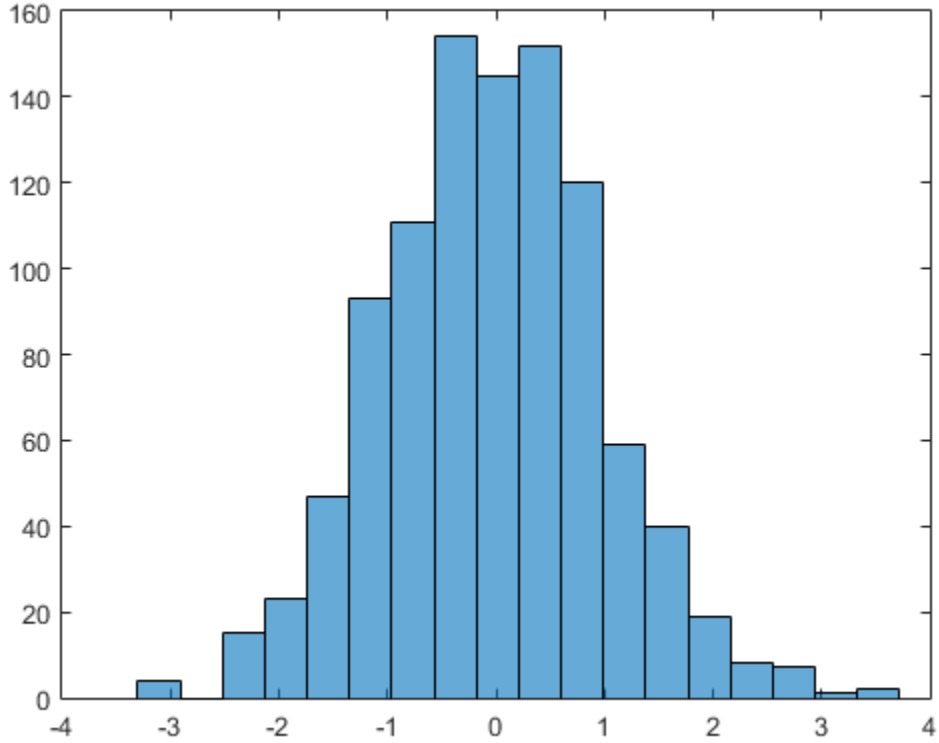
```
Use GET to show all properties
```



Use `fewerbins` to decrease the number of bins in the histogram.

```
fewerbins(h);
fewerbins(h)
```

```
ans =

    18
```



Input Arguments

h — Input histogram

histogram object

Input histogram, specified as a histogram object. For more information, see Using histogram Objects.

Output Arguments

`N` — Number of bins

scalar

Number of bins, returned as a scalar. `N` is the new number of bins in the histogram after decrease.

More About

- Using histogram Objects
- Histogram Properties

See Also

`histcounts` | `histogram` | `morebins`

Introduced in R2014b

Feval (COM)

Evaluate MATLAB function in Automation server

Syntax

IDL Method Signature

```
HRESULT Feval([in] BSTR functionname, [in] long nargout,
[out] VARIANT* result, [in, optional] VARIANT arg1, arg2, ...)
```

Microsoft Visual Basic Client

```
Feval(String functionname, long numout,
arg1, arg2, ...) As Object
```

MATLAB Client

```
result = Feval(h,'functionname',numout,arg1,arg2,...)
```

Description

`result = Feval(h,'functionname',numout,arg1,arg2,...)` executes the MATLAB function specified by the string `functionname` in the Automation server attached to handle `h`. The function name is case-sensitive.

COM functions are available on Microsoft Windows systems only.

Indicate the number of outputs to be returned by the function in a `1`-by-`1` `double` array, `numout`. The server returns output from the function in the cell array, `result`.

You can specify as many as 32 input arguments to be passed to the function. These arguments follow `numout` in the `Feval` argument list. The following table shows ways to pass an argument.

Passing Mechanism	Description
Pass the value itself	To pass any numeric or string value, specify the value in the `Feval` argument list: `a = Feval(h,'sin',1,-pi:0.01:pi);`

Pass a client variable	To pass an argument assigned to a variable in the client, specify the variable name alone:
	
	`x = -pi:0.01:pi;`
	`a = Feval(h,'sin',1,x);`

Reference a server variable	To reference a variable defined in the server, specify the variable name followed by an equals (=) sign:
	
	`PutWorkspaceData(h,'x','base',-pi:0.01:pi);`
	`a = Feval(h,'sin',1,'x=');`
	
	MATLAB does not reassign the server variable.

Visual Basic .NET Examples

Passing Arguments

This example shows how to pass arguments using `Feval` to execute MATLAB commands on a MATLAB Automation server from a Visual Basic® .NET client.

- Pass two strings to the MATLAB function `strcat` on the server:

```
Dim Matlab As Object
Dim out As Object
out = Nothing
Matlab = CreateObject("matlab.application")
Matlab.Feval("strcat",1,out,"hello"," world")
```

- Define `clistr` locally and pass this variable:

```
Dim clistr As String
clistr = " world"
Matlab.Feval("strcat",1,out,"hello",clistr)
```

- Pass the name of a variable defined on the server:

```
Matlab.PutCharArray("srvstr","base"," world")
Matlab.Feval("strcat",1,out,"hello","srvstr=")
```

Defining Feval Return Values

`Feval` returns data from the evaluated function in a cell array. The cell array has one row for every return value. You control the number of return values using the `numout` argument.

```
Dim Matlab As Object
Dim out As Object
Matlab = CreateObject("matlab.application")
Matlab.Feval("fileparts",3,out,"d:\work\ConsoleApp.cpp")
```

Creating Server Variables

Use variables defined in the client, `rows` and `cols`, to modify a server variable, `A`.

- Create a matrix, `A`, in the server.

  ```
  Dim Matlab As Object
  Dim server_version As String
  Matlab = CreateObject("matlab.application")
  MatLab.PutWorkspaceData("A","base",rand(6))
  ```

- Reshape `A`.

  ```
  rows = 6
  cols = 3
  Matlab.Feval("reshape",0,"A=",rows,cols)
  ```

  MATLAB interprets `A` in the expression `'A='` as a server variable name.

- The `reshape` function does not modify variable `A`.

  ```
  Matlab.GetWorkspaceData("A","base",B)
  ```

  `A` is unchanged.

- To get the result of the `reshape` function, use the `numout` argument to assign the value to `C`.

  ```
  Matlab.Feval("reshape",1,"A=",rows,cols,C)
  ```

More About

Tips

To display the output from `Feval` in the client window, assign a return value.

See Also

`Execute` | `PutFullMatrix` | `GetFullMatrix` | `PutCharArray` | `GetCharArray`

Introduced before R2006a

fft

Fast Fourier transform

Syntax

```
Y = fft(x)
Y = fft(X,n)
Y = fft(X,[],dim)
Y = fft(X,n,dim)
```

Definitions

The functions `Y = fft(x)` and `y = ifft(X)` implement the transform and inverse transform pair given for vectors of length N by:

$$X(k) = \sum_{j=1}^{N} x(j)\omega_N^{(j-1)(k-1)}$$

$$x(j) = (1/N)\sum_{k=1}^{N} X(k)\omega_N{}^{-(j-1)(k-1)}$$

where

$$\omega_N = e^{(-2\pi i)/N}$$

is an Nth root of unity.

Description

`Y = fft(x)` returns the discrete Fourier transform (DFT) of vector `x`, computed with a fast Fourier transform (FFT) algorithm.

If the input `X` is a matrix, `Y = fft(X)` returns the Fourier transform of each column of the matrix.

If the input `X` is a multidimensional array, `fft` operates on the first nonsingleton dimension.

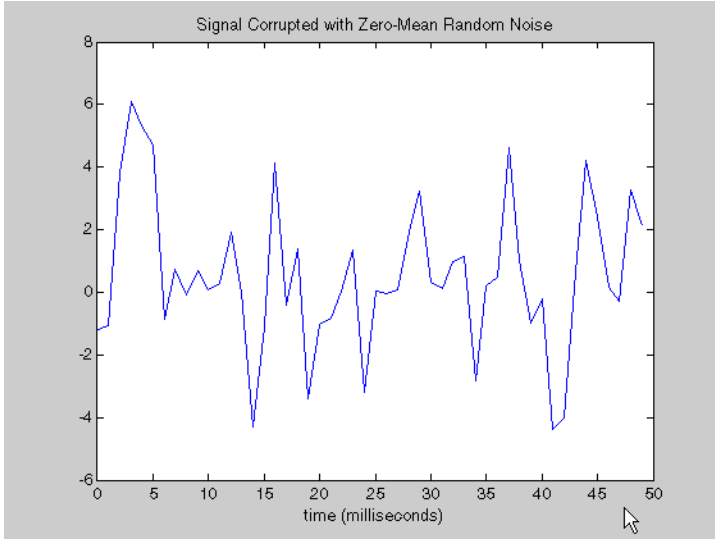
`Y = fft(X,n)` returns the `n`-point DFT. `fft(X)` is equivalent to `fft(X, n)` where `n` is the size of `X` in the first nonsingleton dimension. If the length of `X` is less than `n`, `X` is padded with trailing zeros to length `n`. If the length of `X` is greater than `n`, the sequence `X` is truncated. When `X` is a matrix, the length of the columns are adjusted in the same manner.

`Y = fft(X,[],dim)` and `Y = fft(X,n,dim)` applies the FFT operation across the dimension `dim`.

Examples

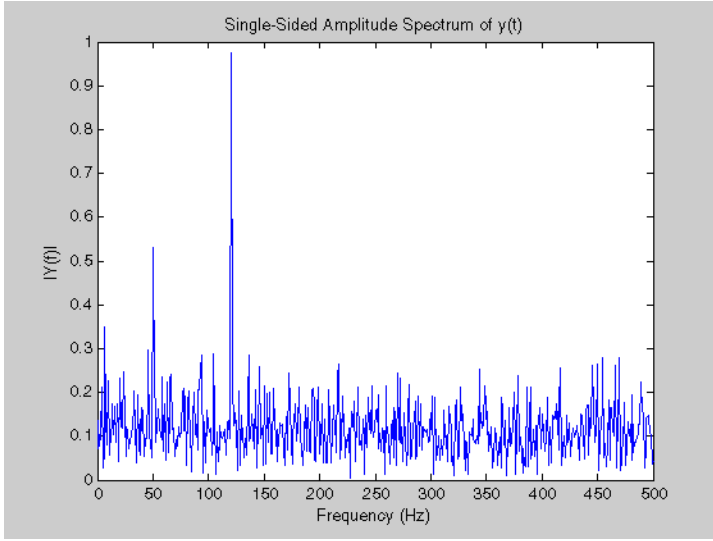
A common use of Fourier transforms is to find the frequency components of a signal buried in a noisy time domain signal. Consider data sampled at 1000 Hz. Form a signal containing a 50 Hz sinusoid of amplitude 0.7 and 120 Hz sinusoid of amplitude 1 and corrupt it with some zero-mean random noise:

```
Fs = 1000;                    % Sampling frequency
T = 1/Fs;                     % Sample time
L = 1000;                     % Length of signal
t = (0:L-1)*T;                % Time vector
% Sum of a 50 Hz sinusoid and a 120 Hz sinusoid
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(size(t));     % Sinusoids plus noise
plot(Fs*t(1:50),y(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (milliseconds)')
```



It is difficult to identify the frequency components by looking at the original signal. Converting to the frequency domain, the discrete Fourier transform of the noisy signal `y` is found by taking the fast Fourier transform (FFT):

```
NFFT = 2^nextpow2(L); % Next power of 2 from length of y
Y = fft(y,NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2+1);

% Plot single-sided amplitude spectrum.
plot(f,2*abs(Y(1:NFFT/2+1)))
title('Single-Sided Amplitude Spectrum of y(t)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
```



The main reason the amplitudes are not exactly at 0.7 and 1 is because of the noise. Several executions of this code (including recomputation of `y`) will produce different approximations to 0.7 and 1. The other reason is that you have a finite length signal. Increasing `L` from 1000 to 10000 in the example above will produce much better approximations on average.

Data Type Support

`fft` supports inputs of data types `double` and `single`. If you call `fft` with the syntax `y = fft(X, ...)`, the output `y` has the same data type as the input `X`.

More About

Algorithms

The FFT functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`) are based on a library called FFTW [3],[4]. To compute an N-point DFT when N is composite (that is, when $N = N_1N_2$), the FFTW library decomposes the problem using the Cooley-Tukey algorithm [1], which first computes N_1 transforms of size N_2, and then computes N_2 transforms of size N_1. The

decomposition is applied recursively to both the N_1- and N_2-point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey [5], a prime factor algorithm [6], and a split-radix algorithm [2]. The particular factorization of N is chosen heuristically.

When N is a prime number, the FFTW library first decomposes an N-point problem into three $(N-1)$-point problems using Rader's algorithm [7]. It then uses the Cooley-Tukey decomposition described above to compute the $(N-1)$-point DFTs.

For most N, real-input DFTs require roughly half the computation time of complex-input DFTs. However, when N has large prime factors, there is little or no speed difference.

The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fft` using the utility function `fftw`, which controls the optimization of the algorithm used to compute an FFT of a particular size and dimension.

References

[1] Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Computation of the Complex Fourier Series,"*Mathematics of Computation*, Vol. 19, April 1965, pp. 297-301.

[2] Duhamel, P. and M. Vetterli, "Fast Fourier Transforms: A Tutorial Review and a State of the Art," *Signal Processing*, Vol. 19, April 1990, pp. 259-299.

[3] FFTW (http://www.fftw.org)

[4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

[5] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 611.

[6] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 619.

[7] Rader, C. M., "Discrete Fourier Transforms when the Number of Data Samples Is Prime," *Proceedings of the IEEE*, Vol. 56, June 1968, pp. 1107-1108.

See Also

`fft2` | `fftn` | `fftw` | `fftshift` | `ifft` | `filter`

Introduced before R2006a

fft2

2-D fast Fourier transform

Syntax

```
Y = fft2(X)
Y = fft2(X,m,n)
```

Description

`Y = fft2(X)` returns the two-dimensional discrete Fourier transform (DFT) of `X`. The DFT is computed with a fast Fourier transform (FFT) algorithm. The result, `Y`, is the same size as `X`.

If the dimensionality of `X` is greater than `2`, the `fft2` function returns the 2-D DFT for each higher dimensional slice of `X`. For example, if `size(X) = [100 100 3]`, then `fft2` computes the DFT of `X(:,:,1)`, `X(:,:,2)` and `X(:,:,3)`.

`Y = fft2(X,m,n)` truncates `X`, or pads `X` with zeros to create an `m`-by-`n` array before doing the transform. The result is `m`-by-`n`.

Data Type Support

`fft2` supports inputs of data types `double` and `single`. If you call `fft2` with the syntax `y = fft2(X, ...)`, the output `y` has the same data type as the input `X`.

More About

Algorithms

`fft2(X)` can be simply computed as

```
fft(fft(X).').'
```

This computes the one-dimensional DFT of each column `X`, then of each row of the result. The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fft2` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

See Also

`fft` | `fftn` | `fftw` | `fftshift` | `ifft2`

Introduced before R2006a

fftn

N-D fast Fourier transform

Syntax

```
Y = fftn(X)
Y = fftn(X,siz)
```

Description

`Y = fftn(X)` returns the discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`Y = fftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the transform. The size of the result `Y` is `siz`.

Data Type Support

`fftn` supports inputs of data types `double` and `single`. If you call `fftn` with the syntax `y = fftn(X, ...)`, the output `y` has the same data type as the input `X`.

More About

Algorithms

`fftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = fft(Y,[],p);
end
```

This computes in-place the one-dimensional fast Fourier transform along each dimension of `X`. The execution time for `fft` depends on the length of the transform. It is fastest

for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fftn` using the utility function `fftw`, which controls the optimization of the algorithm used to compute an FFT of a particular size and dimension.

See Also

`fft` | `fft2` | `fftw` | `ifftn`

Introduced before R2006a

fftshift

Shift zero-frequency component to center of spectrum

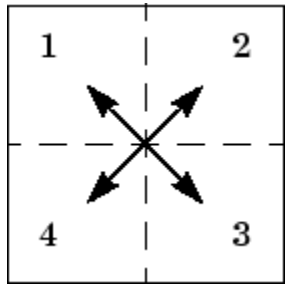
Syntax

```
Y = fftshift(X)
Y = fftshift(X,dim)
```

Description

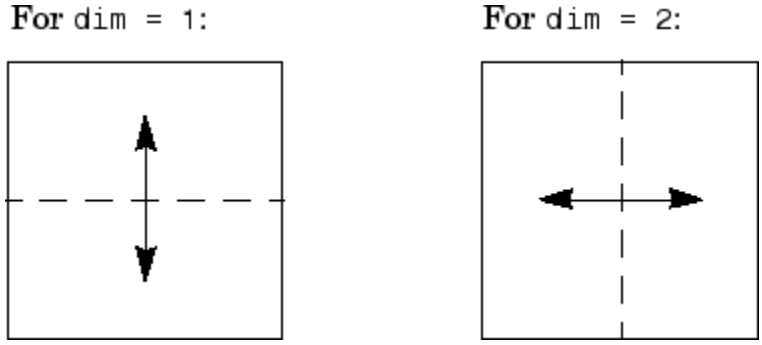
`Y = fftshift(X)` rearranges the outputs of `fft`, `fft2`, and `fftn` by moving the zero-frequency component to the center of the array. It is useful for visualizing a Fourier transform with the zero-frequency component in the middle of the spectrum.

For vectors, `fftshift(X)` swaps the left and right halves of `X`. For matrices, `fftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth.



For higher-dimensional arrays, `fftshift(X)` swaps “half-spaces” of `X` along each dimension.

`Y = fftshift(X,dim)` applies the `fftshift` operation along the dimension `dim`.



Note: `ifftshift` will undo the results of `fftshift`. If the matrix `X` contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original `X`. Simply performing `fftshift(X)` twice will not produce `X`.

Examples

For any matrix `X`

```
Y = fft2(X)
```

has `Y(1,1) = sum(sum(X))`; the zero-frequency component of the signal is in the upper-left corner of the two-dimensional FFT. For

```
Z = fftshift(Y)
```

this zero-frequency component is near the center of the matrix.

The difference between `fftshift` and `ifftshift` is important for input sequences of odd-length.

```
N = 5;
X = 0:N-1;
Y = fftshift(fftshift(X));
Z = ifftshift(fftshift(X));
```

Notice that `Z` is a correct replica of `X`, but `Y` is not.

```
isequal(X,Y),isequal(X,Z)
```

```
ans =

     0

ans =

     1
```

See Also

circshift | fft | fft2 | fftn | ifftshift

Introduced before R2006a

fftw

Interface to FFTW library run-time algorithm tuning control

Syntax

```
fftw('planner', method)
method = fftw('planner')
str = fftw('dwisdom')
str = fftw('swisdom')
fftw('dwisdom', str)
fftw('swisdom', str)
```

Description

`fftw` enables you to optimize the speed of the MATLAB FFT functions `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, and `ifftn`. You can use `fftw` to set options for a tuning algorithm that experimentally determines the fastest algorithm for computing an FFT of a particular size and dimension at run time. MATLAB software records the optimal algorithm in an internal data base and uses it to compute FFTs of the same size throughout the current session. The tuning algorithm is part of the FFTW library that MATLAB software uses to compute FFTs.

`fftw('planner', method)` sets the method by which the tuning algorithm searches for a good FFT algorithm when the dimension of the FFT is not a power of 2. You can specify `method` to be one of the following. The default method is `estimate`:

- `'estimate'`
- `'measure'`
- `'patient'`
- `'exhaustive'`
- `'hybrid'`

When you call `fftw('planner', method)`, the next time you call one of the FFT functions, such as `fft`, the tuning algorithm uses the specified method to optimize the

FFT computation. Because the tuning involves trying different algorithms, the first time you call an FFT function, it might run more slowly than if you did not call `fftw`. However, subsequent calls to any of the FFT functions, for a problem of the same size, often run more quickly than they would without using `fftw`.

Note The FFT functions only use the optimal FFT algorithm during the current MATLAB session. "Reusing Optimal FFT Algorithms" on page 1-2537 explains how to reuse the optimal algorithm in a future MATLAB session.

If you set the method to `'estimate'`, the FFTW library does not use run-time tuning to select the algorithms. The resulting algorithms might not be optimal.

If you set the method to `'measure'`, the FFTW library experiments with many different algorithms to compute an FFT of a given size and chooses the fastest. Setting the method to `'patient'` or `'exhaustive'` has a similar result, but the library experiments with even more algorithms so that the tuning takes longer the first time you call an FFT function. However, subsequent calls to FFT functions are faster than with `'measure'`.

If you set `'planner'` to `'hybrid'`, MATLAB software

- Sets method to `'measure'` method for FFT dimensions 8192 or smaller.
- Sets method to `'estimate'` for FFT dimensions greater than 8192.

`method = fftw('planner')` returns the current planner method.

`str = fftw('dwisdom')` returns the information in the FFTW library's internal double-precision database as a string. The string can be saved and then later reused in a subsequent MATLAB session using the next syntax.

`str = fftw('swisdom')` returns the information in the FFTW library's internal single-precision database as a string.

`fftw('dwisdom', str)` loads `fftw` wisdom represented by the string `str` into the FFTW library's internal double-precision wisdom database. `fftw('dwisdom','')` or `fftw('dwisdom',[])` clears the internal wisdom database.

`fftw('swisdom', str)` loads `fftw` wisdom represented by the string `str` into the FFTW library's internal single-precision wisdom database. `fftw('swisdom','')` or `fftw('swisdom',[])` clears the internal wisdom database.

Note on large powers of 2 For FFT dimensions that are powers of 2, between 2^{14} and 2^{22}, MATLAB software uses special preloaded information in its internal database to optimize the FFT computation. No tuning is performed when the dimension of the FTT is a power of 2, unless you clear the database using the command `fftw('wisdom', [])`.

For more information about the FFTW library, see http://www.fftw.org.

Examples

Comparison of Speed for Different Planner Methods

The following example illustrates the run times for different settings of `planner`. The example first creates some data and applies `fft` to it using the default method, `estimate`.

```
t=0:.001:5;
x = sin(2*pi*50*t)+sin(2*pi*120*t);
y = x + 2*randn(size(t));

tic; Y = fft(y,1458); toc
Elapsed time is 0.000521 seconds.
```

If you execute the commands

```
tic; Y = fft(y,1458); toc
Elapsed time is 0.000151 seconds.
```

a second time, MATLAB software reports the elapsed time as essentially 0. To measure the elapsed time more accurately, you can execute the command `Y = fft(y,1458)` 1000 times in a loop.

```
tic; for k=1:1000
Y = fft(y,1458);
end; toc
Elapsed time is 0.056532 seconds.
```

This tells you that it takes on order of 1/10000 of a second to execute `fft(y, 1458)` a single time.

For comparison, set `planner` to `patient`. Since this `planner` explores possible algorithms more thoroughly than `hybrid`, the first time you run `fft`, it takes longer to compute the results.

```
fftw('planner','patient')
tic;Y = fft(y,1458);toc
Elapsed time is 0.100637 seconds.
```

However, the next time you call `fft`, it runs at approximately the same speed as before you ran the method `patient`.

```
tic;for k=1:1000
Y=fft(y,1458);
end;toc
Elapsed time is 0.057209 seconds.
```

Reusing Optimal FFT Algorithms

In order to use the optimized FFT algorithm in a future MATLAB session, first save the "wisdom" using the command

```
str = fftw('wisdom')
```

You can save `str` for a future session using the command

```
save str
```

The next time you open a MATLAB session, load `str` using the command

```
load str
```

and then reload the "wisdom" into the FFTW database using the command

```
fftw('wisdom', str)
```

See Also

`fft` | `fft2` | `fftn` | `ifft` | `ifft2` | `ifftn` | `fftshift`

Introduced before R2006a

fgetl

Read line from file, removing newline characters

Syntax

```
tline = fgetl(fileID)
```

Description

`tline = fgetl(fileID)` returns the next line of the specified file, removing the newline characters. `fileID` is an integer file identifier obtained from `fopen`. `tline` is a text string unless the line contains only the end-of-file marker. In this case, `tline` is the numeric value `-1`.

`fgetl` reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use `fopen`.

Examples

Read and display the file `fgetl.m` one line at a time:

```
fid = fopen('fgetl.m');

tline = fgetl(fid);
while ischar(tline)
    disp(tline)
    tline = fgetl(fid);
end

fclose(fid);
```

Compare these results to the `fgets` example, which replaces the calls to `fgetl` with `fgets`.

More About

- "Testing for EOF with fgetl and fgets"

See Also

`fclose` | `feof` | `ferror` | `fgets` | `fopen` | `fprintf` | `fread` | `fscanf` | `fwrite`

Introduced before R2006a

fgetl (serial)

Read line of ASCII text from device and discard terminator

Syntax

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
```

Description

`tline = fgetl(obj)` reads one line of ASCII text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data does not include the terminator with the text line. To include the terminator, use `fgets`.

`[tline,count] = fgetl(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgetl(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Examples

On a Windows platform, create the serial port object `s`, connect `s` to a Tektronix® TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
```

```
ans =
    17
```

Use `fgetl` to read the data returned from the previous write operation, and discard the terminator.

```
settings = fgetl(s)
```

```
settings =
9600;0;0;NONE;LF
```

```
length(settings)
```

```
ans =
    16
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

More About

Tips

Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgetl` is issued.

Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.

- The time specified by the `Timeout` property passes.
- The input buffer is filled.

See Also

`fgets` | `fopen` | `BytesAvailable` | `InputBufferSize` | `ReadAsyncMode` | `Status` | `Terminator` | `Timeout` | `ValuesReceived`

Introduced before R2006a

fgets

Read line from file, keeping newline characters

Syntax

```
tline = fgets(fileID)
tline = fgets(fileID, nchar)
```

Description

`tline = fgets(fileID)` reads the next line of the specified file, including the newline characters. `fileID` is an integer file identifier obtained from `fopen`. `tline` is a text string unless the line contains only the end-of-file marker. In this case, `tline` is the numeric value `-1`. `fgets` reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use `fopen`.

`tline = fgets(fileID, nchar)` returns at most `nchar` characters of the next line. `tline` does not include any characters after the newline characters or the end-of-file marker.

Examples

Read and display the file `fgets.m`. Because `fgets` keeps newline characters and `disp` adds a newline character, this code displays the file with double-spacing:

```
fid = fopen('fgets.m');

tline = fgets(fid);
while ischar(tline)
    disp(tline)
    tline = fgets(fid);
end

fclose(fid);
```

Compare these results to the `fgetl` example, which replaces the calls to `fgets` with `fgetl`.

More About

- "Testing for EOF with fgetl and fgets"

See Also

`fclose` | `feof` | `ferror` | `fgetl` | `fopen` | `fprintf` | `fread` | `fscanf` | `fwrite`

Introduced before R2006a

fgets (serial)

Read line of text from device and include terminator

Syntax

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
```

Description

`tline = fgets(obj)` reads one line of text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Examples

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
```

```
ans =
```

```
    17
```

Use `fgets` to read the data returned from the previous write operation, and include the terminator.

```
settings = fgets(s)
```

```
settings =
9600;0;0;NONE;LF
```

```
length(settings)
```

```
ans =
    17
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

More About

Tips

Before you can read text from the device, it must be connected to `obj` with the `fopen`function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgets` is issued.

Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.
- The time specified by the `Timeout` property passes.

- The input buffer is filled.

See Also

`fgetl` | `fopen` | `BytesAvailable` | `BytesAvailableFcn` | `InputBufferSize` | `Status` | `Terminator` | `Timeout` | `ValuesReceived`

Introduced before R2006a

fieldnames

Field names of structure, or public fields of object

Syntax

```
names = fieldnames(s)
names = fieldnames(obj)
names = fieldnames(obj,'-full')
```

Description

`names = fieldnames(s)` returns a cell array of strings containing the names of the fields in structure `s`.

`names = fieldnames(obj)` returns a cell array of strings containing the names of the public properties of `obj`. MATLAB objects can overload `fieldnames` and define their own behavior.

`names = fieldnames(obj,'-full')` returns a cell array of strings containing the name, type, attributes, and inheritance of the properties of `obj`. Only supported for COM or Java objects.

Examples

Structure Fields

Create a structure array and view its fields.

```
s(1,1).name = 'alice';
s(1,1).ID = 0;
s(2,1).name = 'gertrude';
s(2,1).ID = 1;

names = fieldnames(s)
```

```
names =

    'name'
    'ID'
```

Java Object Properties

Create a Java® object and view its public properties.

```
obj = java.lang.Integer(0);
names = fieldnames(obj)
```

```
names =

    'MIN_VALUE'
    'MAX_VALUE'
    'TYPE'
    'SIZE'
```

More About

- "Generate Field Names from Variables"

See Also

`setfield` | `getfield` | `isfield` | `orderfields` | `rmfield`

Introduced before R2006a

figure

Create figure window

Syntax

```
figure
figure('PropertyName',propertyvalue,...)
figure(h)
h = figure(...)
```

Properties

For a list of properties, see Figure Properties.

Description

`figure` creates a new figure window using default property values. This new figure window becomes the current figure, and it displays on top of all other figures on the screen. The title of the figure is an integer value that is not already used by an existing figure. MATLAB saves this integer value in the figure's `Number` property.

`figure('PropertyName',propertyvalue,...)` creates a new figure window using specific property values. For a list of available properties, see Figure Properties. MATLAB uses default values for any properties that you do not explicitly define as arguments.

`figure(h)` does one of the following:

- If `h` is the handle or the `Number` property value of an existing figure, then `figure(h)` makes that existing figure the current figure, makes it visible, and moves it on top of all other figures on the screen. The current figure is the target for graphics output.
- If `h` is not the handle and is not the `Number` property value of an existing figure, but is an integer, then `figure(h)` creates a figure object and assigns its `Number` property the value `h`.

- If `h` is not the handle to a figure and is not a positive integer, then MATLAB returns an error.

`h = figure(...)` returns the handle to the figure object.

Examples

Specifying Figure Size and Screen Location

To create a figure window that is one quarter the size of your screen and is positioned in the upper left corner, use the root object's `ScreenSize` property to determine the size. The `ScreenSize` property value is a four-element vector: `[left bottom width height]`.

```
scrsz = get(groot,'ScreenSize');
figure('Position',[1 scrsz(4)/2 scrsz(3)/2 scrsz(4)/2])
```

To position the full figure window including the menu bar, title bar, tool bars, and outer edges, use the `OuterPosition` property in the same manner.

Specifying the Figure Window Title

You can add your own title to a figure by setting the `Name` property and turning off the `NumberTitle` property:

```
figure('Name','Simulation Plot Window','NumberTitle','off')
```

Setting Default Properties

You can set default figure properties only on the `groot` level.

```
set(groot,'DefaultFigure'Property',PropertyValue...)
```

where *`Property`* is the name of the figure property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access figure properties.

More About

Tips

Use dot notation to query or modify a particular figure's property. For example, `f.Units = 'inches'` sets the Units property of figure `f` to inches.

Figures can be docked in the desktop. The `DockControls` property determines whether you can dock the figure.

Making a Figure Current

The current figure is the target for graphics output. There are two ways to make a figure `h` the current figure.

- Make the figure `h` current, visible, and displayed on top of other figures:

  ```
  figure(h);
  ```

- Make the figure `h` current, but do not change its visibility or stacking with respect to other figures:

  ```
  set(groot,'CurrentFigure',h);
  ```

See Also

`axes` | `clf` | `close` | Figure Properties | `gcf` | `groot` | `ishghandle` | `uicontrol` | `uimenu`

Introduced before R2006a

Figure Properties

Control figure window appearance and behavior

Figures are windows that contain graphics or user interface components. Figure properties control the appearance and behavior of a particular instance of a figure. To modify aspects of a figure, change property values.

Starting in R2014b, you can use dot notation to query and set properties.

```
fig = figure;
u = fig.Units;
fig.Units = 'inches';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Figure Appearance

`Color` — Figure window background color

RGB triplet | short name | long name | `'none'`

The figure window background color, specified as an RGB triplet, a predefined color name string, or `'none'`. If you specify `'none'`, the figure background color appears black on screen, but if you print it, the background prints as though the figure window is transparent.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`

Long Name	Short Name	RGB Triplet
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Data Types: double | char

DockControls — Interactive figure docking

'on' (default) | 'off'

Interactive figure docking, specified as one of the following:

- 'on' — Figure can be docked in the MATLAB desktop. The **Desktop** > **Dock Figure** menu item and the Dock Figure button  in the menu bar are enabled.
- 'off' — MATLAB disables the **Desktop** > **Dock Figure** menu item and does not display the figure dock button.

  You cannot set the DockControls property to 'off' if the WindowStyle is set to 'docked'.

MenuBar — Figure menu bar display

'figure' (default) | 'none'

Figure menu bar display, specified as 'figure' or 'none'. The MenuBar property enables you to display or hide the default menus at the top of a figure window. Specify 'figure' to display the menu bar. Specify 'none' to hide it.

This property affects only default menus, and does not affect menus defined with the uimenu command.

Menu bars do not appear in figures whose WindowStyle property is set to 'Modal'. If a figure containing uimenu children is changed to 'Modal', the uimenu children still exist in the Children property of the figure. However, the uimenus do not display while WindowStyle is set to 'Modal'.

Note: If you do not want to display the default menus in the figure, then set this property to 'none' when you create the figure.

`Name` — Figure window title

`''` (default) | string

Figure window title, specified as a string. The Name property specifies the title that the figure window displays. By default, the figure title displays as `Figure 1`, `Figure 2`, and so on. When you set this property to a string, the figure title becomes `Figure:`*`n string`*. If you want only the Name value to display, set `IntegerHandle` or `NumberTitle` to `'off'`.

Example: `'Results'`

`NumberTitle` — Figure window title number

`'on'` (default) | `'off'`

Figure window title number, specified as `'on'` or `'off'`. The NumberTitle property determines whether MATLAB includes the string `Figure` *n* in the title bar, where *n* is the figure `Number` property value.

If you set `IntegerHandle` to `'off'`, then a number does not display in the figure window title, regardless of the NumberTitle property setting.

`ToolBar` — Figure toolbar display

`'auto'` (default) | `'figure'` | `'none'`

Figure toolbar display, specified as one of the following:

- `'auto'` — Uses the same value as the `MenuBar` property.
- `'figure'` — Toolbar displays.
- `'none'` — Toolbar does not display.

This property affects only the default toolbar. It does not affect other toolbars such as, the Camera Toolbar or Plot Edit Toolbar. Selecting **Figure Toolbar** from the figure **View** menu sets this property to `'figure'`.

Toolbars do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a toolbar is changed to `'Modal'`, the tool bar children still exist in the `Children` property of the figure. However, the toolbar does not display while WindowStyle is set to `'Modal'`.

Note: If you want to hide the default tool bar, then set this property to `'none'` when you create the figure.

`Visible` — Figure visibility

`'on'` (default) | `'off'`

Figure visibility, specified as `'on'` or `'off'`. The Visible property determines whether the figure displays on the screen. If the Visible property of a figure is set to `'off'`, the entire figure is invisible, but you can still specify and access its properties.

Changing the size of an invisible figure triggers the SizeChangedFcn callback when the figure becomes visible.

Note Changing the Visible property of a figure does *not* change the Visible property of its child components even though hiding the figure prevents its children from displaying.

`Clipping` — Clipping of child components to figure

`'on'` (default) | `'off'`

This property has no effect on figures.

Axes and Plot Appearance

`GraphicsSmoothing` — Axes graphics smoothing

`'on'` (default) | `'off'`

Axes graphics smoothing, specified as `'on'` or `'off'`. Smoothing reduces the appearance of jagged lines in an axes graphic. MATLAB applies a smoothing technique to an axes graphic (and the axes rulers) if `GraphicsSmoothing` is set to `'on'`, and either of these conditions is true:

- The `Renderer` property is set to `'painters'`.
- The Renderer property is set to `'opengl'` and your hardware card supports OpenGL.

If your axes graphic contains mostly vertical or horizontal lines, consider setting the GraphicsSmoothing property to `'on'` and the line or lines `AlignVertexCenters` property to `'on'`. The smoothing technique sacrifices some sharpness for smoothness, which might be particularly noticeable in such graphics.

Note: Graphics smoothing has no affect on text. MATLAB smooths text regardless of the value of the GraphicsSmoothing property.

`Renderer` — Rendering method used for screen display and printing

`'opengl'` (default) | `'painters'`

Rendering method used for screen display and printing, specified as one of these values:

- `'opengl'` — OpenGL renderer. This option enables MATLAB to access graphics hardware if it is available on your system. The OpenGL renderer displays objects sorted in front to back order, as seen on the monitor. Lines always draw in front of faces when at the same location on the plane of the monitor.
- `'painters'` — Painters renderer. This option works well for axes in a 2-D view. In 2-D, the Painters renderer sorts graphics objects by child order (order specified). In 3-D, the Painters renderer sorts objects in front to back order. However, it might not correctly draw intersecting polygons in 3-D.

Note: The `'zbuffer'` option has been removed. Use `'opengl'` or `'painters'` instead.

OpenGL Hardware and Software Implementations

OpenGL is available on all computers that run MATLAB since a software version of OpenGL is built-into MATLAB. However, if you have graphics hardware that supports a hardware-accelerated version of OpenGL, then MATLAB automatically uses the hardware-accelerated version to increase performance.

In some cases, MATLAB automatically uses software OpenGL even if a hardware version is available. For example, MATLAB uses the software version if it detects graphics hardware with known driver issues or detects that you are using a virtual machine or remote desktop on Windows.

MATLAB issues a warning if it cannot find a usable OpenGL library.

Software OpenGL Selection

To switch from hardware to software OpenGL, do the following:

- On Linux systems, start MATLAB with the command `matlab -softwareopengl`.
- On Windows systems, execute the command `opengl software` in MATLAB or start MATLAB with the command `matlab -softwareopengl`.

- On Macintosh systems, software OpenGL is not supported.

The following software versions are available:

- On Linux systems, MATLAB uses the software implementation of OpenGL that is included in the MATLAB distribution.
- On Windows, OpenGL is available as part of the operating system. If you experience problems with OpenGL, contact your graphics driver vendor to obtain the latest qualified version of OpenGL.
- On Macintosh systems, software OpenGL is not available.

Determine OpenGL Library Version

To determine the version and vendor of the OpenGL library that MATLAB is using on your system, type the following command at the MATLAB prompt:

```
opengl info
```

The returned information contains a line that indicates if MATLAB is using software OpenGL (`Software = true`) or hardware-accelerated OpenGL (`Software = false`).

This command also returns a string of extensions to the OpenGL specification that are available with the particular library MATLAB is using. Include this information if you report a bug.

Be aware that issuing the `opengl info` command causes MATLAB to initialize OpenGL.

XServer Connection Lost

When using Linux, if there is a break in the connection to the XServer, MATLAB can crash with a segmentation violation . If this happens, ensure that the system has the latest XServer installed.

On a Linux system, you also can try upgrading the OpenGL driver or starting MATLAB with software OpenGL using this command:

```
matlab -softwareopengl
```

`RendererMode` — Renderer selection

`'auto'` (default) | `'manual'`

Renderer selection, specified as:

- `'auto'` — MATLAB selects the rendering method for printing and screen display based on the size and complexity of the graphics objects in the figure.
- `'manual'` — MATLAB uses the renderer specified with the `Renderer` property.

MATLAB sets the `RendererMode` property to `'manual'` if you explicitly set the Renderer property to `'painters'` or `'opengl'`.

Color and Transparency Mapping

`Alphamap` — Transparency map for axes content of figure

array of 64 values from 0 to 1 (default) | array of finite alpha values from 0 to 1

Transparency map for axes content of figure, specified as an array of finite alpha values that progress linearly from 0 to 1. The size of the array can be m-by-1 or 1-by-m. MATLAB accesses alpha values by their row number. For example, an index of 1 specifies the first alpha value, an index of 2 specifies the second alpha value, and so on. Alphamaps can be any length.

Alphamaps affect the rendering of objects created with the `surface`, `image`, and `patch` functions, but do not affect other graphics objects.

`Colormap` — Color map for axes content of figure

parula (default) | `m`-by-3 array of RGB triplets

Color map for axes content of a figure, specified as an `m`-by-3 array of RGB (red, green, blue) triplets that define `m` individual colors. MATLAB accesses these colors by their row number. For example, an index of 1 specifies the first RGB triplet, an index of 2 specifies the second RGB triplet, and so on.

Colormaps affect the rendering of objects created with the `surface`, `image`, and `patch` functions, but generally do not affect other graphics objects.

Colormaps can be any length, but must be three columns wide.

Example: `[1 0 1; 0 0 1; 1 1 0]`

Location and Size

`Position` — Location and size of figure's drawable area

`[left bottom width height]`

Location and size of figure's drawable area, specified as the vector, `[left bottom width height]`. The drawable area is the inner area of the window, excluding the title bar, menu bar, and tool bars. This table describes each element in the Position vector.

Element	Description
`left`	Distance from the left edge of the primary display to the inner left edge of the figure window. This value can be negative on systems that have more than one monitor.
`bottom`	Distance from the bottom edge of the primary display to the inner bottom edge of the figure window. This value can be negative on systems that have more than one monitor.
`width`	Distance between the right and left inner edges of the figure.
`height`	Distance between the top and bottom inner edges of the figure.

All measurements are in units specified by the Units property.

You cannot specify the figure Position property when the figure is docked.

To place the full window, including the title bar, menu bar, tool bars, and outer edges, use the `OuterPosition` property.

Note: On Windows systems, figure windows cannot be less than 104 pixels wide, regardless of the value of the Position property.

Example: `[230 250 570 510]`

Data Types: `double`

`OuterPosition` — Location and size of figure's outer bounds

`[left bottom width height]`

Location and size of outer bounds, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the left edge of the primary display to the outer left edge of the figure window. This value can be negative on systems that have more than one monitor.
`bottom`	Distance from the bottom edge of the primary display to the outer bottom edge of the figure window. This value can be negative on systems that have more than one monitor.
`width`	Distance between the right and left outer edges of the figure.
`height`	Distance between the top and bottom outer edges of the figure.

All measurements are in units specified by the Units property.

If you attempt to set the OuterPosition property when the figure is docked, this warning displays:

```
Warning: Cannot set OuterPosition while
WindowStyle is 'docked'
```

Note: On Microsoft Windows systems, figure windows are always at least 104 pixels wide, regardless of the value of the OuterPosition property.

Example: `[230 250 570 510]`

Data Types: `double`

`Units` — Units of measurement

`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as `'pixels'`, `'normalized'`, `'inches'`, `'centimeters'`, `'points'`, or `'characters'`.

MATLAB uses these units to interpret the location and size values of the Position property:

- The size of a figure specified in pixel units depends on the system display settings and resolution.
- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- Character units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the Units before the Position property, then MATLAB sets Position using the units you specify.
- If you specify the Units property after the Position property, MATLAB sets the position using the default Units. Then, MATLAB converts the Position value to the equivalent value in units you specify.

`Resize` — Window resize mode

`'on'` (default) | `'off'`

Window resize mode, specified as:

- `'on'` — Users can resize the figure window.
- `'off'` — Users cannot resize the figure window. The figure window does not display any resizing controls.

`SizeChangedFcn` — Window resize callback function

`''` (default) | function handle | cell array | string

Callback function that executes when the figure size changes, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Define this callback function when you want to customize a UI layout beyond what the Position and Units properties provide.

The SizeChangedFcn callback executes under these circumstances:

- The figure becomes visible for the first time.
- The figure is visible while its drawable area changes. The drawable area is the area inside the outer bounds of the figure.
- The figure becomes visible for the first time after its drawable area changes. This situation occurs when the drawable area changes while the figure is invisible, and then it becomes visible later.

These are some of the important characteristics of the SizeChangedFcn callback and some recommended best practices:

- Consider delaying the display of the figure until after all the variables that the figure's SizeChangedFcn uses are defined. This practice can prevent the figure's SizeChangedFcn callback from returning an error. To delay the display of the figure, set its Visible property to `'off'`. Then, set the Visible property to `'on'` after you define the variables that your SizeChangedFcn callback uses.
- Use the `gcbo` function in your SizeChangedFcn code to get the figure object that the user is resizing.

Example: Uicontrol That has Constant Height

This example code shows how to create a UI containing a uicontrol having a constant height at the top of the figure. To see how it works, copy and paste this code into an editor and save it as `sbar.m`.

```
function sbar(src,callbackdata)
   u = findobj(gcbo,'Tag','StatusBar');
   fig = gcbo;
   old_units = fig.Units;
   fig.Units = 'pixels';
   sbar_units = u.Units;
   u.Units = 'pixels';
   figpos = fig.Position;
   upos = [1 figpos(4) - 20 figpos(3) 20];
```

```
   u.Position = upos;
   u.Units = sbar_units;
   fig.Units = old_units;
   u.Visible = 'on';
end
```

Then, run the following code:

```
f = figure('Visible','off');
u = uicontrol('Style','edit', 'Tag','StatusBar');
f.SizeChangedFcn = @sbar;
f.Visible = 'on';
```

The `sbar` function maintains a 20-pixel uicontrol height and sets the uicontrol width equal to the width of the figure. Notice the use of the `findobj` function to retrieve the uicontrol object. This function also calls the `gcbo` function to access the figure object.

Data Types: `function_handle` | `cell` | `char`

`ResizeFcn` — Figure resize callback function

`''` (default) | function handle | cell array | string

Callback function that executes when the figure size changes, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Note: Use of the ResizeFcn property is not recommended. It might be removed in a future release. Use SizeChangedFcn instead.

Data Types: `function_handle` | `cell` | `char`

Multiple Plots

`NextPlot` — Directive on how to add next plot

`'add'` (default) | `'new'` | `'replace'` | `'replacechildren'`

Directive on how to add next plot, specified as `'add'`, `'new'`, `'replace'`, or `'replacechildren'`.

This table describes the effects of each value.

Property Value	Effect
`'new'`	Creates a new figure and uses it as the current figure.
`'add'`	Adds new graphics objects without clearing or resetting the current figure.
`'replacechildren'`	Removes all axes objects who are not hidden before adding new objects. Does not reset figure properties. Equivalent to using the `clf` command.
`'replace'`	Removes all axes objects and resets figure properties to their defaults before adding new graphics objects. Equivalent to using the `clf reset` command.

Consider using the `newplot` function to handle the NextPlot property. For more information, see the axes `NextPlot` property and "Prepare Figures and Axes for Graphs".

Interactive Control

`Selected` — Selection status of figure

`'off'` (default) | `'on'`

Note: The behavior of the Selected property changed in R2014b, and it is not recommended. It no longer has any effect on figures. This property might be removed in a future release.

`SelectionHighlight` — Ability of figure to highlight when user selects it

`'on'` (default) | `'off'`

Note: Use of the figure SelectionHighlight property is not recommended. This property might be removed in a future release.

This property has no effect on figures.

Callback Execution Control

`Interruptible` — Callback interruption

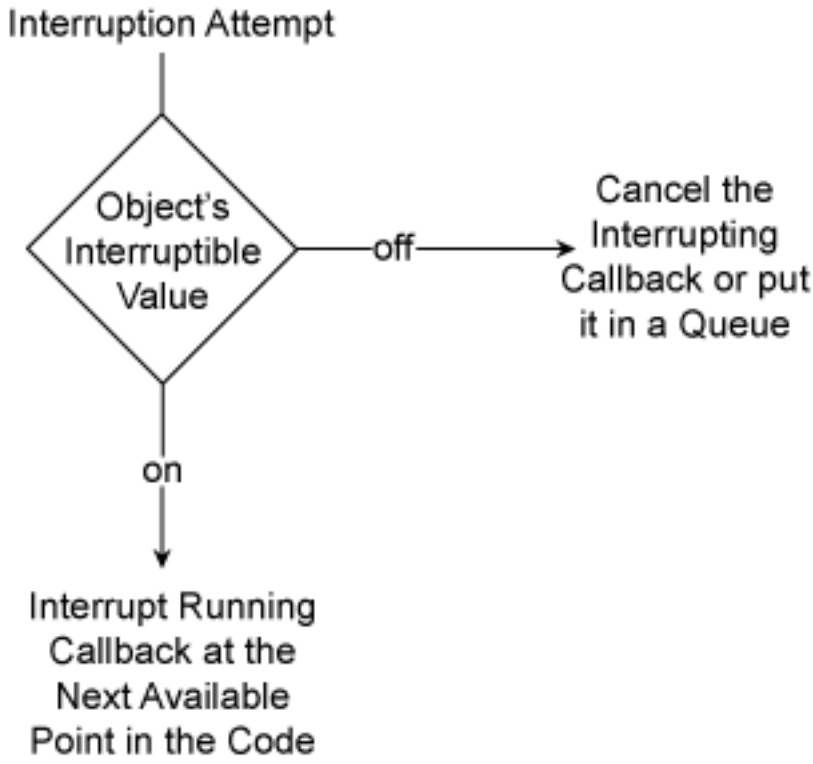
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a figure callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See "Interrupt Callback Execution" for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

Ability to become current object, specified as `'on'` or `'off'`:

- Setting the value to `'on'` allows the figure to become the current object when the user clicks on it. A value of `'on'` also allows the figure CurrentObject property and the `gco` function to report the figure as the current object.
- Setting the value to `'off'` sets the figure CurrentObject property to an empty `GraphicsPlaceholder` array when the user clicks on the figure.

Keyboard Control

`CurrentCharacter` — Last key pressed in figure

`''` (default) | 1-character string

The last key pressed in the figure, returned as a 1-character string. Use the CurrentCharacter property to obtain user input.

Example: `'t'`

`KeyPressFcn` — Key-press callback function

`''` (default) | function handle | cell array | string

The callback function that executes when a user presses a key, specified as one of these values:

- Function handle

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes when the figure window has focus and the user presses a key. If you do not define a function for this property, MATLAB passes key presses to the Command Window. Repeated key presses retain the focus of the figure, and the function executes with each key press. If the user presses multiple keys at approximately the same time, MATLAB detects the key press for the last key pressed.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Contents
`Character`	The character that displays as a result of pressing the key or keys. The character can be empty or unprintable.
`Modifier`	A cell array containing the names of one or more modifier keys that are being pressed (such as , **control**, **alt**, **shift**). On Macintosh computers, the cell array contains `'command'` when the **command** modifier key is pressed.
`Key`	The key being pressed, identified by the (lowercase) label on the key, or a descriptive string.
`Source`	The object that has focus when the user presses the key.
`Eventname`	The action that caused the callback function to execute.

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the CurrentCharacter property of the figure to determine which character the user pressed.

Example: `@myfun`

Example: `{@myfun,x}`

`KeyReleaseFcn` — Key-release callback function

`''` (default) | function handle | cell array | string

Key press callback function, specified as one of these values

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes when the figure object has focus and the user releases a key.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Character`	Character interpretation of the key that was released.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	Current modifier, such as `'control'`, or an empty cell array if there is no modifier	`{1x0 cell}`	`{1x0 cell}`	`{'shift'`	`{'shift'}`
`Key`	The key being released, identified by the (lowercase) label on the key, or a descriptive string.	`'a'`	`'equal'`	`'shift'`	`'a'`

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Source`	The object that has focus when the user presses the key.	figure	figure	figure	figure
`Eventname`	The action that caused the callback function to execute.	`'KeyRel`	`'KeyRel`	`'KeyRele`	`'KeyRelease'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the CurrentCharacter property of the figure to determine which character the user pressed.

Example: `@myfun`

Example: `{@myfun,x}`

`WindowKeyPressFcn` — Key-press callback function for the figure window

`''` (default) | function handle | cell array | string

Key-press callback function for the figure window, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback executes whenever a key press occurs while either the figure window or any of its children has focus.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to

the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Contents
`Character`	The character displayed as a result of pressing the key. The character which can be empty or unprintable
`Modifier`	A cell array containing the names of one or more modifier keys being pressed (such as **control**, **alt**, **shift**). On Macintosh computers, it contains `'command'` when pressing the **command** modifier key.
`Key`	The key being pressed, identified by the (lowercase) label on the key, or a descriptive string.
`Source`	The object that has focus when the user presses the key.
`Eventname`	The action that caused the callback function to execute.

Example: `@myfun`

Example: `{@myfun,x}`

`WindowKeyReleaseFcn` — Key-release callback function for the figure window

`''` (default) | function handle | cell array | string

Key-release callback function for the figure window, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback executes whenever a key release occurs while the figure window or any of its children has focus.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Contents
`Character`	The character displayed as a result of the releasing the key or keys. The character which can be empty or unprintable.
`Modifier`	A cell array containing the names of one or more modifier keys being released (such as **control**, **alt**, **shift**). On Macintosh computers, it contains `'command'` when releasing the **command** modifier key.
`Key`	The key being released, identified by the (lowercase) label on the key, or a descriptive string.
`Source`	The object that has focus when the user releases the key.
`Eventname`	The action that caused the callback function to execute.

Example: `@myfun`

Example: `{@myfun,x}`

Mouse Control

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback executes whenever the user clicks a mouse button while the pointer is in the figure window, but not over a child object such as a uicontrol, uipanel, axes, or axes child.

See the figure's `SelectionType` property to determine whether modifier keys are also pressed.

`CurrentPoint` — Location of last button click in figure

two-element vector: `[x-coordinate, y-coordinate]`

Location of the last button click in this figure, returned as a two-element vector. The CurrentPoint property value is measured from the lower left corner of the figure window, in units determined by the Units property. MATLAB updates this property whenever a user presses the mouse button while the pointer is in the figure window.

If a user selects a point in the figure, and you use the values returned by the CurrentPoint property to plot that point, there can be differences in the position due to round-off errors.

CurrentPoint and Cursor Motion

MATLAB updates CurrentPoint before executing callback functions defined for the figure `WindowButtonMotionFcn` and `WindowButtonUpFcn` properties. This enables you to query CurrentPoint from these callback functions. It behaves like this:

- If you define a callback function for the `WindowButtonMotionFcn` property or the `WindowButtonUpFcn` property, then MATLAB updates the CurrentPoint property only when the user presses the mouse button within the figure window.
- If you define a callback function for the WindowButtonMotionFcn property, then MATLAB updates the CurrentPoint property just before executing the callback. The WindowButtonMotionFcn property executes only within the figure window, unless the user presses the mouse button within the figure and *keeps the mouse button down* while moving the pointer around the screen. In this case, the function executes (and MATLAB updates the CurrentPoint property) anywhere on the screen until the user releases the mouse button.
- If you define a callback function for the WindowButtonUpFcn property, then MATLAB updates the CurrentPoint property just before executing the callback. The WindowButtonUpFcn callback executes only while the pointer is within the figure window, unless the user presses the mouse button within the window and *releases the mouse button* anywhere on the screen. In this case, the function executes, preceded by an update of the CurrentPoint property value.
- If you add a uicontrol or uitable component to the figure, then MATLAB updates the CurrentPoint property when the user right-clicks the component, or when they left-click the component while the Enable property of that component is `'off'` or `'inactive'`.

In some situations (such as when the WindowButtonMotionFcn callback takes a long time to execute and the user moves the pointer very rapidly), the CurrentPoint property might not reflect the actual location of the pointer, but rather the location at the time when the WindowButtonMotionFcn callback began execution.

The root `PointerLocation` property contains the location of the pointer updated synchronously with pointer movement. However, the location is measured with respect to the screen, not a figure window.

`SelectionType` — Mouse selection type

`'normal'` (default) | `'extend'` | `'alt'` | `'open'`

Mouse selection type, returned as `'normal'`, `'extend'` , `'alt'` , or `'open'`. MATLAB maintains this property to provide information about the last mouse-button press that occurred within the figure window. This information indicates the type of selection made. Selection types are actions that MATLAB generally associates with particular responses from the user interface software (for example, single-clicking a graphics object places it in move or resize mode; double-clicking a file name opens it, and so on).

The value of this property depends on the type of mouse click and the user's operating system.

Selection Type	Microsoft Windows	Linux	Mac
`'normal'`	Click left mouse button.	Click left mouse button.	Click left mouse button.
`'extend'`	Either of the following: • **Shift**-click left mouse button. • Click both left and right mouse buttons.	Either of the following: • **Shift**-click left mouse button. • Click middle mouse button.	Any one of the following: • **Shift**-click left mouse button. • Click middle mouse button. • Click both left and right mouse buttons.
`'alt'`	**Control**-click left mouse button or click right mouse button.	**Control**-click left mouse button or click right mouse button.	**Control**-click left mouse button or click right mouse button.
`'open'`	Double-click any mouse button.	Double-click any mouse button.	Double-click any mouse button.

Note: For a list box (uicontrol), the second click of a double-click sets the SelectionType property to `'open'`.

`WindowButtonDownFcn` — Button press callback function

`''` (default) | function handle | cell array | string

Button press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes whenever a user clicks a mouse button while the pointer is in the figure window. See the `WindowButtonMotionFcn` property for an example.

Note: When using a two-button or three-button mouse on Macintosh systems, right-button and middle-button presses are not always reported. This happens *only* when a new figure window appears under the mouse cursor and the user clicks the mouse without first moving the mouse. In this circumstance, for the WindowButtonDownFcn callback to work, the user needs to do one of the following:

- Move the mouse after the figure is created, and then click any mouse button.
- Press **Shift** or **Ctrl** while clicking the left mouse button to perform the Extend and Alternate selection types.

Pressing the left mouse button (or single mouse button) works without having to take either of the above actions.

For information on how callbacks interact, see the Interruptible and BusyAction properties.

Example: `@myfun`

Example: `{@myfun,x}`

`WindowButtonMotionFcn` — Mouse-motion callback function

`''` (default) | function handle | cell array | string

Mouse-motion callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes whenever the user moves the pointer within the figure window.

Note: On some systems, the WindowButtonMotionFcn callback executes when MATLAB creates a figure, even though there has been no mouse motion within the figure.

Your callback functions might need to update the display by calling the `drawnow` or `pause` function, which causes MATLAB to process all callbacks in the queue. Processing the callback queue can cause your callback function to be reentered. Design your code to handle reentrancy and do not depend on global variables that might change state during reentrance.

For information on how callbacks interact, see the Interruptible and BusyAction properties.

Example: Coding Window Button Callback Functions

This example shows how to code all three window button callback functions so that the user can draw lines using mouse motions.

Copy and save the following code to a file on a writable folder on your system. Then, run the code. Click the left mouse button in the axes and move the cursor. Left-click to define the line end point. Right-click to end drawing mode.

```
function window_motion_test
```

```
%
figure('WindowButtonDownFcn',@wbdcb)
ah = axes('SortMethod','childorder');
axis ([1 10 1 10])
title('Click and drag')
   function wbdcb(src,callbackdata)
     if strcmp(src.SelectionType,'normal')
        src.Pointer = 'circle';
        cp = ah.CurrentPoint;
        xinit = cp(1,1);
        yinit = cp(1,2);
        hl = line('XData',xinit,'YData',yinit,...
        'Marker','p','color','b');
        src.WindowButtonMotionFcn = @wbmcb;
        src.WindowButtonUpFcn = @wbucb;
     end

        function wbmcb(src,callbackdata)
           cp = ah.CurrentPoint;
           xdat = [xinit,cp(1,1)];
           ydat = [yinit,cp(1,2)];
           hl.XData = xdat;
           hl.YData = ydat;
           drawnow
        end

        function wbucb(src,callbackdata)
           if strcmp(src.SelectionType,'alt')
              src.Pointer = 'arrow';
              src.WindowButtonMotionFcn = '';
              src.WindowButtonUpFcn = '';
           else
              return
           end
        end
  end
end
```

`WindowButtonUpFcn` — Button-release callback function

`''` (default) | function handle | cell array | string

Button-release callback function, specified as one of these values:

- Function handle

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes whenever a user releases a mouse button.

The button-up callback is associated with the figure window in which a previous button-down action occurred. Therefore, the pointer need not be in the figure window when the user releases the button to generate the button-up callback.

If the callback functions defined by the `WindowButtonDownFcn` or `WindowButtonMotionFcn` properties contain `drawnow` commands or call other functions that contain `drawnow` commands and the `Interruptible` property is set to `'off'`, then the WindowButtonUpFcn callback might not be called. You can prevent this situation by setting the Interruptible property to `'on'`.

Your callback functions might need to update the display by calling the `drawnow` or `pause` function, which causes MATLAB to process all callbacks in the queue. Processing the queue can cause your callback function to be reentered. For example, a `drawnow` command in the WindowButtonUpFcn callback might result in the WindowButtonUpFcn callback being called again before the first call has finished. Design your code to handle reentrancy and do not depend on global variables that might change state during reentrance.

You can use the Interruptible and BusyAction figure properties to control how callbacks interact.

Example: `@myfun`

Example: `{@myfun,x}`

`WindowScrollWheelFcn` — Mouse-scroll-wheel callback

`''` (default) | function handle | cell array | string

Mouse-scroll-wheel callback, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback executes when the user moves the mouse-scroll-wheel while the figure has focus. MATLAB executes the callback with each single mouse-scroll-wheel click.

Be aware that it is possible for another object to capture the mouse-scroll-wheel movement from MATLAB. For example, if the figure contains Java or ActiveX control objects that are listening for mouse-scroll-wheel movements, then these objects can capture the activity and prevent the WindowScrollWheelFcn callback from executing.

WindowScrollWheelFcn Callback Data

When the callback is a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Contents
`VerticalScrollCount`	A positive or negative integer that indicates the number of mouse-scroll-wheel clicks. Positive values indicate clicks of the wheel scrolled in the down direction. Negative values indicate clicks of the wheel scrolled in the up direction.
`VerticalScrollAmount`	The current system setting for the number of lines that are scrolled for each click of the scroll wheel. If the mouse property setting for scrolling is set to `One screen at a time`, the `VerticalScrollAmount` property is `1`.

Effects on Other Properties

The WindowScrollWheelFcn property value has the following effects on these properties:

- `CurrentObject` property — The WindowScrollWheelFcn property has no effect on the CurrentObject property.

- `CurrentPoint` property — If there is no callback defined for the WindowScrollWheelFcn property, then MATLAB does not update the CurrentPoint property as the user turns the scroll wheel. However, if there is a callback defined for the WindowScrollWheelFcn property, then MATLAB updates the CurrentPoint property just before executing the callback. This enables you to determine the point at which the mouse scrolling occurred.
- `SelectionType` property — The WindowScrollWheelFcn property has no effect on the SelectionType property.

Values Returned by VerticalScrollCount

When a user moves the mouse scroll wheel by one click, MATLAB increments the count by +/- 1, depending on the direction of the scroll (scroll down being positive). When MATLAB calls the WindowScrollWheelFcn callback, the counter resets. In most cases, this means that the absolute value of the returned value is 1. However, if the WindowScrollWheelFcn callback takes a long enough time to return or the user spins the scroll wheel very fast, or both, then the returned value can have an absolute value greater than one.

The actual value returned by `VerticalScrollCount` property is the algebraic sum of all mouse-scroll-wheel clicks that occurred since last processed. This enables your callback to respond correctly to the user action.

Example: Code WindowScrollWheelFcn Callback

This example creates a graph and enables users to use the mouse scroll wheel to change the range over which MATLAB evaluates a mathematical function. In addition, it updates the graph to reflect the new limits as the user turns the scroll wheel.

Copy and save the function to a writable folder on your system. Then, run the code. Mouse over the figure and scroll your mouse wheel.

```
function scroll_wheel
% Illustrates how to use WindowScrollWheelFcn property
%
   f = figure('WindowScrollWheelFcn',@figScroll,'Name','Scroll Wheel Demo');
   x = [0:.1:40];
   y = 4.*cos(x)./(x+2);
   a = axes;
   h = plot(x,y);
```

```
   title('Rotate the scroll wheel')
   function figScroll(src,callbackdata)
      if callbackdata.VerticalScrollCount > 0
         xd = h.XData;
         inc = xd(end)/20;
         x = [0:.1:xd(end)+inc];
         re_eval(x)
      elseif callbackdata.VerticalScrollCount < 0
         xd = h.XData;
         inc = xd(end)/20;
         x = [0:.1:xd(end)-inc+.1]; % Don't let xd = 0;
         re_eval(x)
      end
   end % figScroll

   function re_eval(x)
      y = 4.*cos(x)./(x+2);
      h.YData = y;
      h.XData = x;
      a.XLim = [0 x(end)];
      drawnow
   end % re_eval
end % scroll_wheel
```

`UIContextMenu` — Figure context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Figure context menu, specified as a uicontextmenu object. Use this property to display a context menu when the user right-clicks on the figure. Create the context menu using the `uicontextmenu` function.

Window Control

`CloseRequestFcn` — Figure close request callback function

`''` (default) | function handle | cell array | string

Figure close request callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback executes whenever a user attempts to close a figure window. You can, for example, display a dialog box to ask a user to confirm or cancel the close operation or to prevent users from closing a figure that contains a UI.

The basic mechanism is:

1. A user issues the `close` or `close all` command from the command line, closes the figure from the computer window manager menu, or closes the figure by quitting MATLAB.
2. The close operation executes the function defined by the figure CloseRequestFcn property. The default function is `closereq`.

   The `closereq` function unconditionally deletes the current figure, destroying the window. The `closereq` function takes advantage of the fact that the `close` command makes each figure specified as an argument the current figure before calling its respective close request function.

The `closereq` function honors the ShowHiddenHandles property setting during figure deletion and does not delete hidden figures.

Unless the close request function calls the `delete` or `close` function, MATLAB never closes the figure. (You can call `delete(f)` from the command line if you have created a window with a nondestructive close request function.)

Example: Code CloseRequestFcn to Display Dialog Box

This example shows how to code the close request function to display a question dialog box asking the user to confirm the close operation. Save the code to a writable folder on your system.

```
function my_closereq(src,callbackdata)
% Close request function
% to display a question dialog box
   selection = questdlg('Close This Figure?',...
      'Close Request Function',...
      'Yes','No','Yes');
   switch selection,
      case 'Yes',
         delete(gcf)
      case 'No'
      return
```

```
    end
end
```

Now, create a figure specifying `my_closereq` for the CloseRequestFcn:

```
figure('CloseRequestFcn',@my_closereq)
```

Close the figure window and the question dialog box displays.

`WindowStyle` — Figure window behavior

`'normal'` (default) | `'modal'` | `'docked'`

Figure window behavior, specified as one of the following:

- `'normal'` — The figure window is independent of other windows, and the other windows are accessible while the figure is displaying.
- `'modal'` — The figure displays on top of all existing figure windows, making them inaccessible as long as the top figure exists and remains modal. However, any new figures created after a modal figure will display.

  When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to a normal window style, or is deleted. At that time, focus reverts to the window that last had focus.
- `'docked'` — The figure displays in the desktop or a document window. This code docks the figure and sets the `DockControls` property to `'on'`, if it was set to `'off'`.

  ```
  f = figure;
  f.WindowStyle = 'docked';
  ```

  If the WindowStyle property is set to `'docked'`, you cannot set the DockControls property to `'off'`.

Note: These are some important characteristics of the WindowStyle property and some recommended best practices:

- When you create UI windows, always specify the WindowStyle property. If you also want to set the Resize, Position, or OuterPosition properties of the figure, then set the WindowStyle property first.
- You can change the WindowStyle property of a figure at any time, including when the figure is visible and contains children. However on some systems, setting this

property might cause the figure to flash or disappear and reappear, depending on the system's implementation of normal and modal windows. For best visual results, set the WindowStyle property at creation time or when the figure is invisible.

- Calling `reset` on a figure does not change the value of the WindowStyle property.

Modal Window Style Behavior

When WindowStyle is set to `'modal'`, the figure window traps all keyboard and mouse actions over all MATLAB windows as long as the windows are visible. Windows belonging to applications other than MATLAB are unaffected.

Typing **Ctrl+C** when a modal figure has focus causes that figure to revert to a `'normal'` WindowStyle property setting. This allows the user to type at the command line.

Figures with the WindowStyle property set to `'modal'` and the Visible property set to `'off'` do not behave modally until MATLAB makes them visible. Therefore, you can hide a modal window for later reuse, instead of destroying it.

Modal figures do not display uimenu children, built-in menus, or toolbars. But, it is not an error to create uimenus in a modal figure or to change the WindowStyle property setting to `'modal'` on a figure with uimenu children. The uimenu objects exist and the figure retains them. If you reset the figure's WindowStyle property to `'normal'`, the uimenus display.

Creation and Deletion Control

`BeingDeleted` — Deletion status of figure

`'off'` (default) | `'on'`

Deletion status of figure, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the figure begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the figure no longer exists.

Check the value of the BeingDeleted property to verify that the figure is not about to be deleted before querying or modifying it.

`CreateFcn` — Figure creation function

function handle | cell array | string

Figure creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the figure. MATLAB initializes all figure property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the figure that is being created.

Setting the CreateFcn property on an existing figure has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the figure object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Figure deletion function

function handle | cell array | string

Figure deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the figure (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the figure. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the figure that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`CurrentAxes` — Target axes in current figure

`axes object`

Target axes in the current figure, specified as an axes object. MATLAB sets this property to the figure's current axes object. In all figures for which axes children exist, there is always a current axes. The current axes does not have to be the topmost axes, and setting an axes to be the current axes does not restack it above all other axes. If a figure contains no axes, the `get(gcf,'CurrentAxes')` command returns an empty array.

To get the current axes object without forcing the creation of an axes if one does not exist, query the figure `CurrentAxes` property.

```
fh = gcf;
ah = fh.CurrentAxes
```

MATLAB returns `h` as an empty array if there is no current axes. Be aware that `gcf` creates a figure if one does not exist.

`CurrentObject` — Most recently selected component in figure

object

Most recently selected component in the Figure, specified as an object. MATLAB sets the CurrentObject property to the last object the user clicked. This object is the front-

most object in the view. You can use this property to determine which object a user has selected.

An object's HitTest property controls whether that object can become the CurrentObject.

Clicking an object whose HandleVisibility property is `off` (such as axis labels and title) causes the CurrentObject property to be set to empty. To avoid returning an empty value when users click a hidden object, set HitTest property of the hidden object to `'off'`.

Moving the cursor over objects does not update the CurrentObject. Users must click objects to update this property. See the `CurrentPoint` property for related information.

If you are looking for a quick way to access the current object, consider using the `gco` command.

`FileName` — FIG-file name

string

FIG-file name, specified as a string. GUIDE uses this property to store the name of the UI layout file that it saves. If you are not working in GUIDE, then the FileName property is empty.

Example: `'myguifile.fig'`

`IntegerHandle` — Ability to assign figure number

`'on'` (default) | `'off'`

Ability to assign figure number, specified as `'on'` or `'off'`.

If you set the IntegerHandle property to `'on'`, MATLAB finds the lowest integer value not used by an existing figure and sets the Number property to that value. If you delete a figure, MATLAB can reuse the number on a new figure window.

If you set the IntegerHandle property to `'off'`, MATLAB does not assign an integer value to the figure and sets the Number property to an empty array (`[]`).

`Number` — Figure number

integer | `[]`

Figure number, returned as an integer or empty array. You can refer to a figure using this value. For example, `figure(2)` makes the figure with a Number property value of 2 the current figure.

If the `IntegerHandle` property is set to `'off'`, Number is an empty array.

If IntegerHandle is `'on'`, the Number property value is an integer value. If a figure is deleted, MATLAB reuses that figure's number for the next figure window created.

Example: `3`

`Type` — Type of Figure object

`'figure'`

Type of Figure object, returned as the string, `'figure'`. Use this property to find all objects of a given type within a plotting hierarchy.

`Tag` — Figure identifier

`' '` (default) | string

Figure identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the figure. When you need access to the figure elsewhere in your code, you can use the `findobj` function to search for the figure based on the Tag value.

Example: `'Plotting Figure'`

`UserData` — Data to associate with the figure object

empty array (default) | array

Data to associate with the figure object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See "Share Data Among Callbacks" for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Figure parent

root object

Figure parent, returned as a root object.

`Children` — Children of figure

empty `GraphicsPlaceholder` array (default) | 1-D array of objects

Children of the figure, returned as an empty `GraphicsPlaceholder` or a 1-D array of objects.

You cannot add or remove children using the `Children` property of the figure. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the front-to-back order (stacking order) of the components on the screen.

To add a child to this list, set the `Parent` property of the child component to be the figure object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of figure handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of figure handle, specified as `'on'`, `'callback'`, or `'off'`.

This property determines whether a figure is in its parent's (the root's) list of children. HandleVisibility is useful for preventing command-line users from accidentally drawing into, or deleting a figure that contains only user interface components (such as a dialog box).

If an object is not in its parent's list of children, functions that find objects by searching the object hierarchy or querying properties cannot return that object. Such functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When the HandleVisibility property value is restricted using the `'callback'` or `'off'` settings, the object does not appear in the parent object Children property, figures do not appear in the root `CurrentFigure` property, objects do not appear in the root `CallbackObject` property or in the figure `CurrentObject` property, and axes do not appear in their parent CurrentAxes property.

Set the root `ShowHiddenHandles` property to `'on'` to make all objects visible, regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties).

Pointers

`Pointer` — Pointer symbol

`'arrow'` (default) | string

Pointer symbol, specified as a string. Valid strings are listed in the following table. By convention, each symbol commonly indicates a particular usage. However, MATLAB does not enforce rules for the use of any symbols. The appearance of the symbol that displays for a given string is operating-system dependent.

Typical Usage Indicated	Symbol String
Editing location in text	`'ibeam'`
Point on a graphics object	`'crosshair'`
Point anywhere in the figure	`'arrow'`
Busy system	`'watch'`
Resizing an object from a top-left or bottom-right corner	`'topl'` or `'botr'`
Resizing an object from a top-right or bottom-left corner	`'topr'` or `'botl'`
A hot spot	`'circle'`
A point	`'cross'`
Moving an object	`'fleur'`
Resizing an object from the left side	`'left'`
Resizing an object from the right side	`'right'`
Resizing an object from the top or bottom	`'top'` or `'bottom'`
Clickable icon	`'hand'`

Note: The `'fullcrosshair'` option was removed in R2014b.

Setting Pointer property to `'custom'` enables you to define your own pointer symbol. See the `PointerShapeCData` property for more information.

`PointerShapeCData` — Pointer shape

`16-by-16 matrix`

Pointer shape MATLAB uses when the Pointer property is set to `'custom'`, specified as a `16-by-16 matrix`. The matrix defines a 16-by-16 pixel pointer using the following values:

- `1` — Color pixel black.
- `2` — Color pixel white.
- `NaN` — Make pixel transparent, such that underlying screen shows through

Element `(1,1)` of the PointerShapeCData matrix corresponds to the upper-left corner of the pointer. Setting the `Pointer` property to one of the predefined pointer symbols does not change the value of the PointerShapeCData. Computer systems supporting `32`-by-`32` pixel pointers fill only one quarter of the available pixmap.

`PointerShapeHotSpot` — Active area of pointer

`[1 1]` — upper-left corner (default) | `two-element vector`

Active area of pointer, specified as a two-element vector. The vector specifies the row and column indices in the `PointerShapeCData` matrix defining the pixel indicating the pointer location. The `CurrentPoint` property and the root object's `PointerLocation` property specify the location.

Printing

`PaperSize` — Size of the current `PaperType`

[width height]

Size of the current PaperType, specified as `[width height]`. The default value is locale-dependent, as follows:

- United States — [8.5000 11]
- Europe and Asia — [21 29.7]

The `PaperUnits` property specifies the units of measurement. You cannot set this property if the PaperUnits property is set to `'normalized'`. Attempting to do so results in an error.

See the `PaperType` property to select a standard paper size.

Tip Use the PaperType property instead of the PaperSize property if you do not want to be concerned with setting the PaperUnits property.

`PaperUnits` — Output measurement units

`'inches' | 'centimeters' | 'normalized' | 'points'`

Output measurement units measured from the bottom of the page, specified as one of the following values:

- `'inches'`— absolute unit

  This is the default when the locale is the United States.
- `'normalized'` — maps the lower left corner of the page to (0, 0) and the upper right corner to (1.0, 1.0)
- `'centimeters'`— absolute unit

  This is the default when the locale is Europe or Asia.
- `'points'` — absolute unit. One point equals 1/72 of an inch

This property affects all printed and exported output, not just output printed to paper.

MATLAB uses these units with the `PaperPosition` and `PaperSize` properties.

If you change the value of the PaperUnits property, it is good practice to return the property to its default value after completing your computation so as not to affect other functions that assume the PaperUnits property is set to the default value.

`PaperOrientation` — Figure orientation on printed page

`'portrait'` (default) | `'landscape'`

Figure orientation on printed page, specified as `'portrait'` or `'landscape'`.

- `'portrait'` — Orients the longest page dimension vertically.
- `'landscape'` — Orients the longest page dimension horizontally.

See the `orient` function for more information.

`PaperPosition` — Figure location and size on page

`[left bottom width height]`

Figure location and size on the page, specified as `[left bottom width height]`. Page refers to printed pages and files exported to PDF or PostScript® formats. The `left` value specifies the distance from the left side of the page to the left side of the figure. The `bottom` value specifies the distance from the bottom of the page to the bottom of the figure. Together these distances specify the lower left corner of the figure position.

The `width` and `height` values define the dimensions of the figure. When exporting to a nonpage format, such as to image file formats or encapsulated PostScript (EPS), MATLAB uses the width and height values only.

Note: If the width and height values are too large, then the figure might not reach the specified size. If the figure does not reach the specified size, then UI components on the figure, such as uicontrols or a uitable, might not save or print as expected.

The `PaperUnits` property specifies the units of measurement.

Example: `[.25 .25 8 6]`

`PaperPositionMode` — Directive for honoring `PaperPosition` property

`'manual'` (default) | `'auto'`

Directive for honoring the `PaperPosition` property, specified as one of the following:

- `'manual'` — MATLAB honors the value specified by the PaperPosition property.
- `'auto'` — MATLAB prints the figure the same size as it appears on the computer screen. For page formats, the output is centered on the page.

  To determine the output size when `PaperPosition` is `'auto'`, the figure's size (width and height) is converted to inches based on the screen resolution. MATLAB generates output to match that size, based on the output resolution. For example, a figure that is 500 x 400 pixels on a screen that has a resolution of 100 dpi measures 5x4 inches; when output at 150 dpi, the image will be 750 x 600 pixels (5 inches * 150 dpi x 4 inches * 150 dpi)

  Tip To generate output that matches the on-screen size in pixels, call `print` with the `-r0` option.

`PaperType` — Standard paper size selection

string

Standard paper size selection, specified as a string.

The default setting is locale-dependent, as follows:

- United States — `'usletter'`

- Europe and Asia — `'a4'`

Valid strings are described in the table that follows. You can enter strings using uppercase or lowercase letters, but MATLAB always stores the strings using lowercase.

To position the printed figure on a new paper size, you might need to change the setting of the `PaperPosition` property. Consider setting the `PaperUnits` property to `'normalized'`. This setting enables MATLAB to automatically size the figure to occupy the same relative amount of the printed page, regardless of the paper size.

Property Value	Size (Width x Height)
`'usletter'`	8.5-by-11 inches
`'uslegal'`	8.5-by-14 inches
`'tabloid'`	11-by-17 inches
`'a0'`	841-by-1189 mm
`'a1'`	594-by-841 mm
`'a2'`	420-by-594 mm
`'a3'`	297-by-420 mm
`'a4'`	210-by-297 mm
`'a5'`	148-by-210 mm
`'b0'`	1029-by-1456 mm
`'b1`	728-by-1028 mm
`'b2'`	514-by-728 mm
`'b3'`	364-by-514 mm
`'b4'`	257-by-364 mm
`'b5'`	182-by-257 mm
`'arch-a'`	9-by-12 inches
`'arch-b'`	12-by-18 inches
`'arch-c'`	18-by-24 inches
`'arch-d'`	24-by-36 inches
`'arch-e'`	36-by-48 inches
`'a'`	8.5-by-11 inches

Property Value	Size (Width x Height)
'b'	11-by-17 inches
'c'	17-by-22 inches
'd'	22-by-34 inches
'e'	34-by-43 inches

InvertHardcopy — Directive to use black on white background when printing or exporting

'on' (default) | 'off'

Directive to use black on white background when printing or exporting, specified as the string 'on' or 'off'. Printing a figure that has a background color (Color property) that is not white results in poor contrast between graphics objects and the figure background; and also consumes a lot of printer toner.

If you set the InvertHardCopy property to 'on', MATLAB changes the color of the figure and axes to white and the axis lines, tick marks, axis labels, etc., that are currently white to black. The lines, text, and edges of patches and surfaces might be changed, depending on the print command options you specify.

If you set InvertHardCopy to 'off', the printed or exported output matches the colors displayed on the screen.

See Also

figure | gcf

More About

- "Access Property Values"
- "Default Property Values"

figurepalette

Show or hide **Figure Palette**

Syntax

```
figurepalette('show')
figurepalette('hide')
figurepalette
figurepalette(figure_handle,...)
```

Description

`figurepalette('show')` displays the palette on the current figure.

`figurepalette('hide')` hides the palette on the current figure.

`figurepalette` toggles the visibility of the palette on the current figure. You can also use `figurepalette('toggle')` instead for the same functionality.

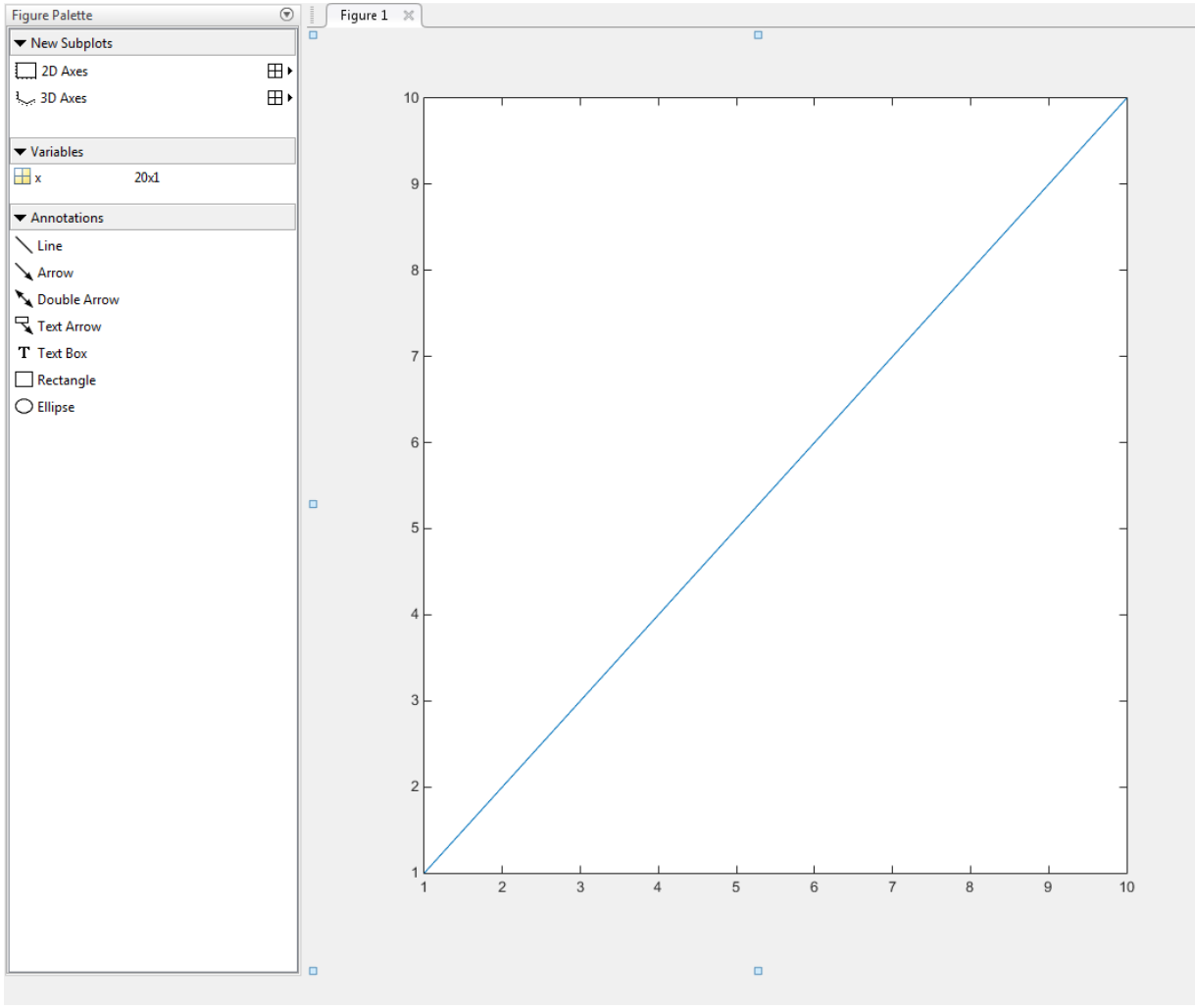
`figurepalette(figure_handle,...)` shows or hides the palette on the figure specified by `figure_handle`.

Examples

Open Figure Palette

Plot a line and open the figure palette.

```
plot(1:10)
figurepalette('show')
```



Alternatives

To collectively enable Plotting Tools, use the large Plotting Tool icon  on the figure toolbar. To collectively disable the Plotting Tools, use the smaller icon  . Open or close the **Figure Palette** tool from the figure's **View** menu.

More About

Tips

If you call `figurepalette` in a MATLAB program and subsequent lines depend on the Figure Palette being fully initialized, follow it by `drawnow` to ensure complete initialization.

See Also

`plottools` | `plotbrowser` | `propertyeditor`

Introduced before R2006a

fileattrib

Set or get attributes of file or folder

Syntax

```
fileattrib
fileattrib(name)

fileattrib(name,attribs)
fileattrib(name,attribs,users)
fileattrib(name,attribs,users,'s')

[status,message,messageid] = fileattrib(name,attribs, ___ )
[status,message] = fileattrib(name)
```

Description

`fileattrib` gets attribute values for the current folder, using the following structure, where `Name` is always a string containing the current folder name. For the other fields, a value of `0` indicates that the attribute is off, `1` indicates that the attribute is on, and `NaN` indicates that the attribute does not apply:

```
Name
archive
system
hidden
directory
UserRead
UserWrite
UserExecute
GroupRead
GroupWrite
GroupExecute
OtherRead
OtherWrite
OtherExecute
```

`fileattrib(name)` gets the attribute values for the named file or folder.

`fileattrib(name,attribs)` sets the specified attributes for the named file or folder.

`fileattrib(name,attribs,users)` sets the file or folder attributes for the specified subset of users.

`fileattrib(name,attribs,users,'s')` sets the specified attributes for the specified users for the contents of the named folder.

`[status,message,messageid] = fileattrib(name,attribs, ___ )` sets the specified file attributes and gets the function status:

- If status is `0`, then message is the error message, and messageid is the error message identifier.
- If status is `1`, then message is a structure containing the attributes of the named file or folder, and messageid is an empty string.

`[status,message] = fileattrib(name)` gets status and the last *successfully* set attribute structure values for the named file or folder and returns the structure to message. (status is always `1`.)

Examples

View Current Folder Attributes

View attributes of the current folder, assuming the current folder is `C:\my_MATLAB_files`.

```
fileattrib
```

```
ans =

           Name: 'C:\my_MATLAB_files'
        archive: 0
         system: 0
         hidden: 0
      directory: 1
       UserRead: 1
      UserWrite: 1
    UserExecute: 1
      GroupRead: NaN
     GroupWrite: NaN
   GroupExecute: NaN
```

```
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

The attributes indicate that you have read, write, and execute permissions for the current folder.

View File Attributes

View attributes of file `collatz.m`.

```
fileattrib('collatz.m')
```

```
ans =

           Name: 'C:\my_MATLAB_files\collatz.m'
        archive: 1
         system: 0
         hidden: 0
      directory: 0
       UserRead: 1
      UserWrite: 0
    UserExecute: 1
      GroupRead: NaN
     GroupWrite: NaN
   GroupExecute: NaN
      OtherRead: NaN
     OtherWrite: NaN
   OtherExecute: NaN
```

The attributes indicate that the specified item is a file. You can read and execute the file, but cannot update it. The file is archived.

View Folder Attributes on a Windows System

View attributes for the folder `C:\my_MATLAB_files\doc`.

```
fileattrib('C:\my_MATLAB_files\doc')
```

```
ans =

           Name: 'C:\my_MATLAB_files\doc'
        archive: 0
         system: 0
         hidden: 0
```

```
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: NaN
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

The attributes indicate that you have read, write, and execute permissions for the specified folder.

View Folder Attributes on a UNIX System

View attributes for the folder `/public` on a UNIX system.

```
fileattrib('/public')

ans =

           Name: '/public'
        archive: NaN
         system: NaN
         hidden: NaN
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: 1
      GroupWrite: 0
    GroupExecute: 1
       OtherRead: 1
      OtherWrite: 0
    OtherExecute: 1
```

The attributes indicate that you have read, write, and execute permissions for the specfied folder. In addition, users in your UNIX group and all others have read and execute permissions for the specified folder, but not write permissions.

Set File Attributes

Make `myfile.m` writeable.

```
fileattrib('myfile.m','+w')
```

Set File Attributes for Specific Users on UNIX

Make the folder `/home/work/results` a read-only folder for *all users* on UNIX platforms.

```
fileattrib('/home/work/results','-w','a')
```

The minus (`-`) preceding the write attribute, `w`, removes the write status.

Set Attributes for Folder and Its Contents

On Windows platforms, make the folder `D:\work\results` and all its contents read only and hidden.

```
fileattrib('D:\work\results','+h -w','','s')
```

Because a value for the users argument is not applicable on Windows systems, the argument is an empty string. The `s` argument applies the hidden and read-only attributes to the contents of the folder.

Get Attributes Structure for a Folder

Get the attributes for the folder `results` and return them to a structure:

```
[stat,struc] = fileattrib('results')
```

```
stat =
     1

struc =
           Name: 'D:\work\results'
        archive: 0
         system: 0
         hidden: 0
      directory: 1
       UserRead: 1
      UserWrite: 1
    UserExecute: 1
      GroupRead: NaN
     GroupWrite: NaN
   GroupExecute: NaN
      OtherRead: NaN
```

```
      OtherWrite: NaN
    OtherExecute: NaN
```

The operation is successful as indicated by the status, `stat`, value of 1. The structure, `struc`, contains the file attributes.

Access the name attribute value in the structure. MATLAB returns the path for `results`.

```
struc.Name
```

```
ans =
D:\work\results
```

Get Attributes Structure for Multiple Files

Get the attributes for all files in the current folder with names that begin with `new`.

```
[stat,struc] = fileattrib('new*')
```

```
stat =
     1

mess =
1x3 struct array with fields:
    Name
    archive
    system
    hidden
    directory
    UserRead
    UserWrite
    UserExecute
    GroupRead
    GroupWrite
    GroupExecute
    OtherRead
    OtherWrite
    OtherExecute
```

The results indicate there are three matching files.

View the file names.

```
struc.Name
```

```
ans =
D:\work\results\newname.m

ans =
D:\work\results\newone.m

ans =
D:\work\results\newtest.m
```

View just the second file name.

```
struct(2).Name

ans =
D:\work\results\newname.m
```

Successfully Set Attributes for a File and Get Messages

Show output that results when an attempt to set file attributes is successful.

```
[status,message,messageid] = fileattrib('C:/my_MATLAB_files\doc',...
'+h -w','','s')

status =

     1


message =

     ''



messageid =

     ''
```

The `status` value of `1` indicates the set operation was successful; therefore, no error `message` or `messageid` is returned.

Unsuccessfully Set Attributes for a File and Get Messages

Show output that results when an attempt to set file attributes is unsuccessful.

```
[status,message,messageid] = fileattrib('C:/my_MATLAB_files\doc',...
'+h w-','','s')

status =

     0


message =

Illegal file mode characters on the current platform.


messageid =

MATLAB:FILEATTRIB:ModeSyntaxError
```

The `status` value of `0` indicates the set operation was unsuccessful. The minus sign incorrectly appears after `w`, instead of before it.

Input Arguments

`name` — File or folder name

string

The absolute or relative path for a folder or file, specified as a string. To specify all names beginning with certain characters, add the wildcard character, `*`.

Example: `fileattrib('myfile.m')`

`attribs` — File or folder attribute values

`'a' | 'h' | 's' | 'w' | 'x'`

File or folder attribute values, specified as a space delimited string. Use the plus (+) qualifier before an attribute to set it, and the minus (-) qualifier before an attribute to clear it.

attribvalue	Description
`'a'`	Archive (Microsoft Windows platform only).
`'h'`	Hidden file (Windows platform only).

attribvalue	Description
`'s'`	System file (Windows platform only).
`'w'`	Write access (Windows and UNIX platforms). Results differ by platform and application. For example, even though `fileattrib` disables the "write" privilege for a folder, making it read only, files in the folder could be writable for some platforms or applications.
`'x'`	Executable (UNIX platform only).

Example: `fileattrib('myfile.m', '+w -h')`

`users` — Subset of users

`'a' | 'g' | 'o' | 'u' | ''`

Subset of users (on UNIX platforms only), specified as a string. For all platforms other than UNIX, specify the users argument as an empty string, `''`. This value is not returned by `fileattrib` get operations.

Value for UNIX Users	Description
`'a'`	All users on UNIX platforms
`'g'`	Group of users
`'o'`	All other users
`'u'`	Current user

Example: `fileattrib('D:/work/results','-w','a')`

Output Arguments

`status` — Indication of whether attempt to set attribute was successful

`0 | 1`

If attempt to set attribute was unsuccessful, `status` is `0`. Otherwise, `status` is `1`.

`message` — Attribute structure or error message

string | structure array

Attribute structure or error message, depending on whether you are setting or getting attributes and `status`.

Getting or Setting Attributes	Status	Message contents
Setting	0	Error message
Setting	1	Empty string
Getting	1	Structure containing file attributes and values

When you are getting file attributes, the structure contains these fields and possible values.

Field name	Possible Values
`Name`	String containing name of file or folder
`archive`	`0` (not set), `1 (set)`, or `NaN (not applicable)`
`system`	`0`, `1`, or `NaN`
`hidden`	`0`, `1`, or `NaN`
`directory`	`0`, `1`, or `NaN`
`UserRead`	`0`, `1`, or `NaN`
`UserWrite`	`0`, `1`, or `NaN`
`UserExecute`	`0`, `1`, or `NaN`
`GroupRead`	`0`, `1`, or `NaN`
`GroupWrite`	`0`, `1`, or `NaN`
`GroupExecute`	`0`, `1`, or `NaN`
`OtherRead`	`0`, `1`, or `NaN`
`OtherWrite`	`0`, `1`, or `NaN`
`OtherExecute`	`0`, `1`, or `NaN`

`messageid` — Error message identifier

string

Error message identifier when the attempt to set attribute is unsuccessful (status is 0), returned as a string. If status is 1, then messageid is an empty string.

More About

Tips

- `fileattrib` is like the DOS `attrib` command, or the UNIX `chmod` command. [1]

See Also

`cd` | `copyfile` | `delete` | `dir` | `ls` | `mkdir` | `movefile` | `rmdir`

Introduced before R2006a

1. UNIX is a registered trademark of The Open Group in the United States and other countries.

filebrowser

Open Current Folder browser, or select it if already open

Syntax

```
filebrowser
```

Description

`filebrowser` opens the Current Folder browser, or if it is already open, makes it the selected tool.

More About

- "Manage Files and Folders"

See Also

`cd` | `copyfile` | `ls` | `mkdir` | `fileattrib` | `movefile` | `pwd` | `rmdir`

Introduced before R2006a

filemarker

Character to separate file name and internal function name

Syntax

```
M = filemarker
```

Description

`M = filemarker` returns the character that separates a file and a within-file function name.

Examples

On the Microsoft Windows platform, for example, `filemarker` returns the '>' character:

```
filemarker

ans =
   >
```

You can use the following command on any platform to get the help text for the local function `validateSizes` defined in `imwrite.m`:

```
 helptext = help(['imwrite' filemarker 'validateSizes'])

helptext =
  How many bytes does each element occupy in memory?
```

You can use the `filemarker` character to indicate a location within a MATLAB program file where you want to set a breakpoint, for example. On all platforms, if you need to distinguish between two nested functions with the same name, use the forward slash (/) character to indicate the path to a particular instance of a function.

For instance, suppose `myfile.m` contains the following code:

```
function x = A(p1, p2)
```

```
...
  function y = B(p3)
    ...
  end
  function m = C(p4)
    ...
    end
end

function z = C(p5)
...
  function y = D(p6)
    ...
  end
end
```

To indicate that you want to set a breakpoint at `function y` nested within `function x`, use the following command on the Windows platform:

```
dbstop myfile>x/y
```

To indicate that you want to set a breakpoint at `function m` nested within `function x` use the following command on the Windows platform:

```
dbstop myfile>m
```

In the first case, you specify `x/y` because `myfile.m` contains two nested functions named `y`. In the second case, there is no need to specify `x/m` because there is only one function `m` within `myfile.m`.

See Also

`filesep`

fileparts

Parts of file name and path

Syntax

```
[pathstr,name,ext] = fileparts(filename)
```

Description

`[pathstr,name,ext] = fileparts(filename)` returns the path name, file name, and extension for the specified file. The file does not have to exist. `filename` is a string enclosed in single quotes. The returned `ext` field contains a dot (`.`) before the file extension.

Input Arguments

`filename`

String containing a name of a file or folder, which can include a path and file extension. The function interprets all characters following the right-most delimiter as a file name plus extension.

Output Arguments

`pathstr`

String containing the part of `filename` interpreted as a path name

`name`

String containing the name of the file without any extension

`ext`

String containing the file extension only, beginning with a period (`.`)

Examples

Return the pieces of a file specification string to the separate string outputs `pathstr`, `name`, and `ext`. The full file specification is:

```
file = 'H:\user4\matlab\myfile.txt';
[pathstr,name,ext] = fileparts(file)

pathstr =
H:\user4\matlab

name =
myfile

ext =
.txt
```

Query parts of a user `.cshrc` file:

```
[pathstr,name,ext] = fileparts('/home/jsmith/.cshrc')

pathstr =
/home/jsmith

name =
   Empty string: 1-by-0

ext =
.cshrc
```

`fileparts` interprets the entire file name as an extension because it begins with a period.

Alternatives

Use `uigetfile` to interactively select and return a file name and path, or `uigetdir` to interactively select and return a path name. If you call `fileparts` with the output of `uigetfile`, you can parse out the file name and extension.

More About

Path Name

The full or partial path to a destination folder location, always the initial portion of the filename string. Path names end with a slash character and, where appropriate, can begin with a drive letter. Windows paths use backward slashes (\). UNIX and Macintosh paths use forward slashes (/).

Tips

- `fileparts` only parses file names. It does not verify that a file or a folder exists.
- You can reconstruct the file from the parts using:

  ```
  fullfile(pathstr,[name ext])
  ```

- On Microsoft Windows systems, you can use either forward (/) or back (\) slashes as path delimiters, even within the same string. On UNIX and Macintosh systems, use only / as a delimiter. You can use the `filesep` function to insert the correct separator character for the platform on which your code executes:

  ```
  sep = filesep;
  file = ['H:' sep 'user4' sep 'matlab' sep 'myfile.txt'];

  file =
  H:\user4\matlab\myfile.txt
  ```

- If the input consists of a folder name only, be sure that the right-most character is a delimiter (/ or \). Otherwise, `fileparts` parses the trailing portion of `filename` as the name of a file and returns it in `name` instead of in `pathstr`.

See Also

`filesep` | `fullfile` | `pathsep` | `uigetdir` | `uigetfile`

Introduced before R2006a

fileread

Read contents of file into string

Syntax

```
text = fileread(filename)
```

Description

`text = fileread(filename)` returns the contents of the file *`filename`* as a MATLAB string.

Examples

Read and search the file `Contents.m` in the MATLAB `iofun` directory for the reference to `fileread`:

```
% find the correct directory and file
io_contents = ...
  fullfile(matlabroot,'toolbox','matlab','iofun','Contents.m');

% read the file
filetext = fileread(io_contents);

% search for the line of code that includes 'fileread'
% each line is separated by a newline ('\n')

expr = '[^\n]*fileread[^\n]*';
fileread_info = regexp(filetext,expr,'match');
```

See Also

`fgetl` | `fgets` | `fscanf` | `fread` | `importdata` | `type` | `textscan`

filesep

File separator for current platform

Syntax

```
f = filesep
```

Description

`f = filesep` returns the platform-specific file separator character. The file separator is the character that separates individual folder and file names in a path string.

Examples

Create a path to the `iofun` folder on a Microsoft Windows platform:

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']

iofun_dir =
   toolbox\matlab\iofun
```

Create a path to `iofun` on a UNIX[2] platform.

```
iodir = ['toolbox' filesep 'matlab' filesep 'iofun']

iodir =
   toolbox/matlab/iofun
```

See Also

`fullfile` | `fileparts` | `pathsep`

Introduced before R2006a

2. UNIX is a registered trademark of The Open Group in the United States and other countries.

fill

Filled 2-D polygons

Syntax

```
fill(X,Y,C)
fill(X,Y,ColorSpec)
fill(X1,Y1,C1,X2,Y2,C2,...)
fill(...,'PropertyName',PropertyValue)
h = fill(...)
```

Description

The `fill` function creates colored polygons.

`fill(X,Y,C)` creates filled polygons from the data in `X` and `Y` with vertex color specified by `C`. `C` is a vector or matrix used as an index into the colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`. If necessary, `fill` closes the polygon by connecting the last vertex to the first.

`fill(X,Y,ColorSpec)` fills two-dimensional polygons specified by `X` and `Y` with the color specified by `ColorSpec`.

`fill(X1,Y1,C1,X2,Y2,C2,...)` specifies multiple two-dimensional filled areas.

`fill(...,'PropertyName',PropertyValue)` allows you to specify property names and values for a patch graphics object.

`h = fill(...)` returns a vector of handles to patch graphics objects, one handle per patch object.

Examples

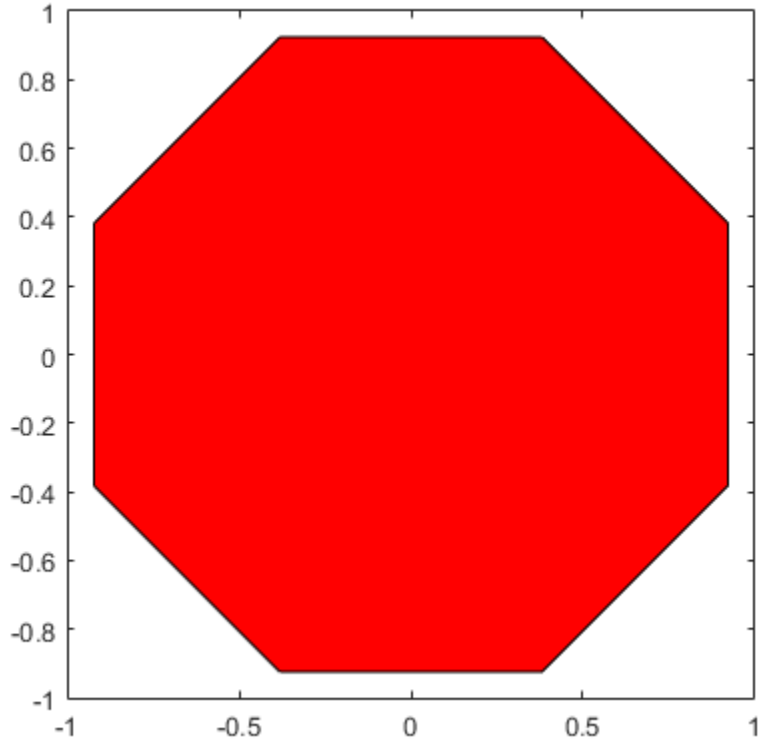
Create Red Octagon

Define the data.

```
t = (1/16:1/8:1)'*2*pi;
x = cos(t);
y = sin(t);
```

Create a red octagon using the `fill` function.

```
fill(x,y,'r')
axis square
```



More About

Tips

If `X` or `Y` is a matrix, and the other is a column vector with the same number of elements as rows in the matrix, `fill` replicates the column vector argument to produce a matrix of the required size. `fill` forms a vertex from corresponding elements in `X` and `Y` and creates one polygon from the data in each column.

If `X` or `Y` contains one or more `NaN` values, then `fill` does not fill the polygons.

The type of color shading depends on how you specify color in the argument list. If you specify color using `ColorSpec`, `fill` generates flat-shaded polygons by setting the patch object's `FaceColor` property to the corresponding RGB triplet.

If you specify color using `C`, `fill` scales the elements of `C` by the values specified by the axes property `CLim`. After scaling `C`, `C` indexes the current colormap.

If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the `n`th patch object, where *n* is the corresponding column in `X` or `Y`.

If `C` is a column vector or a matrix, `fill` uses a linear interpolation of the vertex colors to generate polygons with interpolated colors. It sets the patch graphics object `FaceColor` property to `'interp'` and the elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill` replicates the column vector to produce the required sized matrix.

See Also

Functions

`axis` | `caxis` | `colormap` | `ColorSpec` | `fill3` | `patch`

Properties

Patch Properties

Introduced before R2006a

fill3

Filled 3-D polygons

Syntax

```
fill3(X,Y,Z,C)
fill3(X,Y,Z,ColorSpec)
fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)
fill3(...,'PropertyName',PropertyValue)
h = fill3(...)
```

Description

The `fill3` function creates flat-shaded and Gouraud-shaded polygons.

`fill3(X,Y,Z,C)` fills three-dimensional polygons. `X`, `Y`, and `Z` triplets specify the polygon vertices. If `X`, `Y`, or `Z` is a matrix, `fill3` creates n polygons, where n is the number of columns in the matrix. `fill3` closes the polygons by connecting the last vertex to the first when necessary.

`C` specifies color, where `C` is a vector or matrix of indices into the current colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`.

`fill3(X,Y,Z,ColorSpec)` fills three-dimensional polygons defined by `X`, `Y`, and `Z` with color specified by `ColorSpec`.

`fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)` specifies multiple filled three-dimensional areas.

`fill3(...,'PropertyName',PropertyValue)` allows you to set values for specific patch properties.

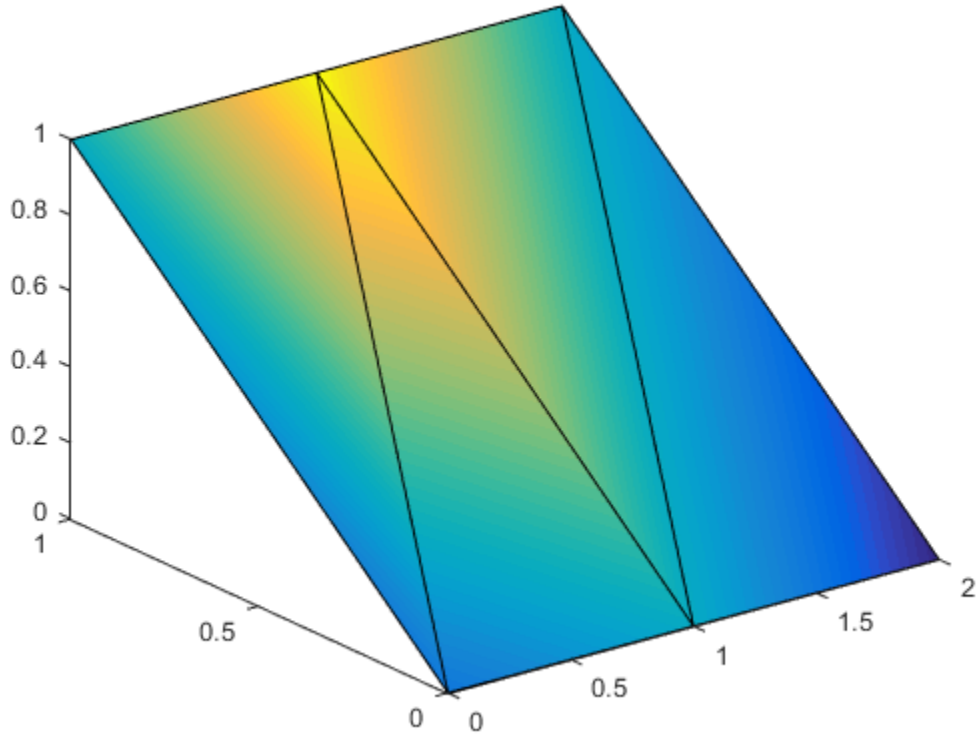
`h = fill3(...)` returns a vector of handles to patch graphics objects, one handle per patch.

Examples

Create Filled 3-D Polygon

Create four triangles with interpolated colors.

```
X = [0 1 1 2; 1 1 2 2; 0 0 1 1];
Y = [1 1 1 1; 1 0 1 0; 0 0 0 0];
Z = [1 1 1 1; 1 0 1 0; 0 0 0 0];
C = [0.5000 1.0000 1.0000 0.5000;
     1.0000 0.5000 0.5000 0.1667;
     0.3330 0.3330 0.5000 0.5000];

figure
fill3(X,Y,Z,C)
```



More About

Algorithms

If `X`, `Y`, and `Z` are matrices of the same size, `fill3` forms a vertex from the corresponding elements of `X`, `Y`, and `Z` (all from the same matrix location), and creates one polygon from the data in each column.

If `X`, `Y`, or `Z` is a matrix, `fill3` replicates any column vector argument to produce matrices of the required size.

If you specify color using `ColorSpec`, `fill3` generates flat-shaded polygons and sets the patch object `FaceColor` property to an RGB triplet.

If you specify color using `C`, `fill3` scales the elements of `CLim` by the axes property `CLim`, which specifies the color axis scaling parameters, before indexing the current colormap.

If `C` is a row vector, `fill3` generates flat-shaded polygons and sets the `FaceColor` property of the patch objects to `'flat'`. Each element becomes the `CData` property value for the respective patch object.

If `C` is a column vector or a matrix, `fill3` generates polygons with interpolated colors and sets the patch object `FaceColor` property to `'interp'`. `fill3` uses a linear interpolation of the vertex colormap indices when generating polygons with interpolated colors. The elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill3` replicates the column vector to produce the required sized matrix.

See Also

Functions

`axis` | `caxis` | `colormap` | `ColorSpec` | `fill` | `patch`

Properties

Patch Properties

Introduced before R2006a

filter

1-D digital filter

Syntax

```
y = filter(b,a,x)
y = filter(b,a,x,zi)
y = filter(b,a,x,zi,dim)
[y,zf] = filter( ___ )
```

Description

`y = filter(b,a,x)` filters the input data, x, using a rational transfer function defined by the numerator and denominator coefficients b and a, respectively.

If `a(1)` is not equal to `1`, then `filter` normalizes the filter coefficients by `a(1)`. Therefore, `a(1)` must be nonzero.

- If `x` is a vector, then `filter` returns the filtered data as a vector of the same size as `x`.
- If `x` is a matrix, then `filter` acts along the first dimension and returns the filtered data for each column.
- If `x` is a multidimensional array, then `filter` acts along the first array dimension whose size does not equal 1.

`y = filter(b,a,x,zi)` uses initial conditions, zi, for the filter delays. The length of zi must equal `max(length(a),length(b))-1`.

`y = filter(b,a,x,zi,dim)` acts along dimension `dim`. For example, if `x` is a matrix, then `filter(b,a,x,zi,2)` returns the filtered data for each row.

`[y,zf] = filter( ___ )` also returns the final conditions, `zf`, of the filter delays, using any of the previous syntax arguments.

Examples

Moving-Average Filter of Vector Data

Find the moving-average of a vector without using a `for` loop.

Create a 1-by-100 row vector of sinusoidal input data corrupted by random noise. Initialize the random number generator to make the output of `rand` repeatable.

```
t = linspace(-pi,pi,100);
rng default
x = sin(t) + 0.25*rand(size(t));
```

A moving-average filter is represented by the following difference equation,

$$y(n) = \frac{1}{windowSize}\left(x(n) + x(n-1) + \ldots + x(n-(windowSize-1))\right).$$

Define the numerator coefficients of the rational transfer function. Use a window size of 5.

```
windowSize = 5;

b = (1/windowSize)*ones(1,windowSize)
```

```
b =

    0.2000    0.2000    0.2000    0.2000    0.2000
```

Define the denominator coefficients of the rational transfer function.

```
a = 1;
```

Find the moving-average of the data with a window size of 5.

```
y = filter(b,a,x);
```

`y(1)` is equivalent to `0.2*x(1)`.

`y(2)` is equivalent to `0.2*(x(1)+x(2))`.

...

`y(5)` is equivalent to `0.2*(sum(x(1:5)) = mean(x(1:5))`.

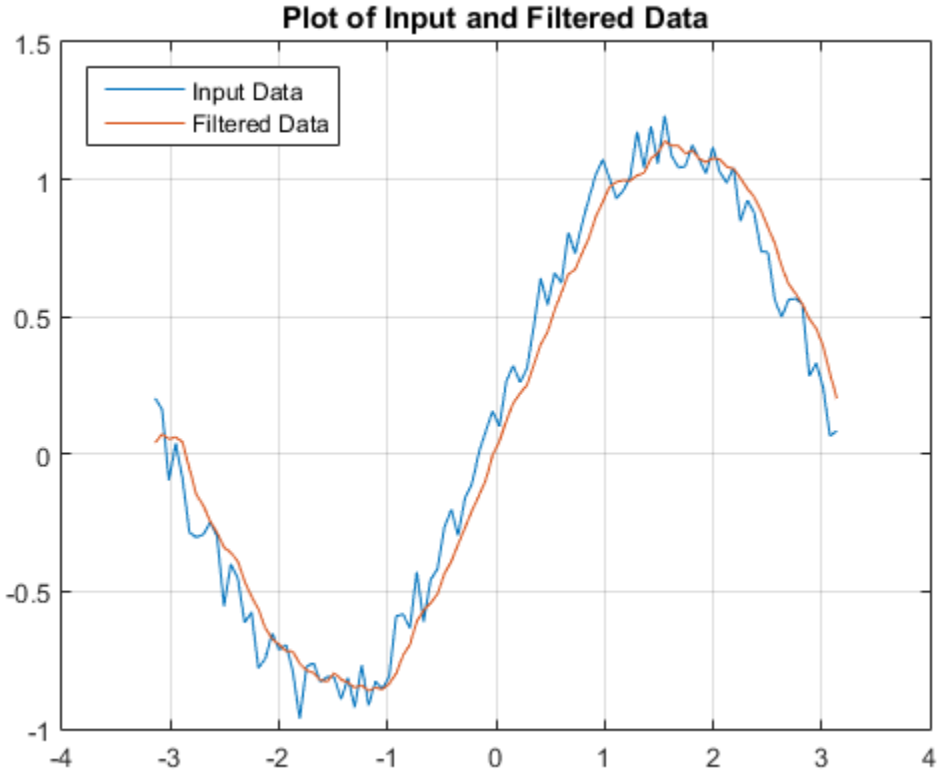
`y(6)` is equivalent to `mean(x(2:6))`.

...

`y(100)` is equivalent to `mean(x(96:100))`.

Compare the original data and the filtered data using an overlaid plot.

```
plot(t,x)
hold on
plot(t,y)

grid on
legend('Input Data','Filtered Data','Location','NorthWest')
title('Plot of Input and Filtered Data')
```



Apply Rational Transfer Function to Matrix

Filter the rows or columns of a matrix with the following rational transfer function,

$$H(z) = \frac{b(1)}{a(1) + a(2)z^{-1}} = \frac{1}{1 - 0.2z^{-1}}.$$

Create a 15-by-2 matrix of random input data. Initialize the random number generator to make the output of `rand` repeatable.

```
rng default
x = rand(15,2)
```

```
x =

    0.8147    0.1419
    0.9058    0.4218
    0.1270    0.9157
    0.9134    0.7922
    0.6324    0.9595
    0.0975    0.6557
    0.2785    0.0357
    0.5469    0.8491
    0.9575    0.9340
    0.9649    0.6787
    0.1576    0.7577
    0.9706    0.7431
    0.9572    0.3922
    0.4854    0.6555
    0.8003    0.1712
```

Define the numerator and denominator coefficients for the rational transfer function.

```
b = 1;
a = [1 -0.2];
```

Apply the transfer function along the first dimension (default) and return the 1-D digital filter of each column.

```
y = filter(b,a,x)
```

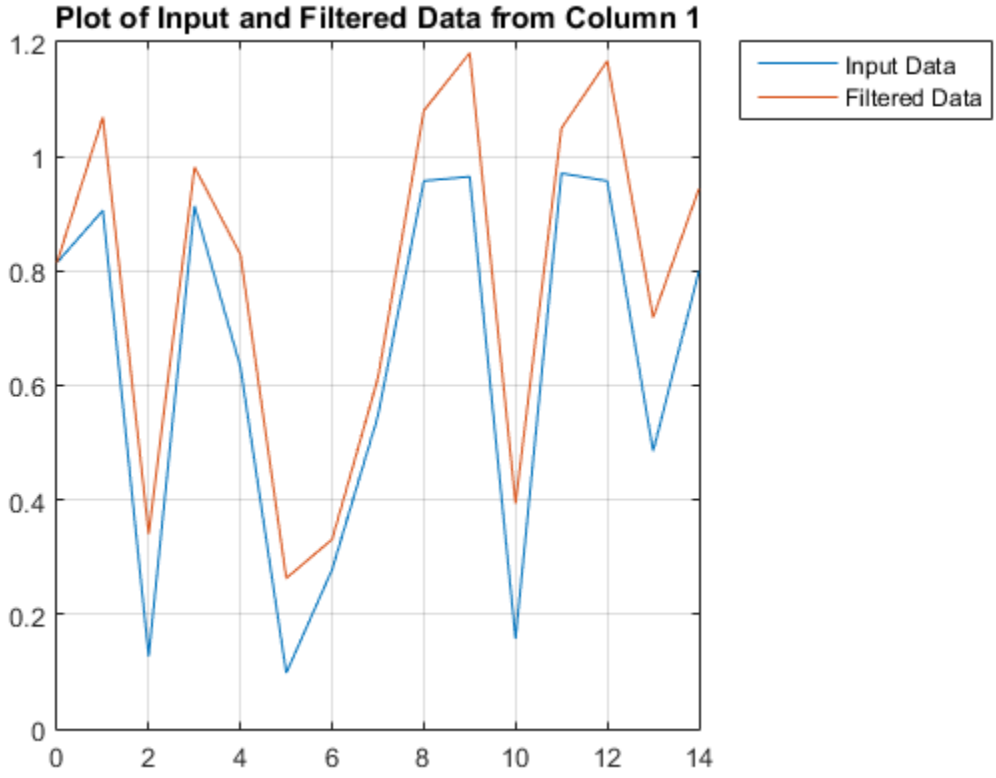
```
y =

    0.8147    0.1419
    1.0687    0.4501
    0.3407    1.0058
    0.9815    0.9934
    0.8287    1.1582
    0.2633    0.8874
    0.3312    0.2132
    0.6131    0.8918
    1.0801    1.1123
    1.1809    0.9012
    0.3938    0.9380
    1.0494    0.9307
    1.1670    0.5784
```

```
    0.7188    0.7712
    0.9440    0.3254
```

Compare the original data and the filtered data using an overlaid plot for each column.

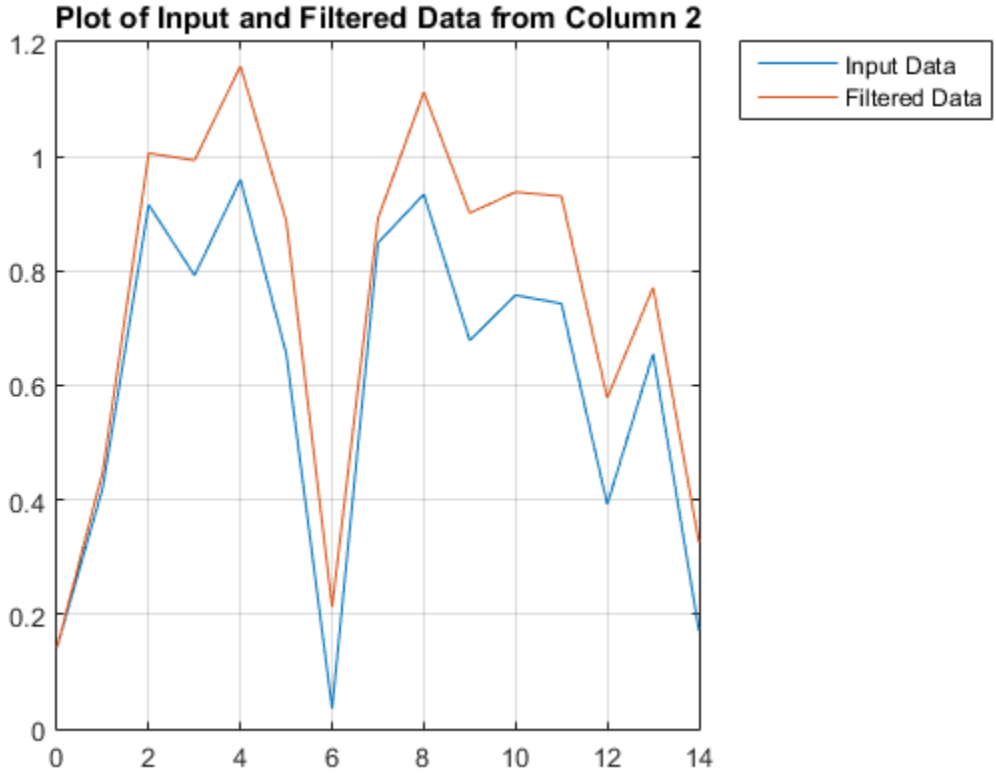
Plot the first column of input and filtered data.

```
t = 0:length(x)-1;

plot(t,x(:,1))
hold on
plot(t,y(:,1))

grid on
legend('Input Data','Filtered Data','Location','BestOutside')
title('Plot of Input and Filtered Data from Column 1')
```



Plot the second column of input and filtered data.

```
figure

plot(t,x(:,2))
hold on
plot(t,y(:,2))

grid on
legend('Input Data','Filtered Data','Location','BestOutside')
title('Plot of Input and Filtered Data from Column 2')
```



Alternatively, you can filter the rows of a matrix by specifying `dim = 2`.

Consider the 2-by-15 matrix, `x'`, whose rows contain the data to filter.

Filter the matrix along the second dimension. Use the default initial conditions for filter delays.

```
dim = 2;
y2 = filter(b,a,x',[],dim)
```

```
y2 =

  Columns 1 through 7
```

```
    0.8147    1.0687    0.3407    0.9815    0.8287    0.2633    0.3312
    0.1419    0.4501    1.0058    0.9934    1.1582    0.8874    0.2132

  Columns 8 through 14

    0.6131    1.0801    1.1809    0.3938    1.0494    1.1670    0.7188
    0.8918    1.1123    0.9012    0.9380    0.9307    0.5784    0.7712

  Column 15

    0.9440
    0.3254
```

`y2` is the transpose of `y` from above.

Filter Data in Sections

Use initial and final conditions for filter delays to filter data in sections, especially if memory limitations are a consideration.

Generate a large random data sequence and split it into two segments, `x1` and `x2`.

```
x = randn(10000,1);

x1 = x(1:5000);
x2 = x(5001:end);
```

The whole sequence, `x`, is the vertical concatenation of `x1` and `x2`.

Define the numerator and denominator coefficients for the rational transfer function,

$$H(z) = \frac{b(1) + b(2)z^{-1}}{a(1) + a(2)z^{-1}} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}.$$

```
b = [2,3];
a = [1,0.2];
```

Filter the subsequences `x1` and `x2` one at a time. Output the final conditions from filtering `x1` to store the internal status of the filter at the end of the first segment.

```
[y1,zf] = filter(b,a,x1);
```

Use the final conditions from filtering `x1` as initial conditions to filter the second segment, `x2`.

```
y2 = filter(b,a,x2,zf);
```

`y1` is the filtered data from `x1`, and `y2` s the filtered data from `x2`. The entire filtered sequence is the vertical concatenation of `y1` and `y2`.

Filter the entire sequence simultaneously for comparison.

```
y = filter(b,a,x);
```

```
isequal(y,[y1;y2])
```

```
ans =

     1
```

Input Arguments

b — Numerator coefficients of rational transfer function

vector

Numerator coefficients of the rational transfer function, specified as a vector.

Data Types: `double` | `single`
Complex Number Support: Yes

a — Denominator coefficients of rational transfer function

vector

Denominator coefficients of the rational transfer function, specified as a vector.

Data Types: `double` | `single`
Complex Number Support: Yes

x — Input data

vector | matrix | multidimensional array

Input data, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single`
Complex Number Support: Yes

`zi` — Initial conditions for filter delays

`[]` (default) | vector | matrix | multidimensional array

Initial conditions for filter delays, specified as a vector, matrix, or multidimensional array.

- If `zi` is a vector, then its length must be `max(length(a),length(b))-1`.
- If `zi` is a matrix or multidimensional array, then the size of the leading dimension must be `max(length(a),length(b))-1`. The size of each remaining dimension must match the size of the corresponding dimension of `x`.

The default value, specified by `[]`, initializes all filter delays to zero.

Data Types: `double` | `single`
Complex Number Support: Yes

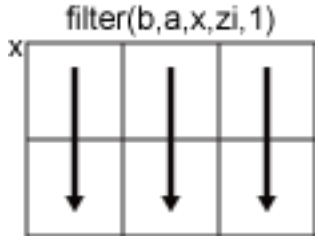
`dim` — Dimension to operate along

positive integer scalar

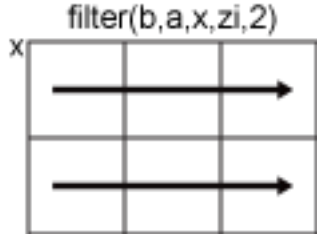
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `x`.

- If `dim = 1`, then `filter(b,a,x,zi,1)` works along the rows of `x` and returns the filter applied to each column.



- If `dim = 2`, then `filter(b,a,zi,2)` works along the columns of `x` and returns the filter applied to each row.



If `dim` is greater than `ndims(x)`, then `filter` returns `x`.

Data Types: `double` | `single`

Output Arguments

`y` — Filtered data

vector | matrix | multidimensional array

Filtered data, returned as a vector, matrix, or multidimensional array of the same size as the input data, `x`.

`zf` — Final conditions for filter delays

vector | matrix | multidimensional array

Final conditions for filter delays, returned as a vector, matrix, or multidimensional array.

- If `x` is a vector, then `zf` is a column vector of length `max(length(a),length(b))-1`.
- If `x` is a matrix or multidimensional array, then `zf` is an array of column vectors of length `max(length(a),length(b))-1`, such that the number of columns in `zf` is equivalent to the number of columns in `x`.

More About

Rational Transfer Function

The input-output description of the `filter` operation on a vector in the Z-transform domain is a rational transfer function. A rational transfer function is of the form,

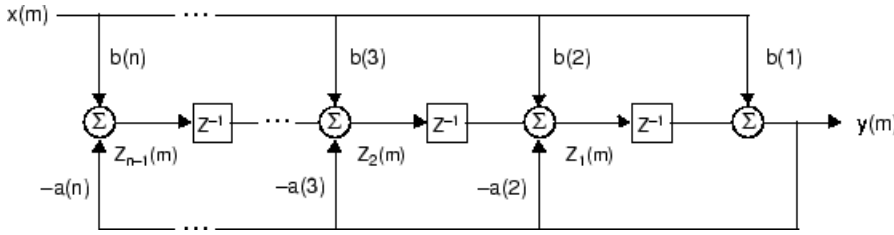
$$Y(z) = \frac{b(1) + b(2)z^{-1} + ... + b(n_b + 1)z^{-n_b}}{1 + a(2)z^{-1} + ... + a(n_a + 1)z^{-n_a}} X(z),$$

which handles both FIR and IIR filters [1]. n_a is the feedback filter order, and n_b is the feedforward filter order.

You also can express the rational transfer function as the following difference equation,

$$\begin{aligned} a(1)y(n) = {} & b(1)x(n) + b(2)x(n-1) + ... + b(n_b + 1)x(n - n_b) \\ & - a(2)y(n-1) - ... - a(n_a + 1)y(n - n_a). \end{aligned}$$

Furthermore, you can represent the rational transfer function using its direct form II transposed implementation, as in the following diagram. Due to normalization, assume `a(1) = 1`.



The operation of `filter` at sample m is given by the time domain difference equations

$$\begin{aligned} y(m) &= b(1)x(m) + z_1(m-1) \\ z_1(m) &= b(2)x(m) + z_2(m-1) - a(2)y(m) \\ \vdots \quad &= \quad \vdots \qquad \vdots \\ z_{n-2}(m) &= b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m) \\ z_{n-1}(m) &= b(n)x(m) - a(n)y(m). \end{aligned}$$

Tips

- If you have the Signal Processing Toolbox™, you can design a filter, `d`, using `designfilt`. Then, you can use `Y = filter(d,X)` to filter your data.
- “Filtering Data”

References

[1] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999.

See Also

`conv` | `designfilt` | `filter2` | `filtfilt` | `filtic`

Introduced before R2006a

filter2

2-D digital filter

Syntax

```
Y = filter2(h,X)
Y = filter2(h,X,shape)
```

Description

`Y = filter2(h,X)` filters the data in `X` with the two-dimensional FIR filter in the matrix `h`. It computes the result, `Y`, using two-dimensional correlation, and returns the central part of the correlation that is the same size as `X`.

`Y = filter2(h,X,shape)` returns the part of `Y` specified by the `shape` parameter. *`shape`* is a string with one of these values:

`'full'`	Returns the full two-dimensional correlation. In this case, `Y` is larger than `X`.
`'same'`	(default) Returns the central part of the correlation. In this case, `Y` is the same size as `X`.
`'valid'`	Returns only those parts of the correlation that are computed without zero-padded edges. In this case, `Y` is smaller than `X`.

More About

Tips

Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how `filter2` performs linear filtering.

Algorithms

Given a matrix `X` and a two-dimensional FIR filter `h`, `filter2` rotates your filter matrix 180 degrees to create a convolution kernel. It then calls `conv2`, the two-dimensional convolution function, to implement the filtering operation.

`filter2` uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` then extracts the central part of the convolution that is the same size as the input matrix, and returns this as the result. If the `shape` parameter specifies an alternate part of the convolution for the result, `filter2` returns the appropriate part.

See Also

`conv2` | `filter`

Introduced before R2006a

find

Find indices and values of nonzero elements

Syntax

```
k = find(X)
k = find(X,n)
k = find(X,n,direction)

[row,col] = find( ___ )
[row,col,v] = find( ___ )
```

Description

`k = find(X)` returns a vector containing the linear indices of each nonzero element in array `X`.

- If `X` is a vector, then `find` returns a vector with the same orientation as `X`.
- If `X` is a multidimensional array, then `find` returns a column vector of the linear indices of the result.
- If `X` contains no nonzero elements or is empty, then `find` returns an empty array.

`k = find(X,n)` returns the first `n` indices corresponding to the nonzero elements in `X`.

`k = find(X,n,direction)`, where `direction` is `'last'`, finds the last `n` nonzero elements in `X`. The default for `direction` is `'first'`, which finds the first `n` nonzero elements.

`[row,col] = find( ___ )` returns the row and column subscripts of each nonzero element in array `X` using any of the input arguments in previous syntaxes.

`[row,col,v] = find( ___ )` also returns vector `v`, which contains the nonzero elements of `X`.

Examples

Zero and Nonzero Elements in Matrix

Find the nonzero elements in a 3-by-3 matrix.

```
X = [1 0 2; 0 1 1; 0 0 4]
```

```
X =

     1     0     2
     0     1     1
     0     0     4
```

```
k = find(X)
```

```
k =

     1
     5
     7
     8
     9
```

Use the logical `not` operator on `X` to locate the zeros.

```
k2 = find(~X)
```

```
k2 =

     2
     3
     4
     6
```

Elements Satisfying a Condition

Find the first five elements that are less than `10` in a 4-by-4 magic square matrix.

```
X = magic(4)
```

```
X =

    16     2     3    13
     5    11    10     8
     9     7     6    12
```

```
     4    14    15     1
```

```
k = find(X<10,5)
```

```
k =

     2
     3
     4
     5
     7
```

View the corresponding elements of `X`.

```
X(k)
```

```
ans =

     5
     9
     4
     2
     7
```

Elements Equal to Specific Values

To find a specific integer value, use the `==` operator. For instance, find the element equal to `13` in a 1-by-10 vector of odd integers.

```
x = 1:2:20
```

```
x =

     1     3     5     7     9    11    13    15    17    19
```

```
k = find(x==13)
```

```
k =

     7
```

To find a noninteger value, use a tolerance value based on your data. Otherwise, the result is sometimes an empty matrix due to floating-point roundoff error.

```
y = 0:0.1:1
```

```
y =
```

```
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000    0.8
k = find(y==0.3)

k =

   Empty matrix: 1-by-0

k = find(abs(y-0.3) < 0.001)

k =

     4
```

Last Several Nonzero Elements

Create a 6-by-6 magic square matrix with all of the odd-indexed elements equal to zero.

```
X = magic(6);
X(1:2:end) = 0

X =

     0     0     0     0     0     0
     3    32     7    21    23    25
     0     0     0     0     0     0
     8    28    33    17    10    15
     0     0     0     0     0     0
     4    36    29    13    18    11
```

Locate the *last* four nonzeros.

```
k = find(X,4,'last')

k =

    30
    32
    34
    36
```

Elements Satisfying Multiple Conditions

Find the first three elements in a 4-by-4 matrix that are greater than 0 and less than 10. Specify two outputs to return the row and column subscripts to the elements.

```
X = [18 3 1 11; 8 10 11 3; 9 14 6 1; 4 3 15 21]

X =

    18     3     1    11
     8    10    11     3
     9    14     6     1
     4     3    15    21

[row,col] = find(X>0 & X<10,3)

row =

     2
     3
     4


col =

     1
     1
     1
```

The first instance is `X(2,1)`, which is `8`.

Subscripts and Values for Nonzero Elements

Find the nonzero elements in a 3-by-3 matrix. Specify three outputs to return the row subscripts, column subscripts, and element values.

```
X = [3 2 0; -5 0 7; 0 0 1]

X =

     3     2     0
    -5     0     7
     0     0     1

[row,col,v] = find(X)

row =

     1
     2
     1
```

```
     2
     3


col =

     1
     1
     2
     3
     3


v =

     3
    -5
     2
     7
     1
```

- “Find Array Elements That Meet a Condition”

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. If `X` is an empty array or has no nonzero elements, then `k` is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

n — Number of nonzeros to find

positive integer scalar

Number of nonzeros to find, specified as a positive integer scalar. By default, `find(X,n)` looks for the first `n` nonzero elements in `X`.

direction — Search direction

`'first'` (default) | `'last'`

Search direction, specified as the string `'first'` or `'last'`. Look for the *last* `n` nonzero elements in `X` using `find(X,n,'last')`.

Output Arguments

`k` — Indices to nonzero elements

vector

Indices to nonzero elements, returned as a vector. If `X` is a row vector, then `k` is also a row vector. Otherwise, `k` is a column vector. `k` is an empty array when `X` is an empty array or has no nonzero elements.

You can return the nonzero values in `X` using `X(k)`.

`row` — Row subscripts

vector

Row subscripts, returned as a vector. Together, `row` and `col` specify the `X(row,col)` subscripts corresponding to the nonzero elements in `X`.

`col` — Column subscripts

vector

Column subscripts, returned as a vector. Together, `row` and `col` specify the `X(row,col)` subscripts corresponding to the nonzero elements in `X`.

`v` — Nonzero elements of `X`

vector

Nonzero elements of `X`, returned as a vector.

More About

Linear Indices

A linear index allows use of a single subscript to index into an array, such as `A(k)`. MATLAB treats the array as a single column vector with each column appended to the bottom of the previous column. Thus, linear indexing numbers the elements in the columns from top to bottom, left to right.

For example, consider a 3-by-3 matrix. You can reference the `A(2,2)` element with `A(5)`, and the `A(2,3)` element with `A(8)`. The linear index changes depending on the size of the array; `A(5)` returns a differently located element for a 3-by-3 matrix than it does for a 4-by-4 matrix.

The `sub2ind` and `ind2sub` functions are useful in converting between subscripts and linear indices.

Tips

- To find array elements that meet a condition, use `find` in conjunction with a relational expression. For example, `find(X<5)` returns the linear indices to the elements in `X` that are less than `5`.
- To directly find the elements in `X` that satisfy the condition `X<5`, use `X(X<5)`. Avoid function calls like `X(find(X<5))`, which unnecessarily use `find` on a logical matrix.
- When you execute `find` with a relational operation like `X>1`, it is important to remember that the result of the relational operation is a logical matrix of ones and zeros. For example, the command `[row,col,v] = find(X>1)` returns a column vector of logical `1` (`true`) values for `v`.
- The row and column subscripts, `row` and `col`, are related to the linear indices in `k` by `k = sub2ind(size(X),row,col)`.

- "Matrix Indexing"
- "Relational Operations"
- "Sparse Matrix Manipulation"

See Also

`ind2sub` | `ismember` | `Logical Operators: Short-Circuit` | `nonzeros` | `strfind` | `sub2ind`

Introduced before R2006a

findall

Find all graphics objects

Syntax

```
object_handles = findall(handle_list)
object_handles = findall(handle_list,'property','value',...)
```

Description

`object_handles = findall(handle_list)` returns the handles, including hidden handles, of all objects in the hierarchy under the objects identified in `handle_list`.

`object_handles = findall(handle_list,'property','value',...)` returns the handles of all objects in the hierarchy under the objects identified in `handle_list` that have the specified properties set to the specified values.

Examples

```
plot(1:10)
xlabel xlab
a = findall(gcf)
b = findobj(gcf)
c = findall(b,'Type','text')
% returns the xlabel handle twice
d = findobj(b,'Type','text')
% can't find the xlabel handle
```

More About

Tips

`findall` is similar to `findobj`, except that it finds objects even if their `HandleVisibility` is set to `off`.

See Also

`allchild` | `findobj`

Introduced before R2006a

findfigs

Find visible offscreen figures

Syntax

```
findfigs
```

Description

`findfigs` finds all visible figure windows whose display area is off the screen and positions them on the screen.

A window appears to the MATLAB software to be offscreen when its display area (the area not covered by the window's title bar, menu bar, and toolbar) does not appear on the screen.

This function is useful when you are bringing an application from a larger monitor to a smaller one (or one with lower resolution). Windows visible on the larger monitor may appear offscreen on a smaller monitor. Using `findfigs` ensures that all windows appear on the screen.

Introduced before R2006a

findobj

Locate graphics objects with specific properties

Syntax

```
findobj
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h = findobj('PropertyName',PropertyValue,'-logicaloperator',
PropertyName',PropertyValue,...)
h = findobj('-regexp','PropertyName','regexp',...)
h = findobj('-property','PropertyName')
h = findobj(objhandles,...)
h = findobj(objhandles,'-depth',d,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)
```

Description

`findobj` returns handles of the root object and all its descendants without assigning the result to a variable.

`h = findobj` returns handles of the root object and all its descendants.

`h = findobj('PropertyName',PropertyValue,...)` returns handles of all graphics objects having the property *`PropertyName`*, set to the value `PropertyValue`. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h = findobj('PropertyName',PropertyValue,'-logicaloperator', PropertyName',PropertyValue,...)` applies the logical operator to the property value matching. Possible values for *`-logicaloperator`* are:

- `-and`
- `-or`
- `-xor`
- `-not`

For more information on logical operators, see “Logical Operations”.

`h = findobj('-regexp','PropertyName','regexp',...)` matches objects using regular expressions as if the value of you passed the property `PropertyName` to the `regexp` function as

`regexp(PropertyValue,'regexp')`
If a match occurs, `findobj` returns the object handle. See the `regexp` function for information on how the MATLAB software uses regular expressions.

`h = findobj('-property','PropertyName')` finds all objects having the specified property.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendants.

`h = findobj(objhandles,'-depth',d,...)` specifies the depth of the search. The depth argument `d` controls how many levels under the handles in `objhandles` MATLAB traverses. Specify `d` as `inf` to get the default behavior of all levels. Specify `d` as `0` to get the same behavior as using the `flat` argument.

`h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendants.

`findobj` returns an error if a handle refers to a nonexistent graphics object.

`findobj` correctly matches any legal property value. For example,

`findobj('Color','r')`
finds all objects having a `Color` property set to `red`, `r`, or `[1 0 0]`.

When a graphics object is a descendant of more than one object identified in `objhandles`, MATLAB searches the object each time `findobj` encounters its handle. Therefore, implicit references to a graphics object can result in multiple returns of its handle.

To find handle objects that meet specified conditions, use `findobj`.

Examples

Find all line objects in the current axes:

```
h = findobj(gca,'Type','line')
```

Find all objects having a `Label` set to `'foo'` and a `String` set to `'bar'`:

```
h = findobj('Label','foo','-and','String','bar');
```

Find all objects whose `String` is not `'foo'` and is not `'bar'`:

```
h = findobj('-not','String','foo','-not','String','bar');
```

Find all objects having a `String` set to `'foo'` and a `Tag` set to `'button one'` and whose `Color` is not `'red'` or `'blue'`:

```
h = findobj('String','foo','-and','Tag','button one',...
 '-and','-not',{'Color','red','-or','Color','blue'})
```

Find all objects for which you have assigned a value to the `Tag` property (that is, the value is not the empty string `''`):

```
h = findobj('-regexp','Tag','[^'']')
```

Find all children of the current figure that have their `BackgroundColor` property set to a certain shade of gray (`[.7 .7 .7]`). This statement also searches the current figure for the matching property value pair.

```
h = findobj(gcf,'-depth',1,'BackgroundColor',[.7 .7 .7])
```

See Also

`copyobj` | `findall` | `findobj` | `gcf` | `gca` | `gcbo` | `gco` | `get` | `regexp` | `set`

Introduced before R2006a

findstr

Find string within another, longer string

Note: `findstr` is not recommended. Use `strfind` instead.

Syntax

```
k = findstr(str1, str2)
```

Description

`k = findstr(str1, str2)` searches the longer of the two input strings for any occurrences of the shorter string, returning the starting index of each such occurrence in the double array `k`. If no occurrences are found, then `findstr` returns the empty array, `[]`.

The search performed by `findstr` is case sensitive. Any leading and trailing blanks in either input string are explicitly included in the comparison.

Unlike the `strfind` function, the order of the input arguments to `findstr` is not important. This can be useful if you are not certain which of the two input strings is the longer one.

Examples

```
s = 'Find the starting indices of the shorter string.';

findstr(s, 'the')
ans =
     6    30

findstr('the', s)
ans =
```

```
     6    30
```

See Also

`strfind` | `strtok` | `strcmp` | `strncmp` | `strcmpi` | `strncmpi` | `regexp` | `regexpi` | `regexprep`

Introduced before R2006a

finish

Termination file for MATLAB program

Description

When the MATLAB program quits, it runs a script called `finish.m`, if the script exists and is on the search path MATLAB uses or in the current directory. `finish.m` is a file you create that instructs MATLAB to perform any final tasks before terminating. For example, you can save the data in your workspace to a MAT-file.

`finish.m` is invoked whenever you do one of the following:

- Click the Close box in the MATLAB desktop on Microsoft Windows platforms or the equivalent on UNIX platforms
- Type `quit` or `exit` at the command prompt

Examples

Two sample termination files are provided with MATLAB in *`matlabroot`*`/toolbox/local`.

- `finishsav.m` — Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m` — Displays a dialog box allowing you to cancel quitting and saves the workspace. See also the "Confirmation Dialogs Preferences" and the option for exiting MATLAB.

To create a termination file, make a copy of one of these sample files and change the name to `finish.m`. Make sure that the file is on the MATLAB path. You can modify the file to include any operations you want the termination file to perform.

More About

Tips

When using Handle Graphics features in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible.

- “Exit MATLAB”

See Also

`quit` | `exit` | `startup`

Introduced before R2006a

fitsdisp

Display FITS metadata

Syntax

```
fitsdisp(filename)
fitsdisp(filename,Name,Value)
```

Description

`fitsdisp(filename)` displays metadata for all the Header/Data Units (HDUs) found in the FITS file specified by `filename`.

`fitsdisp(filename,Name,Value)` displays metadata for all the Header/Data Units (HDUs) found in the FITS file with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`filename`

Text string specifying the name of an existing FITS file.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Index'`

Positive scalar value or vector specifying the HDUs.

Default:

`'Mode'`

One of the following strings:

- `standard` – Display standard keywords
- `min` – Display only HDU types and sizes
- `full` – Display all HDU keywords

Default: `standard`

Examples

Display metadata in the 2nd HDU in the FITS file.

```
fitsdisp('tst0012.fits','Index',2);
```

Display the metadata in the 1st, 3rd, and 5th HDUs in a file.

```
fitsdisp('tst0012.fits','Index',[1 3 5]);
```

Display all metadata in the 5th HDU in a file

```
fitsdisp('tst0012.fits','Index',5,'Mode','full');
```

References

For copyright information, see the `cfitsiocopyright.txt` file.

See Also

`fitsread` | `fitswrite` | `fitsinfo`

Introduced in R2012a

fitsinfo

Information about FITS file

Syntax

```
info = fitsinfo(filename)
```

Description

`info = fitsinfo(filename)` returns the structure, `info`, with fields that contain information about the contents of a Flexible Image Transport System (FITS) file. `filename` is a string enclosed in single quotes that specifies the name of the FITS file.

The `info` structure contains the following fields, listed in the order they appear in the structure. In addition, the info structure can also contain information about any number of optional file components, called *extensions* in FITS terminology. For more information, see "FITS File Extensions" on page 1-2665.

Field Name	Description	Return Type
`Filename`	Name of the file	String
`FileModDate`	File modification date	String
`FileSize`	Size of the file in bytes	`double`
`Contents`	List of extensions in the file in the order that they occur	Cell array of strings
`PrimaryData`	Information about the primary data in the FITS file	Structure array

PrimaryData

The `PrimaryData` field is a structure that describes the primary data in the file. The following table lists the fields in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String

Field Name	Description	Return Type
`Size`	Size of each dimension. The number of rows correspond to the value of the NAXIS2 keyword, while the number of columns correspond to the value of the NAXIS1 keyword. Any further dimensions correspond to NAXIS3, NAXIS4, and so on.	`double` array
`DataSize`	Size of the primary data in bytes	`double`
`MissingDataValue`	Value used to represent undefined data	`double`
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	`double`
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	`double`
`Offset`	Number of bytes from beginning of the file to the location of the first data value	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing keywords, values, and comments of the header in each column	Cell array of strings

FITS File Extensions

A FITS file can also include optional extensions. If the file contains any of these extensions, the `info` structure can contain these additional fields.

- AsciiTable — Numeric information in tabular format, stored as ASCII characters
- BinaryTable — Numeric information in tabular format, stored in binary representation
- Image — A multidimensional array of pixels

- Unknown — Nonstandard extension

AsciiTable Extension

The `AsciiTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	`double`
`RowSize`	Number of characters in each row	`double`
`NFields`	Number of fields in each row	`double` array
`FieldFormat`	A 1-by-`NFields` cell containing formats in which each field is encoded. The formats are FORTRAN-77 format codes.	Cell array of strings
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of strings
`FieldWidth`	A 1-by-`NFields` array containing the number of characters in each field	`double` array
`FieldPos`	A 1-by-`NFields` array of numbers representing the starting column for each field	`double` array
`DataSize`	Size of the data in the table in bytes	`double`
`MissingDataValue`	A 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of strings
`Intercept`	A 1-by-`NFields` array of numbers used along with `Slope` to calculate actual data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	`double` array
`Slope`	A 1-by-`NFields` array of numbers used with `Intercept` to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	`double` array

Field Name	Description	Return Type
`Offset`	Number of bytes from beginning of the file to the location of the first data value in the table	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, Values and Comments in the ASCII table header	Cell array of strings

BinaryTable Extension

The `BinaryTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	`double`
`RowSize`	Number of bytes in each row	`double`
`NFields`	Number of fields in each row	`double`
`FieldFormat`	A 1-by-`NFields` cell array containing the data type of the data in each field. The data type is represented by a FITS binary table format code.	Cell array of strings
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of strings
`FieldSize`	A 1-by-`NFields` array, where each element contains the number of values in the Nth field	`double` array
`DataSize`	Size of the data in the Binary Table, in bytes. Includes any data past the main table.	`double`
`MissingDataValue`	An 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of `double`
`Intercept`	A 1-by-`NFields` array of numbers used along with Slope to calculate actual data values from the array data values	`double` array

Field Name	Description	Return Type
	using the equation: `actual_value = slope*array_value+Intercept`	
`Slope`	A 1-by-`NFields` array of numbers used with Intercept to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	`double` array
`Offset`	Number of bytes from beginning of the file to the location of the first data value	`double`
`ExtensionSize`	Size of any data past the main table, in bytes	`double`
`ExtensionOffset`	Number of bytes from the beginning of the file to any data past the main table	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Image Extension

The `Image` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Size of each dimension. The number of rows correspond to the value of the NAXIS2 keyword, while the number of columns correspond to the value of the NAXIS1 keyword. Any further dimensions correspond to NAXIS3, NAXIS4, and so on.	`double` array
`DataSize`	Size of the data in the Image extension in bytes	`double`
`Offset`	Number of bytes from the beginning of the file to the first data value	`double`

Field Name	Description	Return Type
`MissingDataValue`	Value used to represent undefined data	`double`
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value+Intercept`	`double`
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Unknown Structure

The `Unknown` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Sizes of each dimension	`double` array
`DataSize`	Size of the data in nonstandard extensions, in bytes	`double`
`Offset`	Number of bytes from beginning of the file to the first data value	`double`
`MissingDataValue`	Representation of undefined data	`double`
`Intercept`	Value, used with `Slope`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	`double`

Field Name	Description	Return Type
`Slope`	Value, used with `Intercept`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Examples

Get Information About FITS File

Use `fitsinfo` to obtain information about the FITS file `tst0012.fits`. In addition to its primary data, the file also contains an example of the extensions `BinaryTable`, `Unknown`, `Image`, and `AsciiTable`.

```
S = fitsinfo('tst0012.fits')
```

```
S =
          Filename: 'matlabroot\toolbox\matlab\demos\tst0012.fits'
       FileModDate: '12-Mar-2001 18:37:46'
          FileSize: 109440
          Contents: {'Primary' 'Binary Table' 'Unknown'
'Image' 'ASCII Table'}
       PrimaryData: [1x1 struct]
       BinaryTable: [1x1 struct]
           Unknown: [1x1 struct]
             Image: [1x1 struct]
        AsciiTable: [1x1 struct]
```

View the `PrimaryData` field.

```
S.PrimaryData
```

```
ans =

              DataType: 'single'
                  Size: [109 102]
              DataSize: 44472
```

```
    MissingDataValue: []
           Intercept: 0
               Slope: 1
              Offset: 2880
            Keywords: {25x3 cell}
```

The `PrimaryData` field describes the data in the file. For example, the `Size` field indicates the data is a 109-by-102 matrix.

View the `AsciiTable` field.

```
S.AsciiTable
```

```
ans =
              Rows: 53
           RowSize: 59
           NFields: 8
       FieldFormat: {'A9' 'F6.2' 'I3' 'E10.4' 'D20.15' 'A5' 'A1' 'I4'}
    FieldPrecision: {1x8 cell}
        FieldWidth: [9 6.2000 3 10.4000 20.1500 5 1 4]
          FieldPos: [1 11 18 22 33 54 54 55]
          DataSize: 3127
  MissingDataValue: {'*' '---.--' '  *' [] '*' '*' '*' ''}
         Intercept: [0 0 -70.2000 0 0 0 0 0]
             Slope: [1 1 2.1000 1 1 1 1 1]
            Offset: 103680
          Keywords: {65x3 cell}
```

The `AsciiTable` field describes the `AsciiTable` extension. For example, using the `FieldWidth` and `FieldPos` fields you can determine the length and location of each field within a row.

See Also

`fitsread` | `fitswrite` | `fitsdisp`

Introduced before R2006a

fitsread

Read data from FITS file

Syntax

```
data = fitsread(filename)
data = fitsread(filename,extname)
data = fitsread(filename,extname,index)
data = fitsread(filename,Name,Value)
```

Description

`data = fitsread(filename)` reads the primary data of the Flexible Image Transport System (FITS) file specified by the text string `filename`. The function replaces undefined data values with `NaN` and scales numeric data by the slope and intercept values, always returning double precision values.

`data = fitsread(filename,extname)` reads data from the FITS file extension specified by `extname`.

`data = fitsread(filename,extname,index)` reads data from the FITS file extension specified by `extname` . If there is more than one of the specified extensions in the file, `index` specifies the one to read.

`data = fitsread(filename,Name,Value)` reads data from the FITS file with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`filename`

Text string specifying the name of a FITS file.

Default:

`extname`

One of the following text strings specifying the name of a data array or extension in the FITS file. To determine the contents of a FITS, view the `Contents` field of the structure returned by `fitsinfo`.

Data Arrays or Extensions

Extname	Description
`'primary'`	Read data from the primary data array.
`'asciitable'`	Read data from the ASCII Table extension. The return value, `data`, is a 1-D cell array.
`'binarytable'`	Read data from the Binary Table extension. The return value, `data`, is a 1-D cell array.
`'image'`	Read data from the Image extension.
`'unknown'`	Read data from the Unknown extension.

Default:

`index`

Numeric value specifying which extension to read, if more than one exists in the file.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'info'`

`info` structure returned by `fitsinfo` specifying the location of data to read.

Note: Using the `info` structure returned by `fitsinfo` to specify the location of data in a FITS file can significantly improve performance, especially when reading multiple images from the file.

Default:

`'PixelRegion'`

Cell array `{rows,cols,...}` specifying the boundaries of a subimage region to read from the file. Each dimension (`rows`, `cols`) is a vector of 1-based indices given either as `START`, `[START STOP]`, or `[START INCREMENT STOP]`. This parameter is valid only for primary or image extensions.

Default:

`'raw'`

Specifies that `fitsread` should not scale the data read from the file or replace undefined values with `NaN`. Data read from the file is the same class as it is stored in the file.

Default:

`'TableColumns'`

Vector of 1-based indices specifying the columns to read from the ASCII or Binary table extension. This vector should contain unique and valid indices into the table data specified in increasing order. This parameter is valid only for ASCII or Binary extensions.

Default:

`'TableRows'`

Vector of 1-based indices specifying the rows to read from the ASCII or Binary table extension. This vector should contain unique and valid indices into the table data specified in increasing order. This parameter is valid only for ASCII or Binary extensions.

Default:

Output Arguments

`data`

Data returned from the FITS file.

Examples

Read primary data from FITS file

```
data = fitsread('tst0012.fits');

Name        Size             Bytes  Class     Attributes

data      109x102            88944  double
```

Inspect available extensions, read 'image' extension using the `extname` option.

```
info = fitsinfo('tst0012.fits');
 % List of contents, includes any extensions if present.
 disp(info.Contents);
 imageData = fitsread('tst0012.fits','image');
```

Subsample the fifth plane of 'image' extension by 2.

```
info        = fitsinfo('tst0012.fits');
 rowend      = info.Image.Size(1);
 colend      = info.Image.Size(2);
 primaryData = fitsread('tst0012.fits','image',...
               'Info', info,...
               'PixelRegion',{[1 2 rowend], [1 2 colend], 5 });
```

Read every other row from an ASCII table.

```
info      = fitsinfo('tst0012.fits');
rowend    = info.AsciiTable.Rows;
tableData = fitsread('tst0012.fits','asciitable',...
                     'Info',info,...
                     'TableRows',[1:2:rowend]);
```

Read all data for the first, second and fifth columns of the Binary table.

```
info      = fitsinfo('tst0012.fits');
rowend    = info.BinaryTable.Rows;
tableData = fitsread('tst0012.fits','binarytable',...
                     'Info',info,...
                     'TableColumns',[1 2 5]);
```

More About

extension

A FITS file contains primary data and can optionally contain any number of optional components, called *extensions* in FITS terminology.

See Also

`fitswrite` | `fitsinfo` | `fitsdisp`

Introduced before R2006a

fitswrite

Write image to FITS file

Syntax

```
fitswrite(imagedata,filename)
fitswrite(imagedata,filename,Name,Value)
```

Description

`fitswrite(imagedata,filename)` writes `imagedata` to the FITS file specified by `filename`. If `filename` does not exist, `fitswrite` creates the file as a simple FITS file. If `filename` exists, `fitswrite` overwrites the file or appends the image to the end of the file, depending on the value of the `writemode` argument.

`fitswrite(imagedata,filename,Name,Value)` writes `imagedata` to the FITS file specified by `filename` with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`imagedata`

Image array.

Default:

`filename`

Text string specifying the name of an existing FITS file or the name you want to assign to a new FITS file.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'WriteMode'`

One of these strings:

- `overwrite`
- `append`

Default: `overwrite`

`'Compression'`

One of these strings:

- `none`
- `gzip`
- `rice`
- `hcompress`
- `plio`

Default: `none`

Examples

Create a FITS file containing the red channel of an RGB image.

```
X = imread('ngc6543a.jpg');
R = X(:,:,1);
fitswrite(R,'myfile.fits');
fitsdisp('myfile.fits');
```

Create a FITS file with three images constructed from the channels of an RGB image.

```
X = imread('ngc6543a.jpg');
```

```
R = X(:,:,1);  G = X(:,:,2);  B = X(:,:,3);
fitswrite(R,'myfile.fits');
fitswrite(G,'myfile.fits','writemode','append');
fitswrite(B,'myfile.fits','writemode','append');
fitsdisp('myfile.fits');
```

References

For copyright information, see the `cfitsiocopyright.txt` file.

See Also

`fitsinfo` | `fitsread`

Introduced in R2012a

fix

Round toward zero

Syntax

```
Y = fix(X)
```

Description

`Y = fix(X)` rounds each element of `X` to the nearest integer toward zero. For positive `X`, the behavior of `fix` is the same as `floor`. For negative `X`, the behavior of `fix` is the same as `ceil`.

Examples

Round Matrix Elements Toward Zero

```
X = [-1.9 -3.4; 1.6 2.5; -4.5 4.5]
```

```
X =

   -1.9000   -3.4000
    1.6000    2.5000
   -4.5000    4.5000
```

```
Y = fix(X)
```

```
Y =

    -1    -3
     1     2
    -4     4
```

Round Complex Numbers Toward Zero

```
X = [1.4+2.3i 3.1-2.2i -5.3+10.9i]
```

```
X =

   1.4000 + 2.3000i   3.1000 - 2.2000i  -5.3000 +10.9000i
```

```
Y = fix(X)
```

```
Y =

   1.0000 + 2.0000i   3.0000 - 2.0000i  -5.0000 +10.0000i
```

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex `X`, `fix` treats the real and imaginary parts independently.

`fix` converts logical and `char` elements of `X` into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

More About

- “Integers”
- “Floating-Point Numbers”

See Also

`ceil` | `floor` | `round`

Introduced before R2006a

matlab.unittest.fixtures

Summary of classes in MATLAB Fixtures Interface

Description

Fixtures ease creation of setup and teardown code. The `matlab.unittest.fixtures` package consists of the following customized MATLAB fixtures.

matlab.unittest.fixtures.CurrentFolderFixture	Fixture for changing current working folder
matlab.unittest.fixtures.Fixture	Interface class for test fixtures
matlab.unittest.fixtures.PathFixture	Fixture for adding a folder to the MATLAB path
matlab.unittest.fixtures.SuppressedWarningsFixture	Fixture to suppress display of warnings
matlab.unittest.fixtures.TemporaryFolderFixture	Fixture for creating a temporary folder

Related Examples

- "Write Tests Using Shared Fixtures"
- "Create Basic Custom Fixture"
- "Create Advanced Custom Fixture"

matlab.unittest.fixtures.CurrentFolderFixture class

Package: matlab.unittest.fixtures

Fixture for changing current working folder

Description

The `CurrentFolderFixture` class provides a fixture for changing the current working folder. When the test framework sets up the fixture, it changes the working folder. When the test framework tears down the fixture, it restores the working folder to its previous state.

Construction

`matlab.unittest.fixtures.CurrentFolderFixture(folder)` constructs a fixture for changing the current working folder to `folder`.

Input Arguments

`folder` — Folder to make the current working folder

string

Folder to make the current working folder, specified as a string. MATLAB throws an error if `folder` does not exist.

Properties

`Folder`

Folder to make the current working folder, specified as a string in the `folder` input argument.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Fixture to Change Current Working Folder

Create the following `changeFolderFixtureTest` class definition on your MATLAB path. This example assumes that the subfolder `helperFiles` exists in your working folder. Create the `changeToFolder`in your working folder if it does not exist.

The `test1` function includes a call to `pwd` to demonstrate the current path changed to the `helperFiles` folder.

```
classdef changeFolderFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.CurrentFolderFixture

            changeToFolder = 'helperFiles';
            testCase.applyFixture(CurrentFolderFixture ...
                (changeToFolder));
            pwd
        end
    end
end
```

At the command prompt, run the test. For the purposes of this example, call `pwd` before and after `run` to show the fixture was properly torn down and the path returned to the pre-test state.

```
currentFolderBeforeTest = pwd
run(changeFolderFixtureTest);
currentFolderAfterTest = pwd
```

```
currentFolderBeforeTest =

H:\Documents\doc_examples

Running changeFolderFixtureTest

ans =

H:\Documents\doc_examples\helperFiles

.
Done changeFolderFixtureTest
```

```
__________

currentFolderAfterTest =

H:\Documents\doc_examples
```

See Also

matlab.unittest.fixtures.PathFixture | matlab.unittest.TestCase.applyFixture | matlab.unittest.fixtures

matlab.unittest.fixtures.Fixture class

Package: matlab.unittest.fixtures

Interface class for test fixtures

Description

The `Fixture` interface class is the means by which test authors create custom fixtures. Fixtures configure the environment state required for tests.

Classes deriving from the `Fixture` interface must implement the `setup` method. This method executes the changes to the environment. A fixture should restore the environment to its initial state when it is torn down. To restore the environment, use the `addTeardown` method in the `setup` method or implement the fixture's `teardown` method.

Subclasses can set the `SetupDescription` and `TeardownDescription` properties in their constructors to provide descriptions for the actions performed by the `setup` and `teardown` methods. The testing framework can display these descriptions when setting up and tearing down the fixture.

A class that derives from `Fixture` must implement the `isCompatible` method if its constructor accepts any input arguments or is otherwise configurable. `Fixture` subclasses use this method to define a notion of interchangeability of fixtures. Two `matlab.unittest.fixtures` instances of the same class are considered to be interchangeable if the `isCompatible` method returns `true`. The `TestRunner` uses the result of `isCompatible` to determine whether two fixture instances of the same class correspond to the same shared test fixture state.

Properties

`SetupDescription`

Description of fixture setup actions, specified as a string. The `SetupDescription` property describes the actions the fixture performs when the testing framework invokes the fixture's `setup` method.

`TeardownDescription`

Description of fixture teardown actions, specified as a string. The `TeardownDescription` property describes the actions the fixture performs when the testing framework invokes the fixture's `teardown` method.

Methods

log	Record diagnostic information
setup	Set up fixture
teardown	Tear down fixture
addTeardown	Dynamically add teardown routine
isCompatibile	Determine if two fixtures of the same class are interchangeable

Events

`AssertionFailed`	Triggered upon failing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssertionPassed`	Triggered upon passing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionFailed`	Triggered upon failing assumption. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionPassed`	Triggered upon passing assumption. A `QualificationEventData` object is passed to listener callback functions.

`FatalAssertionFailed`	Triggered upon failing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionPassed`	Triggered upon passing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`ExceptionThrown`	Triggered by the `TestRunner` when an exception is thrown. An `ExceptionEventData` object is passed to listener callback functions.
`DiagnosticLogged`	Triggered by the `TestRunner` upon a call to the `log` method. A `LoggedDiagnosticEventData` object is passed to the listener callback functions.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`matlab.unittest.TestCase.getSharedTestFixtures` | `matlab.unittest.TestCase.applyFixture` | `matlab.unittest.qualifications.QualificationEventData` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.diagnostics.LoggedDiagnosticEventData` | `addTeardown` | `matlab.unittest.fixtures`

Related Examples

- “Create Basic Custom Fixture”
- “Create Advanced Custom Fixture”

log

Class: matlab.unittest.fixtures.Fixture
Package: matlab.unittest.fixtures

Record diagnostic information

Syntax

```
log(f,diagnostic)
log(f,v,diagnostic)
```

Description

`log(f,diagnostic)` logs the supplied diagnostic. The log method provides a means for tests to log information during fixture setup and teardown routines. The testing framework displays logged messages only if you configure it to do so by adding an appropriate plugin, such as the `matlab.unittest.plugins.LoggingPlugin`.

`log(f,v,diagnostic)` logs the diagnostic at the specified verbosity level, `v`.

Input Arguments

`f` — Instance of fixture

`matlab.unittest.fixtures.Fixture` instance

Instance of fixture, specified as a `matlab.unittest.fixtures.Fixture`.

`diagnostic` — Diagnostic information to display upon a failure

string | function handle | `matlab.unittest.diagnostics.Diagnostic` instance

Diagnostic information to display upon a failure, specified as a string, function handle, or `matlab.unittest.diagnostics.Diagnostic` instance.

`v` — Verbosity level

2 (default) | 1 | 3 | 4 | `matlab.unittest.Verbosity` enumeration

Verbosity level, specified as an integer value between 1 and 4 or a `matlab.unittest.Verbosity` enumeration object. The default verbosity level for diagnostic messages is `Concise`. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Corresponding Enumeration Member	Verbosity Description
1	`matlab.unittest.Verbosity.Terse`	Minimal amount of information
2	`matlab.unittest.Verbosity.Concis`	Typical amount of information
3	`matlab.unittest.Verbosity.Detail`	Supplemental amount of information
4	`matlab.unittest.Verbosity.Verbos`	Surplus of information

Examples

Log Diagnostic Information

In a file, `FormatHexFixture.m`, in your current working folder, create the following fixture.

```
classdef FormatHexFixture < matlab.unittest.fixtures.Fixture
    properties (Access=private)
        OriginalFormat
    end
    methods
        function setup(fixture)
            fixture.OriginalFormat = get(0,'Format');
            fixture.log(['The previous format setting was ',...
                fixture.OriginalFormat])
            log(fixture,'Setting Format')
            set(0,'Format','hex')
            log(fixture,3,'Format Set')
        end
        function teardown(fixture)
            log(fixture,'Resetting Format')
            set(0,'Format',fixture.OriginalFormat)
            log(fixture,3,'Original Format Restored')
        end
```

```
    end
end
```

In a file, `SampleTest.m`, in your current working folder, create the following test class.

```
classdef SampleTest < matlab.unittest.TestCase
    methods (Test)
        function test1(testCase)
            testCase.applyFixture(FormatHexFixture);
            actStr = getColumnForDisplay([1;2;3], 'Small Integers');
            expStr = ['Small Integers  '
                '3ff0000000000000'
                '4000000000000000'
                '4008000000000000'];
            testCase.verifyEqual(actStr, expStr)
        end
    end
end

function str = getColumnForDisplay(values, title)
elements = cell(numel(values)+1, 1);
elements{1} = title;
for idx = 1:numel(values)
    elements{idx+1} = displayNumber(values(idx));
end
str = char(elements);
end

function str = displayNumber(n)
str = strtrim(evalc('disp(n);'));
end
```

Run the test.

```
result = run(SampleTest);
```

```
Running SampleTest
.
Done SampleTest
__________
```

None of the logged messages are displayed because the default test runner has a verbosity level of 1 (`Terse`) and the default log message is at level 2 (`Concise`).

Create a test runner to report the diagnostics at levels 1, 2, and 3 and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin

ts = matlab.unittest.TestSuite.fromClass(?SampleTest);
runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(3);
runner.addPlugin(p);

results = runner.run(ts);
```

```
 [Concise] Diagnostic logged (2014-04-23T13:17:35): The previous format setting was sho
 [Concise] Diagnostic logged (2014-04-23T13:17:35): Setting Format
[Detailed] Diagnostic logged (2014-04-23T13:17:35): Format Set
 [Concise] Diagnostic logged (2014-04-23T13:17:35): Resetting Format
[Detailed] Diagnostic logged (2014-04-23T13:17:35): Original Format Restored
```

See Also

`matlab.unittest.plugins.LoggingPlugin` | `matlab.unittest.Verbosity`

Introduced in R2014b

setup

Class: matlab.unittest.fixtures.Fixture
Package: matlab.unittest.fixtures

Set up fixture

Syntax

```
setup(f)
```

Description

`setup(f)` sets up a fixture by performing the defined environment modifications. Classes deriving from the `Fixture` interface must implement the `setup` method. This method executes the changes to the environment. A fixture should restore the environment to its initial state when it is torn down. To restore the environment, use the `addTeardown` method in the `setup` method or implement the fixture's `teardown` method.

Input Arguments

`f`

`matlab.unittest.fixtures.Fixture` instance

See Also

`matlab.unittest.fixtures.Fixture` |
`matlab.unittest.fixtures.Fixture.teardown` |
`matlab.unittest.fixtures.Fixture.addTeardown`

Related Examples

- “Create Basic Custom Fixture”
- “Create Advanced Custom Fixture”

teardown

Class: matlab.unittest.fixtures.Fixture
Package: matlab.unittest.fixtures

Tear down fixture

Syntax

```
teardown(f)
```

Description

`teardown(f)` tears down a fixture by performing the defined actions to restore the environment to the initial state.

Input Arguments

`f`

`matlab.unittest.fixtures.Fixture` instance

Alternatives

Instead of defining a `teardown` method, you can define teardown actions within the `setup` method by implementing the `addTeardown` method.

See Also

`matlab.unittest.fixtures.Fixture` |
`matlab.unittest.fixtures.Fixture.addTeardown` |
`matlab.unittest.fixtures.Fixture.setup`

Related Examples

- “Create Basic Custom Fixture”

- “Create Advanced Custom Fixture”

addTeardown

Class: matlab.unittest.fixtures.Fixture
Package: matlab.unittest.fixtures

Dynamically add teardown routine

Syntax

```
addTeardown(f,tearDownFcn)
addTeardown(f,tearDownFcn,arg1,...,argN)
```

Description

`addTeardown(f,tearDownFcn)` adds the tearDownFcn function handle that defines fixture teardown code to the Fixture instance. The teardown code is executed in the reverse order to which it is added. This is known as LIFO (or Last-In-First-Out).

`addTeardown(f,tearDownFcn,arg1,...,argN)` provides input arguments to the `tearDownFcn`.

Input Arguments

`f`

`matlab.unittest.fixtures.Fixture` instance

`tearDownFcn`

Function that defines the fixture teardown code, specified as a function handle.

`arg1,...,argN`

Input arguments required by tearDownFcn, specified by any type. The argument type is specified by the `tearDownFcn` function argument list.

Alternatives

Instead of defining teardown actions within the `setup` method by implementing the `addTeardown` method, you can implement the `teardown` method.

See Also

`matlab.unittest.fixtures.Fixture` |
`matlab.unittest.fixtures.Fixture.teardown` |
`matlab.unittest.fixtures.Fixture.setup`

Related Examples

- "Create Advanced Custom Fixture"

isCompatibile

Class: matlab.unittest.fixtures.Fixture
Package: matlab.unittest.fixtures

Determine if two fixtures of the same class are interchangeable

Syntax

```
TF = isCompatible(f1, f2)
```

Description

`TF = isCompatible(f1, f2)` determines if two fixtures of the same class are interchangeable. The `isCompatible` method returns either logical `1` (`true`) or logical `0` (`false`).

A class that derives from `Fixture` must implement the `isCompatible` method if its constructor accepts any input arguments or is otherwise configurable. `Fixture` subclasses use this method to define a notion of interchangeability of fixtures. Two `matlab.unittest.fixtures` instances of the same class are considered to be interchangeable if the `isCompatible` method returns `true`. The `TestRunner` uses the result of `isCompatible` to determine whether two fixture instances of the same class correspond to the same shared test fixture state. The test framework always calls the `isCompatible` method with two fixture instances of the same class, so the fixture author does not need to implement code to handle the case where the second fixture is a different `fixtures` class.

Input Arguments

`f`

`matlab.unittest.fixtures.Fixture` instance

Attributes

`Access`	`Protected`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.unittest.fixtures.Fixture`

Related Examples

- "Create Advanced Custom Fixture"

matlab.unittest.fixtures.PathFixture class

Package: matlab.unittest.fixtures

Fixture for adding a folder to the MATLAB path

Description

The `PathFixture` class provides a fixture for adding a folder to the MATLAB path. When the test framework sets up the fixture, it adds the specified folder to the path. When the test framework tears down the fixture, it restores the MATLAB path to its previous state.

Construction

`matlab.unittest.fixtures.PathFixture(folder)` constructs a fixture for adding a folder to the MATLAB path. When the test framework sets up the fixture, it adds `folder` to the path. When it tears down the fixture, it restores the MATLAB path to its previous state.

Input Arguments

`folder` — Folder to add to the MATLAB path

string

Folder to add to the MATLAB path, specified as a string. MATLAB throws an error if `folder` does not exist.

Properties

`Folder`

Folder to add to the MATLAB path, specified as a string in the `folder` input argument.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Add Directory to MATLAB Path for Testing

Create the following `addPathFixtureTest` class definition on your MATLAB path. This example assumes that the subfolder,`helperFiles`, exists in your working folder. If it does not, define `addFolder` to be a directory that exists within your current folder.

```
classdef addPathFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.PathFixture

            addFolder = 'helperFiles';
            f = testCase.applyFixture(PathFixture(addFolder));
            disp(['Added to path: ' f.Folder])
        end
    end
end
```

At the command prompt, run the test.

```
run(addPathFixtureTest);
```

```
Running addPathFixtureTest
Added to path: H:\Documents\doc_examples\helperFiles
.
Done addPathFixtureTest
__________
```

After the tests finish running, the framework removes the folder from the path.

Add Directory to Path Using Shared Test Fixture

Create the following `sharedAddPathFixtureTest` class definition on your MATLAB path. This example assumes that the subfolder, `helperFiles`, exists in your working folder.

```
classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.PathFixture('helperFiles')}) ...
        sharedAddPathFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            f = testCase.getSharedTestFixtures;
```

```
            disp(['Added to path: ' f.Folder])
        end
    end
end
```

At the command prompt, run the test.

```
run(sharedAddPathFixtureTest);
```

```
Setting up PathFixture
Done setting up PathFixture: Added 'H:\Documents\doc_examples\helperFiles' to the path.
__________

Running sharedAddPathFixtureTest
Added to path: H:\Documents\doc_examples\helperFiles
.
Done sharedAddPathFixtureTest
__________

Tearing down PathFixture
Done tearing down PathFixture: Restored the path to its original state.
__________
```

After the tests finish running, the framework removes the folder from the path.

See Also

`matlab.unittest.fixtures.CurrentFolderFixture` | `matlab.unittest.TestCase.applyFixture` | `matlab.unittest.fixtures`

matlab.unittest.fixtures.SuppressedWarningsFixture class

Package: matlab.unittest.fixtures

Fixture to suppress display of warnings

Description

The `SuppressedWarningsFixture` class provides a fixture to suppress the display of warnings. When set up, `SuppressedWarningsFixture` disables one ore more specified warnings. When torn down, the fixture restores the states of warnings to their previous values.

Construction

`matlab.unittest.fixtures.SuppressedWarningsFixture(warnIDs)` constructs a fixture to suppress the display of one or more warnings.

Input Arguments

`warnIDs` — Identifier for warnings disabled when the fixture is set up

string | cell array of strings

Warning identifiers for the warnings to be suppressed, specified as a string or cell array of strings.

Properties

`Warnings`

Warning identifiers describing warnings to suppress specified as a cell array of strings in the `warnings` input argument.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Fixture to Suppress Warnings

Suppress the warning that occurs when you try to remove a folder from the search path that is not on the search path.

Remove the folder, `folderthatisnotonpath` from your path, assuming it does not exist.

```
rmpath('folderthatisnotonpath')
```

```
Warning: "folderthatisnotonpath" not found in path.
> In rmpath at 58
```

A warning appears because `rmpath` cannot find the folder.

Suppress the warning during testing by creating the following `suppressWarningsTest` class definition on your MATLAB path.

```
classdef suppressWarningsTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.SuppressedWarningsFixture

            testCase.applyFixture(...
                SuppressedWarningsFixture('MATLAB:rmpath:DirNotFound'));

            % would otherwise cause warning
            rmpath('folderthatisnotonpath')
        end
    end
end
```

At the command prompt, run the test. For the purposes of this example, call `rmpath` before and after running the test to show the warning is not suppressed outside execution of the test.

```
rmpath('folderthatisnotonpath')
run(suppressWarningsTest);
rmpath('folderthatisnotonpath')
```

```
Warning: "folderthatisnotonpath" not found in path.
> In rmpath at 58
Running suppressWarningsTest
.
Done suppressWarningsTest
__________

Warning: "folderthatisnotonpath" not found in path.
> In rmpath at 58
```

Note that the call to `rmpath` within `suppressWarningsTest` does not result in a warning.

See Also

`matlab.unittest.TestCase.applyFixture` | `matlab.unittest.fixtures` | `warning`

matlab.unittest.fixtures.TemporaryFolderFixture class

Package: matlab.unittest.fixtures

Fixture for creating a temporary folder

Description

The `matlab.unittest.fixtures.TemporaryFolderFixture` provides a fixture to create a temporary folder. When the testing framework sets up the fixture, it creates the temporary folder. When it tears down the fixture, it deletes the folder and all its contents. Before it deletes the folder, the fixture clears from memory the definitions of any MATLAB-files, P-files, and MEX-files that are defined in the temporary folder.

Construction

`matlab.unittest.fixtures.TemporaryFolderFixture` constructs a fixture for creating a temporary folder.

`matlab.unittest.fixtures.TemporaryFolderFixture(Name,Value)` constructs a fixture for creating a temporary folder with additional options specified by one or more Name,Value pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'PreservingOnFailure'` — Preservation state of temporary folder and contents after test failure
`false` (default) | `true`

Indicator of whether the temporary folder and its contents are preserved in the event of a test failure, specified as `false` or `true` (`logical 0` or `1`). This property is `false` by default. You can specify it as `true` during fixture construction.

Data Types: `logical`

`'WithSuffix'` — Suffix for temporary folder name

string (default)

Suffix for temporary folder name, specified as a string.

Properties

`Folder`

Absolute path of the folder created by the fixture, specified as a string.

`PreserveOnFailure`

Indicator of whether the temporary folder and its contents are preserved in the event of a test failure. This property is `logical(0)` or `logical(1)`. It is `logical(0)` by default but is set to `logical(1)` if the `'PreservingOnFailure'` input value is set to `true` during fixture construction.

`Suffix`

Suffix used for temporary folder, specified as a string in the `Name,Value` pair argument, `'WithSuffix'`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Temporary Folder Fixture

Create the following `tempFolderFixtureTest` class definition on your MATLAB path.

```
classdef tempFolderFixtureTest < matlab.unittest.TestCase
    methods(Test)
```

```
        function test1(testCase)
            import matlab.unittest.fixtures.TemporaryFolderFixture

            tempFolder = testCase.applyFixture(TemporaryFolderFixture);

            disp(['The temporary folder: ' tempFolder.Folder])
        end
    end
end
```

At the command prompt, run the test.

```
run(tempFolderFixtureTest);
```

```
Running tempFolderFixtureTest
The temporary folder: C:\Temp\tpfb1ae2cf_c9de_4de3_9557_00d52bfcc1b2
.
Done tempFolderFixtureTest
__________
```

The name of the temporary folder varies.

Create Temporary Folder Fixture Persisting Through Test Failure

Create the following `anotherTempFolderFixtureTest` class definition on your MATLAB path. For the purposes of this example, the `test1` function contains an assertion that causes test failure.

```
classdef anotherTempFolderFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.TemporaryFolderFixture

            testCase.applyFixture(TemporaryFolderFixture( ...
                'PreservingOnFailure',true,'WithSuffix','TestData'));

            % Failed assertion, perserved temporary folder
            testCase.assertEqual(1,2)
        end
    end
end
```

At the command prompt, run the test.

```
run(anotherTempFolderFixtureTest);
```

```
Running anotherTempFolderFixtureTest

================================================================================
Assertion failed in anotherTempFolderFixtureTest/test1 and it did not run to completion.

    ---------------------
    Framework Diagnostic:
    ---------------------
    assertEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
```

```
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        1         2           -1       -0.5

    Actual Value:
             1
    Expected Value:
             2

    ------------------
    Stack Information:
    ------------------
    In C:\Documents\anotherTempFolderFixtureTest.m (anotherTempFolderFixtureTest.test1) at 10
================================================================================
   [Terse] Diagnostic logged (2014-04-01T13:50:51):
Because of a failure in the test using the TemporaryFolderFixture, the following folder will not be deleted:
C:\Temp\tp9f5aa9f1_ead1_4462_91f2_08bbe7d0316cTestData


.
Done anotherTempFolderFixtureTest
__________

Failure Summary:

     Name                                Failed  Incomplete  Reason(s)
    ================================================================================
     anotherTempFolderFixtureTest/test1    X         X       Failed by assertion.
```

The test failed and the temporary folder persists. You can open the temporary folder, shown here as `C:\Temp\tp9f5aa9f1_ead1_4462_91f2_08bbe7d0316cTestData`, and examine any contents.

See Also

`matlab.unittest.fixtures.PathFixture` | `matlab.unittest.fixtures.CurrentFolderFixture` | `matlab.unittest.TestCase.applyFixture` | `matlab.unittest.fixtures`

flintmax

Largest consecutive integer in floating-point format

Syntax

```
f = flintmax
f = flintmax(precision)
```

Description

`f = flintmax` returns the largest consecutive integer in IEEE® double precision, which is `2^53`. Above this value, double-precision format does not have integer precision, and not all integers can be represented exactly.

`f = flintmax(precision)` returns the largest consecutive integer in IEEE single or double precision. `flintmax` returns `single(2^24)` for single precision and `2^53` for double precision.

Examples

Double Precision

Return the largest consecutive integer in IEEE double precision.

```
format long e
f = flintmax
```

```
f =

     9.007199254740992e+15
```

This is `2^53`.

Single Precision

Return the largest consecutive integer in IEEE single precision.

```
f = flintmax('single')

f =

    16777216
```

This is `single(2^24)`.

Check the class of `f`.

```
class(f)

ans =

single
```

Limit of Integer Single Precision

Above the value returned by `flintmax('single')`, not all integers can be represented exactly with single precision.

Return the largest consecutive integer in IEEE single precision.

```
f = flintmax('single')

f =

    16777216
```

This is `single(2^24)`.

Add `1` to the value returned from `flintmax`.

```
f1 = f + 1

f1 =

    16777216
```

`f1` is the same as `f`.

```
isequal(f,f1)

ans =

     1
```

Add 2 to the value returned from `flintmax`.

```
f2 = f + 2
```

```
f2 =

    16777218
```

`16777218` is represented exactly in single precision while `16777217` is not.

Input Arguments

`precision` — Floating-point precision type

`'double'` (default) | `'single'`

Floating-point precision type, specified as `'double'` or `'single'`.

Data Types: `char`

Output Arguments

`f` — Largest consecutive integer in floating-point format

scalar constant

Largest consecutive integer in floating-point format returned as a scalar constant. This constant is `2^53` for double precision and `single(2^24)` for single precision.

More About

- "Floating-Point Numbers"

See Also

`eps` | `format` | `intmax` | `realmax`

flip

Flip order of elements

Syntax

```
B = flip(A)
B = flip(A,dim)
```

Description

`B = flip(A)` returns array `B` the same size as `A`, but with the order of the elements reversed. The dimension that is reordered in `B` depends on the shape of `A`:

- If `A` is vector, then `flip(A)` reverses the order of the elements along the length of the vector.
- If `A` is a matrix, then `flip(A)` reverses the elements in each column.
- If `A` is an N-D array, then `flip(A)` operates on the first dimension of `A` in which the size value is not `1`.

`B = flip(A,dim)` reverses the order of the elements in `A` along dimension `dim`. For example, if `A` is a matrix, then `flip(A,1)` reverses the elements in each column, and `flip(A,2)` reverses the elements in each row.

Examples

Flip String of Characters

```
A = 'no word, no bond, row on.';
B = flip(A)
```

```
B =

.no wor ,dnob on ,drow on
```

Flip Column Vector

```
A = [1;2;3];
```

```
B = flip(A)

B =

     3
     2
     1
```

Flip Matrix

Create a diagonal matrix, A.

```
A = diag([100 200 300])

A =

   100     0     0
     0   200     0
     0     0   300
```

Flip A without specifying the `dim` argument.

```
B = flip(A)

B =

     0     0   300
     0   200     0
   100     0     0
```

Now, flip A along the second dimension.

```
B = flip(A,2)

B =

     0     0   100
     0   200     0
   300     0     0
```

Flip N-D Array

Create a 1-by-3-by-2 array.

```
A = zeros(1,3,2);
```

```
A(:,:,1) = [1 2 3];
A(:,:,2) = [4 5 6];
A
```

```
A(:,:,1) =

     1     2     3


A(:,:,2) =

     4     5     6
```

Flip `A` without specifying the `dim` argument.

```
B = flip(A)
```

```
B(:,:,1) =

     3     2     1


B(:,:,2) =

     6     5     4
```

Now, flip `A` along the third dimension.

```
B = flip(A,3)
```

```
B(:,:,1) =

     4     5     6


B(:,:,2) =

     1     2     3
```

Flip Cell Array

Create a 3-by-2 cell array.

```
A = {'foo',1000; 999,true; 'aaa','bbb'}
```

```
A =
```

```
    'foo'    [1000]
    [999]    [   1]
    'aaa'    'bbb'
```

Flip A without specifying the `dim` argument.

```
B = flip(A)
```

```
B =

    'aaa'    'bbb'
    [999]    [   1]
    'foo'    [1000]
```

Now, flip A along the second dimension.

```
B = flip(A,2)
```

```
B =

    [1000]    'foo'
    [   1]    [999]
    'bbb'     'aaa'
```

Input Arguments

A — Input array

vector | matrix | array | cell array | table

Input array, specified as a vector, matrix, array, cell array, or table.

Example: `[1 2 3 4]`

Example: `['abcde']`

Example: `[1 2; 3 4]`

Example: `{'abcde',[1 2 3]}`

Example: `table(rand(1,5),rand(1,5))`

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

The following illustration shows the difference between `dim=1` and `dim=2` when `A` is a matrix.

See Also

`fliplr` | `flipud` | `permute` | `rot90` | `transpose`

flipdim

Flip array along specified dimension

Compatibility

`flipdim` will be removed in a future release. Use `flip` instead.

Syntax

```
B = flipdim(A,dim)
```

Description

`B = flipdim(A,dim)` returns `A` with dimension `dim` flipped.

When the value of `dim` is `1`, the array is flipped row-wise down. When `dim` is `2`, the array is flipped columnwise left to right. `flipdim(A,1)` is the same as `flipud(A)`, and `flipdim(A,2)` is the same as `fliplr(A)`.

Examples

`flipdim(A,1)` where

```
A =

     1     4
     2     5
     3     6
```

produces

```
     3     6
     2     5
```

1 4

See Also

`fliplr` | `flipud` | `permute` | `rot90` | `flip`

Introduced before R2006a

fliplr

Flip array left to right

Syntax

```
B = fliplr(A)
```

Description

`B = fliplr(A)` returns `A` with its columns flipped in the left-right direction (that is, about a vertical axis).

If `A` is a row vector, then `fliplr(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a column vector, then `fliplr(A)` simply returns `A`. For multidimensional arrays, `fliplr` operates on the planes formed by the first and second dimensions.

Examples

Flip Row Vector

Create a row vector.

```
A = 1:10
```

```
A =

     1     2     3     4     5     6     7     8     9    10
```

Use `fliplr` to flip the elements of `A` in the horizontal direction.

```
B = fliplr(A)
```

```
B =

    10     9     8     7     6     5     4     3     2     1
```

The order of the elements in B is reversed compared to A.

Flip Cell Array of Strings

Create a 3-by-3 cell array of strings.

```
A = {'a' 'b' 'c'; 'd' 'e' 'f'; 'g' 'h' 'i'}
```

```
A =

    'a'    'b'    'c'
    'd'    'e'    'f'
    'g'    'h'    'i'
```

Change the order of the columns in the horizontal direction by using `fliplr`.

```
B = fliplr(A)
```

```
B =

    'c'    'b'    'a'
    'f'    'e'    'd'
    'i'    'h'    'g'
```

The order of the first and third columns of A is switched in B, while the second column remains unchanged.

Flip Multidimensional Array

Create a multidimensional array.

```
A = cat(3, [1 2; 3 4], [5 6; 7 8])
```

```
A(:,:,1) =

     1     2
     3     4


A(:,:,2) =

     5     6
     7     8
```

A is an array of size 2-by-2-by-2.

Flip the elements on each page of A in the horizontal direction.

```
B = fliplr(A)
```

```
B(:,:,1) =

     2     1
     4     3


B(:,:,2) =

     6     5
     8     7
```

The result, B, is the same size as A, but the horizontal order of the elements is flipped. The operation flips the elements on each page independently.

Input Arguments

A — Input array

vector | matrix | array | cell array | categorical array | table

Input array, specified as a vector, matrix, array, cell array, categorical array, or table of any data type.

Complex Number Support: Yes

More About

Tips

- `fliplr(A)` is equivalent to `flip(A,2)`.
- Use the `flipud` function to flip arrays in the vertical direction (that is, about a horizontal axis).
- The `flip` function can flip arrays in any direction.

See Also

`flip` | `flipud` | `rot90`

Introduced before R2006a

flipud

Flip array up to down

Syntax

```
B = flipud(A)
```

Description

`B = flipud(A)` returns `A` with its rows flipped in the up-down direction (that is, about a horizontal axis).

If `A` is a column vector, then `flipud(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a row vector, then `flipud(A)` simply returns `A`. For multidimensional arrays, `flipud` operates on the planes formed by the first and second dimensions.

Examples

Flip Column Vector

Create a column vector.

```
A=(1:10)'
```

```
A =

     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
```

Use `flipud` to flip the elements of A in the vertical direction.

```
B = flipud(A)
```

```
B =

    10
     9
     8
     7
     6
     5
     4
     3
     2
     1
```

The order of the elements in B is reversed compared to A.

Flip Cell Array of Strings

Create a 3-by-3 cell array of strings.

```
A = {'a' 'b' 'c'; 'd' 'e' 'f'; 'g' 'h' 'i'}
```

```
A =

    'a'    'b'    'c'
    'd'    'e'    'f'
    'g'    'h'    'i'
```

Change the order of the rows in the vertical direction by using `flipud`.

```
B = flipud(A)
```

```
B =

    'g'    'h'    'i'
    'd'    'e'    'f'
    'a'    'b'    'c'
```

The order of the first and third rows of A is switched in B, while the second row remains unchanged.

Flip Multidimensional Array

Create a multidimensional array.

```
A = cat(3, [1 2; 3 4], [5 6; 7 8])

A(:,:,1) =

     1     2
     3     4


A(:,:,2) =

     5     6
     7     8
```

A is an array of size 2-by-2-by-2.

Flip the elements on each page of A in the vertical direction.

```
B = flipud(A)

B(:,:,1) =

     3     4
     1     2


B(:,:,2) =

     7     8
     5     6
```

The result, B, is the same size as A, but the vertical order of the elements is flipped. The operation flips the elements on each page independently.

Input Arguments

A — Input array

vector | matrix | array | cell array | categorical array | table

Input array, specified as a vector, matrix, array, cell array, categorical array, or table of any data type.
Complex Number Support: Yes

More About

Tips

- `flipud(A)` is equivalent to `flip(A,1)`.
- Use the `fliplr` function to flip arrays in the horizontal direction (that is, about a vertical axis).
- The `flip` function can flip arrays in any direction.

See Also

`flip` | `fliplr` | `rot90`

Introduced before R2006a

floor

Round toward negative infinity

Syntax

```
Y = floor(X)
```

```
Y = floor(t)
Y = floor(t,unit)
```

Description

`Y = floor(X)` rounds each element of `X` to the nearest integer less than or equal to that element.

`Y = floor(t)` rounds each element of the `duration` array `t` to the nearest number of seconds less than or equal to that element.

`Y = floor(t,unit)` rounds each element of `t` to the nearest number of the specified unit of time less than or equal to that element.

Examples

Round Matrix Elements Toward Negative Infinity

```
X = [-1.9  -0.2  3.4;  5.6  7.0  2.4+3.6i];
Y = floor(X)
```

```
Y =

  -2.0000 + 0.0000i  -1.0000 + 0.0000i   3.0000 + 0.0000i
   5.0000 + 0.0000i   7.0000 + 0.0000i   2.0000 + 3.0000i
```

Round Duration Values Toward Negative Infinity

Round each value in a `duration` array to the nearest number of seconds less than or equal to that value.

```
t = hours(8) + minutes(29:31) + seconds(1.23);
t.Format = 'hh:mm:ss.SS'
```

```
t =

   08:29:01.23   08:30:01.23   08:31:01.23
```

```
Y1 = floor(t)
```

```
Y1 =

   08:29:01.00   08:30:01.00   08:31:01.00
```

Round each value in `t` to the nearest number of hours less than or equal to that value.

```
Y2 = floor(t,'hours')
```

```
Y2 =

   08:00:00.00   08:00:00.00   08:00:00.00
```

Input Arguments

`X` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex `X`, `floor` treats the real and imaginary parts independently.

`floor` converts logical and `char` elements of `X` into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

`t` — Input duration

`duration` array

Input duration, specified as a `duration` array.

`unit` — Unit of time

`'seconds'` (default) | `'minutes'` | `'hours'` | `'days'`

Unit of time, specified as `'seconds'`, `'minutes'`, `'hours'`, or `'days'`.

More About

- "Integers"
- "Floating-Point Numbers"

See Also

`ceil` | `fix` | `round`

Introduced before R2006a

flow

Simple function of three variables

Syntax

```
v = flow
v = flow(n)
v = flow(x,y,z)
[x,y,z,v] = flow(...)
```

Description

`flow`, a function of three variables, generates fluid-flow data that is useful for demonstrating `slice`, `interp3`, and other functions that visualize scalar volume data.

`v = flow` produces a 25-by-50-by-25 array.

`v = flow(n)` produces a `n`-by-`2n`-by-`n` array.

`v = flow(x,y,z)` evaluates the speed profile at the points `x`, `y`, and `z`.

`[x,y,z,v] = flow(...)` returns the coordinates as well as the volume data.

More About

- "Slicing Fluid Flow Data"

See Also

`slice` | `interp3`

Introduced before R2006a

fminbnd

Find minimum of single-variable function on fixed interval

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
x = fminbnd(problem)
[x,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

Description

`fminbnd` finds the minimum of a function of one variable within a fixed interval.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the function that is described in `fun` in the interval `x1 < x < x2`. `fun` is a `function_handle`.

"Parameterizing Functions" in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminbnd` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge. See "Iterative Display" in MATLAB Mathematics for more information.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed.

`MaxIter`	Maximum number of iterations allowed.
`OutputFcn`	User-defined function that is called at each iteration. See "Output Functions" in MATLAB Mathematics for more information.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`). • `@optimplotx` plots the current point • `@optimplotfval` plots the function value • `@optimplotfunccount` plots the function count See "Plot Functions" in MATLAB Mathematics for more information.
`TolX`	Termination tolerance on `x`.

`x = fminbnd(problem)` finds the minimum for `problem`, where `problem` is a structure with the following fields:

`objective`	Objective function
`x1`	Left endpoint
`x2`	Right endpoint
`solver`	`'fminbnd'`
`options`	Options structure created using `optimset`

`[x,fval] = fminbnd(...)` returns the value of the objective function computed in `fun` at `x`.

`[x,fval,exitflag] = fminbnd(...)` returns a value `exitflag` that describes the exit condition of `fminbnd`:

`1`	`fminbnd` converged to a solution `x` based on `options.TolX`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.
`-2`	Bounds are inconsistent (`x1 > x2`).

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization in the following fields:

`algorithm`	`'golden section search, parabolic interpolation'`
`funcCount`	Number of function evaluations
`iterations`	Number of iterations
`message`	Exit message

Arguments

`fun` is the function to be minimized. `fun` accepts a scalar `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for a function file

```
x = fminbnd(@myfun,x1,x2);
```

where `myfun.m` is a function file such as

```
function f = myfun(x)
f = ...         % Compute function value at x.
```

or as a function handle for an anonymous function:

```
x = fminbnd(@(x) sin(x*x),x1,x2);
```

Other arguments are described in the syntax descriptions above.

Examples

`x = fminbnd(@cos,3,4)` computes π to a few decimal places and gives a message on termination.

```
[x,fval,exitflag] = ...
   fminbnd(@cos,3,4,optimset('TolX',1e-12,'Display','off'))
```

computes π to about 12 decimal places, suppresses output, returns the function value at `x`, and returns an `exitflag` of 1.

The argument `fun` can also be a function handle for an anonymous function. For example, to find the minimum of the function $f(x) = x^3 - 2x - 5$ on the interval `(0,2)`, create an anonymous function `f`

```
f = @(x)x.^3-2*x-5;
```

Then invoke `fminbnd` with

```
x = fminbnd(f, 0, 2)
```

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)
```

```
y =
   -6.0887
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following function file:

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminbnd`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminbnd` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminbnd(@(x) myfun(x,a),0,1)
   ```

Limitations

The function to be minimized must be continuous. `fminbnd` may only give local solutions.

`fminbnd` often exhibits slow convergence when the solution is on a boundary of the interval.

`fminbnd` only handles real variables.

More About

Algorithms

`fminbnd` is a function file. Its algorithm is based on golden section search and parabolic interpolation. Unless the left endpoint x_1 is very close to the right endpoint x_2, `fminbnd` never evaluates `fun` at the endpoints, so `fun` need only be defined for x in the interval $x_1 < x < x_2$.

If the minimum actually occurs at x_1 or x_2, `fminbnd` returns a point `x` in the interior of the interval (x_1,x_2) that is close to the minimizer. In this case, the distance of `x` from the minimizer is no more than `2*(TolX + 3*abs(x)*sqrt(eps))`. See [1] or [2] for details about the algorithm.

- anonymous function

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

[2] Brent, Richard. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973

See Also

`fminsearch` | `fzero` | `optimset` | `function_handle`

Introduced before R2006a

fminsearch

Find minimum of unconstrained multivariable function using derivative-free method

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
x = fminsearch(problem)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

Description

`fminsearch` finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminsearch(fun,x0)` starts at the point `x0` and returns a value `x` that is a local minimizer of the function described in `fun`. `x0` can be a scalar, vector, or matrix. `fun` is a `function_handle`.

"Parameterizing Functions" in the MATLAB Mathematics documentation explains how to pass additional parameters to your objective function `fun`. See also "Example 2" on page 1-2741 and "Example 3" on page 1-2742 below.

`x = fminsearch(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminsearch` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge. See "Iterative Display" in MATLAB Mathematics for more information.

`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex`, `Inf` or `NaN`. `'off'` (the default) displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed
`MaxIter`	Maximum number of iterations allowed
`OutputFcn`	User-defined function that is called at each iteration. See "Output Functions" in MATLAB Mathematics for more information.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[ ]`). • `@optimplotx` plots the current point • `@optimplotfval` plots the function value • `@optimplotfunccount` plots the function count See "Plot Functions" in MATLAB Mathematics for more information.
`TolFun`	Termination tolerance on the function value
`TolX`	Termination tolerance on `x`

`x = fminsearch(problem)` finds the minimum for `problem`, where `problem` is a structure with the following fields:

`objective`	Objective function
`x0`	Initial point for `x`
`solver`	`'fminsearch'`
`options`	Options structure created using `optimset`

`[x,fval] = fminsearch(...)` returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch(...)` returns a value `exitflag` that describes the exit condition of `fminsearch`:

`1`	`fminsearch` converged to a solution `x`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.

`[x,fval,exitflag,output] = fminsearch(...)` returns a structure `output` that contains information about the optimization in the following fields:

`algorithm`	`'Nelder-Mead simplex direct search'`
`funcCount`	Number of function evaluations
`iterations`	Number of iterations
`message`	Exit message

Arguments

`fun` is the function to be minimized. It accepts an input `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for a function file

```
x = fminsearch(@myfun, x0)
```

where `myfun` is a function file such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

or as a function handle for an anonymous function, such as

```
x = fminsearch(@(x)sin(x^2), x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Example 1

The Rosenbrock banana function is a classic test example for multidimensional minimization:

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 .$$

The minimum is at `(1,1)` and has the value `0`. The traditional starting point is `(-1.2,1)`. The anonymous function shown here defines the function and returns a function handle called `banana`:

```
banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to `fminsearch`:

```
[x,fval] = fminsearch(banana,[-1.2, 1])
```

This produces

```
x =

    1.0000    1.0000

fval =

    8.1777e-010
```

This indicates that the minimizer was found to at least four decimal places with a value near zero.

Example 2

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following function file:

```
function f = myfun(x,a)
f = x(1)^2 + a*x(2)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminsearch`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminsearch` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

```
x = fminsearch(@(x) myfun(x,a),[0,1])
```

Example 3

You can modify the first example by adding a parameter a to the second term of the banana function:

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (a - x_1)^2.$$

This changes the location of the minimum to the point `[a,a^2]`. To minimize this function for a specific value of `a`, for example `a = sqrt(2)`, create a one-argument anonymous function that captures the value of `a`.

```
a = sqrt(2);
banana = @(x)100*(x(2)-x(1)^2)^2+(a-x(1))^2;
```

Then the statement

```
[x,fval] = fminsearch(banana, [-1.2, 1], ...
   optimset('TolX',1e-8));
```

seeks the minimum `[sqrt(2), 2]` to an accuracy higher than the default on `x`.

Limitations

`fminsearch` can often handle discontinuity, particularly if it does not occur near the solution. fminsearch may only give local solutions.

`fminsearch` only minimizes over the real numbers, that is, x must only consist of real numbers and $f(x)$ must only return real numbers. When x has complex variables, they must be split into real and imaginary parts.

More About

Algorithms

fminsearch uses the simplex search method of Lagarias et al. [1]. This is a direct search method that does not use numerical or analytic gradients.

If `n` is the length of `x`, a simplex in `n`-dimensional space is characterized by the `n+1` distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

For more information, see “fminsearch Algorithm”.

- anonymous function

References

[1] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, “Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions,” *SIAM Journal of Optimization*, Vol. 9 Number 1, pp. 112-147, 1998.

See Also

`fminbnd` | `optimset` | `function_handle`

Introduced before R2006a

fopen

Open file, or obtain information about open files

Syntax

```
fileID = fopen(filename)
fileID = fopen(filename,permission)
fileID = fopen(filename,permission,machinefmt,encodingIn)
[fileID,errmsg] = fopen( ___ )

fIDs = fopen('all')

filename = fopen(fileID)
[filename,permission,machinefmt,encodingOut] = fopen(fileID)
```

Description

`fileID = fopen(filename)` opens the file, `filename`, for binary read access, and returns an integer file identifier equal to or greater than 3. MATLAB reserves file identifiers `0`, `1`, and `2` for standard input, standard output (the screen), and standard error, respectively.

If `fopen` cannot open the file, then `fileID` is `-1`.

`fileID = fopen(filename,permission)` opens the file with the type of access specified by `permission`.

`fileID = fopen(filename,permission,machinefmt,encodingIn)` additionally specifies the order for reading or writing bytes or bits in the file using the `machinefmt` argument. The optional `encodingIn` argument specifies the character encoding scheme associated with the file.

`[fileID,errmsg] = fopen( ___ )` additionally returns a system-dependent error message if `fopen` fails to open the file. Otherwise, `errmsg` is an empty string. You can use this syntax with any of the input arguments of the previous syntaxes.

`fIDs = fopen('all')` returns a row vector containing the file identifiers of all open files. The identifiers reserved for standard input, output, and error are not included. The number of elements in the vector is equal to the number of open files.

`filename = fopen(fileID)` returns the file name that a previous call to `fopen` used when it opened the file specified by `fileID`. The output filename is resolved to the full path. The `fopen` function does not read information from the file to determine the output value.

`[filename,permission,machinefmt,encodingOut] = fopen(fileID)` additionally returns the permission, machine format, and encoding that a previous call to `fopen` used when it opened the specified file. If the file was opened in binary mode, `permission` includes the letter `'b'`. The `encodingOut` output is a standard encoding scheme name. `fopen` does not read information from the file to determine these output values. An invalid `fileID` returns empty strings for all output arguments.

Examples

Open File and Pass Identifier to File I/O Function

Open a file and pass the file identifier to the `fgetl` function to read data.

Open the file, `airfoil.m`, and obtain the file identifier.

```
fileID = fopen('airfoil.m');
```

Pass the `fileID` to the `fgetl` function to read one line from the file. Then, close the file.

```
tline = fgetl(fileID);
fclose(fileID)
```

Request Name of File to Open

Create a prompt to request the name of a file to open. If `fopen` cannot open the file, display the relevant error message.

```
fileID = -1;
errmsg = '';
while fileID < 0
   disp(errmsg);
   filename = input('Open file: ', 's');
   [fileID,errmsg] = fopen(filename);
```

```
end
```

Open File for Writing and Specify Access Type, Writing Order, Character Encoding

Open a file to write to a file using the Shift-JIS character encoding.

```
fileID = fopen('japanese_out.txt','w','n','Shift_JIS');
```

The `'w'` input specifies write access, the `'n'` input specifies native byte ordering, and `'Shift_JIS'` specifies the character encoding scheme.

Get Information About Open Files

Suppose you previously opened a file using `fopen`.

```
fileID = fopen('airfoil.m');
```

Get the file identifiers of all open files.

```
fIDs = fopen('all')
```

```
fIDs =

     3
```

Get the file name and character encoding for the open file. Use ~ in place of output arguments you want to omit.

```
[filename,~,~,encoding] = fopen(3)
```

```
filename =

matlabroot\toolbox\matlab\demos\airfoil.m


encoding =

windows-1252
```

The output shown here is representative. Your results might differ.

Input Arguments

`filename` — Name of file to open

string

Name of the file to open, including the file extension, specified as a string. If the file is not in the current folder, `filename` must include a full or a relative path.

On UNIX systems, if `filename` begins with `'~/'` or `'~username/'`, the `fopen` function expands the path to the current or specified user's home directory, respectively.

- If you open a file with read access and the file is not in the current folder, then `fopen` searches along the MATLAB search path.
- If you open a file with write or append access and the file is not in the current folder, then `fopen` creates a file in the current directory.

Example: `'myFile.txt'`

Data Types: `char`

`permission` — File access type

`'r'` (default) | `'w'` | `'a'` | `'r+'` | `'w+'` | `'a+'` | `'A'` | `'W'` | ...

File access type, specified as a string. You can open a file in binary mode or in text mode. On UNIX systems, both translation modes have the same effect. To open a file in binary mode, specify one of the following strings.

`'r'`	Open file for reading.
`'w'`	Open or create new file for writing. Discard existing contents, if any.
`'a'`	Open or create new file for writing. Append data to the end of the file.
`'r+'`	Open file for reading and writing.
`'w+'`	Open or create new file for reading and writing. Discard existing contents, if any.
`'a+'`	Open or create new file for reading and writing. Append data to the end of the file.
`'A'`	Append without automatic flushing of the current output buffer. (Used with tape drives.)
`'W'`	Write without automatic flushing of the current output buffer. (Used with tape drives.)

To open files in text mode, attach the letter `'t'` to the `permission` argument, such as `'rt'` or `'wt+'`.

On Windows systems, in text mode:

- Read operations that encounter a carriage return followed by a newline character (`'\r\n'`) remove the carriage return from the input.
- Write operations insert a carriage return before any newline character in the output.

Open or create a new file in text mode if you want to write to it in MATLAB and then open it in Microsoft Notepad, or any text editor that does not recognize `'\n'` as a newline sequence. When writing to the file, end each line with `'\r\n'`. For an example, see `fprintf`. Otherwise, open files in binary mode for better performance.

To read and write to the same file:

- Open the file with a value for `permission` that includes a plus sign, `'+'`.
- Call `fseek` or `frewind` between read and write operations. For example, do not call `fread` followed by `fwrite`, or `fwrite` followed by `fread`, unless you call `fseek` or `frewind` between them.

`machinefmt` — Order for reading or writing bytes or bits

`'n'` (default) | `'b'` | `'l'` | `'s'` | `'a'` | ...

Order for reading or writing bytes or bits in the file, specified as one of the following strings.

`'n'` or `'native'`	Your system byte ordering (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big-endian or little-endian ordering.

`encodingIn` — Character encoding

`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding to use for subsequent read and write operations, including `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`, specified as one of the following strings.

`'Big5'`	`'ISO-8859-1'`	`'windows-932'`
`'GB2312'`	`'ISO-8859-2'`	`'windows-936'`

'EUC-KR'	'ISO-8859-3'	'windows-949'
'EUC-JP'	'ISO-8859-4'	'windows-950'
'GBK'	'ISO-8859-9'	'windows-1250'
'KSC_5601'	'ISO-8859-13'	'windows-1251'
'Macintosh'	'ISO-8859-15'	'windows-1252'
'Shift_JIS'		'windows-1253'
'US-ASCII'		'windows-1254'
'UTF-8'		'windows-1257'

The default encoding is system-dependent.

If you specify a value for encoding that is not in the list of supported values, MATLAB issues a warning. Specifying other encoding names sometimes (but not always) produces correct results.

Data Types: `char`

`fileID` — File identifier of an open file

integer

File identifier of an open file, specified as an integer.

Data Types: `double`

More About

Tips

- In most cases, it is not necessary to open a file in text mode. MATLAB import functions, all UNIX applications, and Microsoft Word and WordPad recognize `'\n'` as a newline indicator.

See Also

`fclose` | `feof` | `ferror` | `fprintf` | `fread` | `frewind` | `fscanf` | `fseek` | `ftell` | `fwrite`

Introduced before R2006a

fopen (serial)

Connect serial port object to device

Syntax

```
fopen(obj)
```

Description

`fopen(obj)` connects the serial port object, `obj` to the device.

Examples

This example creates the serial port object `s`, connects `s` to the device using `fopen`, writes and reads text data, and then disconnects `s` from the device. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
```

More About

Tips

Before you can perform a read or write operation, `obj` must be connected to the device with the `fopen` function. When `obj` is connected to the device:

- Data remaining in the input buffer or the output buffer is flushed.
- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the device. You can connect only one serial port object to a given device.

Some properties are read-only while the serial port object is open (connected), and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the device. If any of these properties are incorrectly configured, then an error is returned when `fopen` is issued and `obj` is not connected to the device. Properties of this type include `BaudRate`, and are associated with device settings.

See Also

`fclose` | `BytesAvailable` | `BytesToOutput` | `Status` | `ValuesReceived` | `ValuesSent`

Introduced before R2006a

for

Execute statements specified number of times

Syntax

```
for index = values
   statements
end
```

Description

`for index = values, statements, end` executes a group of statements in a loop for a specified number of times. *`values`* has one of the following forms:

- *`initVal`*`:`*`endVal`* — Increment the *`index`* variable from *`initVal`* to *`endVal`* by `1`, and repeat execution of *`statements`* until *`index`* is greater than *`endVal`*.
- *`initVal`*`:`*`step`*`:`*`endVal`* — Increment *`index`* by the value *`step`* on each iteration, or decrements *`index`* when *`step`* is negative.
- *`valArray`* — Create a column vector, *`index`*, from subsequent columns of array *`valArray`* on each iteration. For example, on the first iteration, *`index`* `=` *`valArray`*`(:,1)`. The loop executes a maximum of *`n`* times, where *`n`* is the number of columns of *`valArray`*, given by `numel(`*`valArray`*`,1,:)`. The input *`valArray`* can be of any MATLAB data type, including a string, cell array, or struct.

Examples

Assign Matrix Values

Create a Hilbert matrix of order 10.

```
s = 10;
H = zeros(s);

for c = 1:s
    for r = 1:s
```

```
        H(r,c) = 1/(r+c-1);
    end
end
```

Decrement Values

Step by increments of `-0.2`, and display the values.

```
for v = 1.0:-0.2:0.0
   disp(v)
end
```

```
     1

    0.8000

    0.6000

    0.4000

    0.2000

     0
```

Execute Statements for Specified Values

```
for v = [1 5 8 17]
   disp(v)
end
```

```
     1

     5

     8

    17
```

Repeat Statements for Each Matrix Column

```
for I = eye(4,3)
    disp('Current unit vector:')
    disp(I)
```

```
end

Current unit vector:
     1
     0
     0
     0

Current unit vector:
     0
     1
     0
     0

Current unit vector:
     0
     0
     1
     0
```

More About

Tips

- To programmatically exit the loop, use a `break` statement. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.
- Avoid assigning a value to the *`index`* variable within the loop statements. The `for` statement overrides any changes made to *`index`* within the loop.
- To iterate over the values of a single column vector, first transpose it to create a row vector.

See Also

`break` | `colon` | `continue` | `end` | `if` | `parfor` | `return` | `switch`

Introduced before R2006a

format

Set Command Window output display format

Syntax

```
format style
format
```

Description

`format style` changes the output display format in the Command Window to the format specified by `style`.

`format`, by itself, resets the output format to the default, which is the short, fixed-decimal format for floating-point notation and loose line spacing for all output lines.

Numeric formats affect only how numbers appear in Command Window output, not how MATLAB computes or saves them.

Examples

Long Format

Set the output format to the long fixed-decimal format and display the value of `pi`.

```
format long
pi
```

```
ans =

   3.141592653589793
```

Reset Format to Default

Set the output format to the short engineering format with compact line spacing, and then reset the format to the default.

```
format shortEng
format compact
x = rand(3)

x =
   814.7237e-003   913.3759e-003   278.4982e-003
   905.7919e-003   632.3592e-003   546.8815e-003
   126.9868e-003    97.5404e-003   957.5068e-003

format
x


x =

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

Hexadecimal Format

Display the maximum values for integers and real numbers in hexadecimal format.

```
format hex
intmax('uint64')


ans =

   ffffffffffffffff


realmax


ans =

   7fefffffffffffff
```

Short and Long Engineering Notation

Display the difference between `shortEng` and `longEng` formats.

Set the output format to `shortEng`.

```
format shortEng
```

Create a variable and increase its value by a multiple of 10 each time through a `for` loop.

```
A = 5.123456789;
for k = 1:10
   disp(A)
   A = A*10;
end
```

```
     5.1235e+000

    51.2346e+000

   512.3457e+000

     5.1235e+003

    51.2346e+003

   512.3457e+003

     5.1235e+006

    51.2346e+006

   512.3457e+006

     5.1235e+009
```

The values display with 4 digits after the decimal point and an exponent that is a multiple of 3.

Set the output format to the long engineering format and view the same values.

```
format longEng
```

```
A = 5.123456789;
for k = 1:10
   disp(A)
   A = A*10;
end
```

```
    5.12345678900000e+000
```

```
   51.2345678900000e+000

   512.345678900000e+000

   5.12345678900000e+003

   51.2345678900000e+003

   512.345678900000e+003

   5.12345678900000e+006

   51.2345678900000e+006

   512.345678900000e+006

   5.12345678900000e+009
```

The values display with 15 digits and an exponent that is a multiple of 3.

Large Data Range Format

Use the `shortG` format when some of the values in an array are short numbers and some have large exponents. The `shortG` format picks whichever of `short` or `shortE` has the most compact display.

Create a variable and display output in the `short` format, which is the default.

```
x = [25 56 255 9876899999];
format short
x
```

```
x =

   1.0e+09 *

    0.0000    0.0000    0.0000    9.8769
```

Set the format to `shortG` and redisplay the values.

```
format shortG
```

```
x
```

```
x =

          25          56         255   9.8769e+09
```

Get Current Format

Get the current numeric format.

```
f = get(0,'Format')
```

```
f =

shortG
```

Get the current line spacing, which can be set to `loose` or `compact`.

```
S = get(0,'FormatSpacing')
```

```
S =

loose
```

- "Format Output in Command Window"

Input Arguments

`style` — Output display format

`short` (default) | `long` | `shortE` | `longE` | ...

Output display format, specified by a string from the tables below.

Numeric Format

These styles control the output display format for numeric variables.

Style	Result	Example
`short` (default)	Short, fixed-decimal format with 4 digits after the decimal point.	`3.1416`

Style	Result	Example
`long`	Long, fixed-decimal format with 15 digits after the decimal point for `double` values, and 7 digits after the decimal point for `single` values.	`3.141592653589793`
`shortE`	Short scientific notation with 4 digits after the decimal point.	`3.1416e+00`
`longE`	Long scientific notation with 15 digits after the decimal point for `double` values, and 7 digits after the decimal point for `single` values.	`3.141592653589793e+00`
`shortG`	`short` or `shortE`, whichever is more compact.	`3.1416`
`longG`	`long` or `longE`, whichever is more compact.	`3.14159265358979`
`shortEng`	Short engineering notation (exponent is a multiple of 3) with 4 digits after the decimal point.	`3.1416e+000`
`longEng`	Long engineering notation (exponent is a multiple of 3) with 15 significant digits.	`3.14159265358979e+000`
`+`	Positive/Negative format with `+`, `-`, and blank characters displayed for positive, negative, and zero elements.	`+`
`bank`	Currency format with 2 digits after the decimal point.	`3.14`
`hex`	Hexadecimal representation of a binary double-precision number.	`400921fb54442d18`
`rat`	Ratio of small integers.	`355/113`

Line Spacing Format

Style	Result	Example

`compact`	Suppress excess blank lines to show more output on a single screen.	`theta = pi/2`
		`theta =`
		`  1.5708`

Style	Result	Example

`loose`	Add blank lines to make output more readable.	`theta = pi/2`
		`theta =`
		`  1.5708`

More About

Tips

- The specified format applies only to the current MATLAB session. To maintain a format across sessions, choose a **Numeric format** or **Numeric display** option in the Command Window Preferences.
- You can insert a space between `short` or `long` and the presentation type, for instance, `format short E`.
- MATLAB always displays integer data types to the appropriate number of digits for the data type. For example, MATLAB uses 3 digits to display `int8` data types (for instance, -128:127). Setting the output format to `short` or `long` does not affect the display of integer-type variables.
- Integer-valued, floating-point numbers with a maximum of 9 digits do not display in scientific notation.
- If you are displaying a matrix with a wide range of values, consider using `shortG`. See “Large Data Range Format” on page 1-2758.

See Also

`disp` | `fprintf` | `rat`

Introduced before R2006a

fplot

Plot function between specified limits

Syntax

```
fplot(fun,limits)
fplot(fun,limits,LineSpec)
fplot(fun,limits,tol)
fplot(fun,limits,tol,LineSpec)
fplot(fun,limits,n)
fplot(fun,lims,...)
fplot(axes_handle,...)
[X,Y] = fplot(fun,limits,...)
```

Description

`fplot` plots a function between specified limits. The function must be of the form $y = f(x)$, where x is a vector whose range specifies the limits, and `y` is a vector the same size as `x` and contains the function's value at the points in x (see the first example). If the function returns more than one value for a given x, then y is a matrix whose columns contain each component of $f(x)$ (see the second example).

`fplot(fun,limits)` plots `fun` between the limits specified by `limits`. `limits` is a vector specifying the x-axis limits (`[xmin xmax]`), or the x- and y-axes limits, (`[xmin xmax ymin ymax]`).

`fun` must be

- The name of a function
- A string with variable `x` that may be passed to `eval`, such as `'sin(x)'`, `'diric(x,10)'`, or `'[sin(x),cos(x)]'`
- A function handle

The function $f(x)$ must return a row vector for each element of vector x. For example, if $f(x)$ returns `[f1(x),f2(x),f3(x)]` then for input `[x1;x2]` the function should return the matrix

```
f1(x1) f2(x1) f3(x1)
f1(x2) f2(x2) f3(x2)
```

`fplot(fun,limits,LineSpec)` plots `fun` using the line specification `LineSpec`.

`fplot(fun,limits,tol)` plots `fun` using the relative error tolerance `tol` (the default is `2e-3`, i.e., 0.2 percent accuracy).

`fplot(fun,limits,tol,LineSpec)` plots `fun` using the relative error tolerance `tol` and a line specification that determines line type, marker symbol, and color. See `LineSpec` for more information.

`fplot(fun,limits,n)` with `n >= 1` plots the function with a minimum of `n+1` points. The default `n` is `1`. The maximum step size is restricted to be `(1/n)*(xmax-xmin)`.

`fplot(fun,lims,...)` accepts combinations of the optional arguments `tol`, `n`, and `LineSpec`, in any order.

`fplot(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

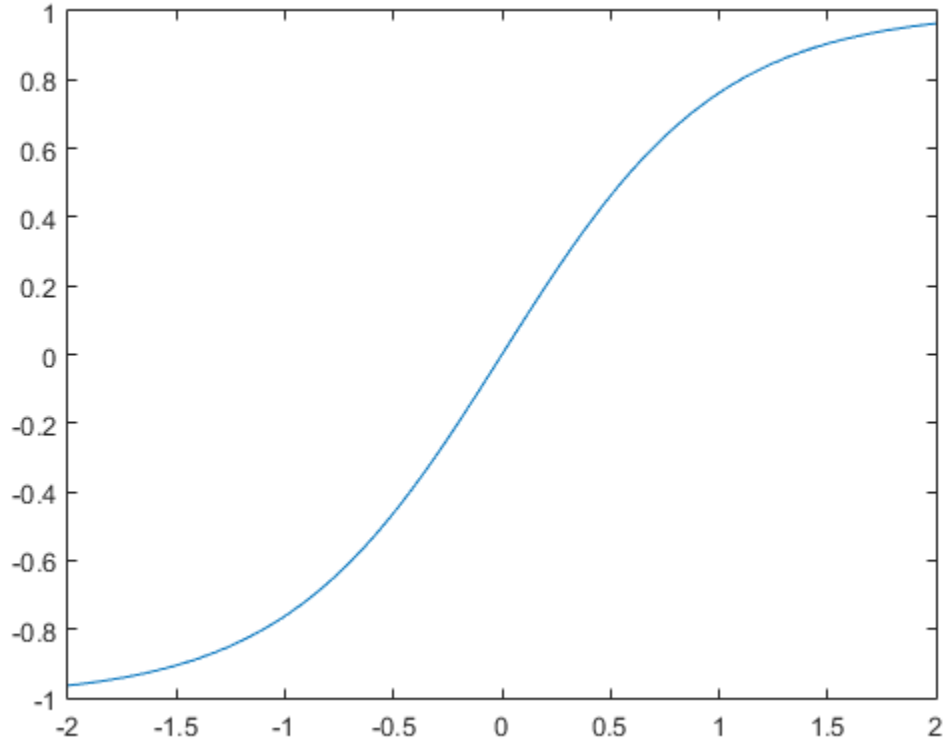
`[X,Y] = fplot(fun,limits,...)` returns the abscissas and ordinates for `fun` in `X` and `Y`. No plot is drawn on the screen; however, you can plot the function using `plot(X,Y)`.

Examples

MATLAB® Function Handle

Plot the hyperbolic tangent function from -2 to 2 using the MATLAB® function `tanh`.

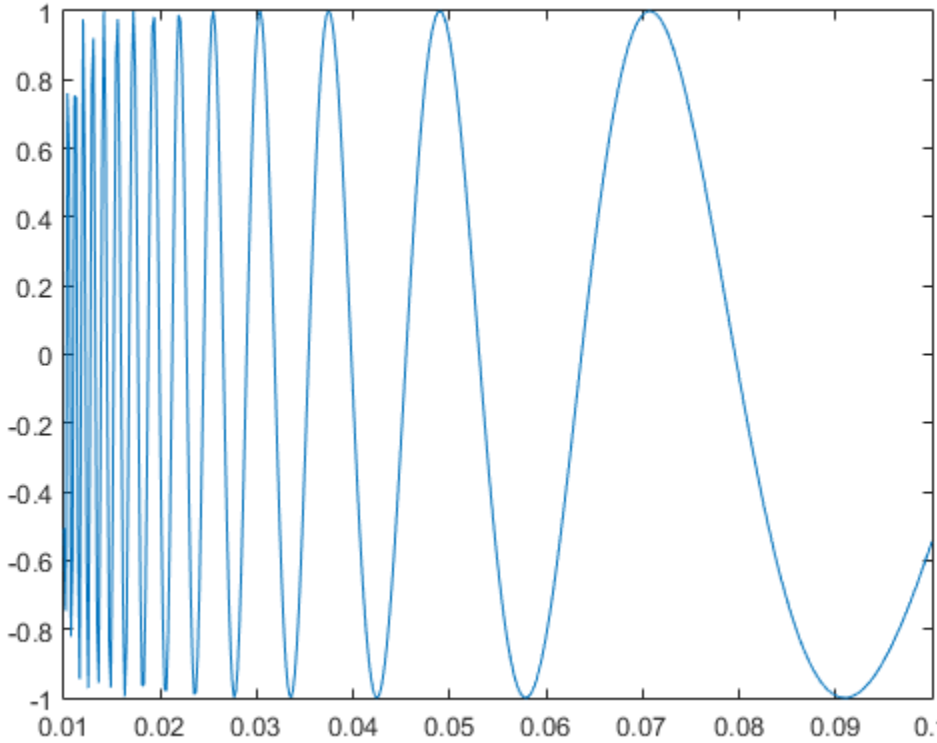
```
fh = @tanh;
fplot(fh,[-2,2])
```



Function Handle Created From Anonymous Function

Create a function handle from an anonymous function. Plot the function from 0.01 to 0.1.

```
sn = @(x) sin(1./x);
fplot(sn,[0.01,0.1])
```



Function Handle Created From Custom Function File

Create a file named `myfun.m` that contains the following code.

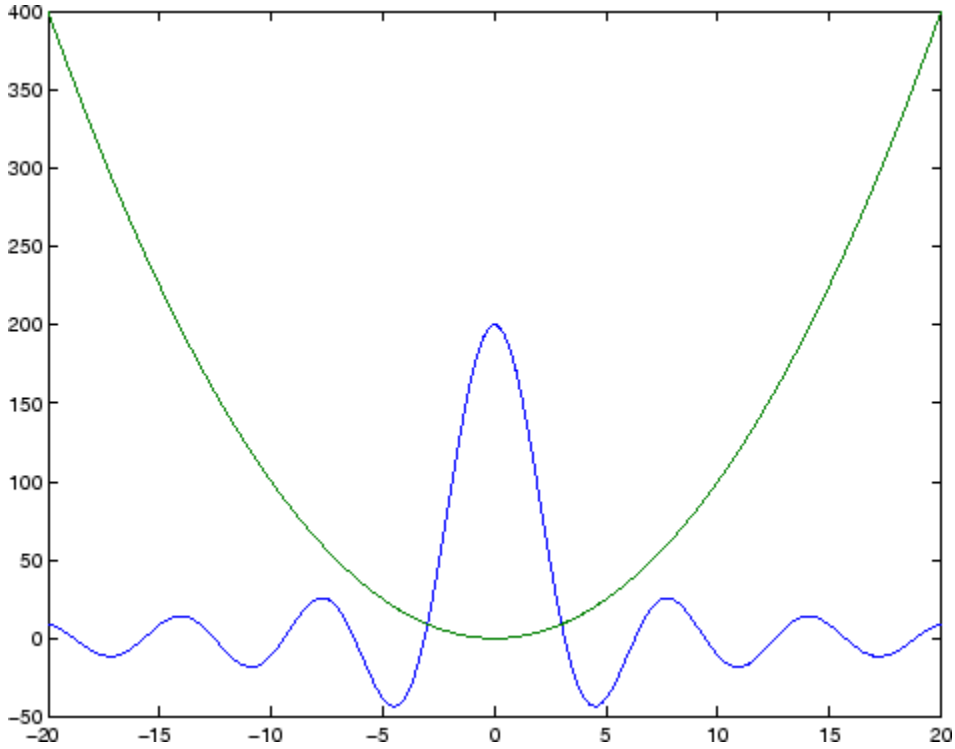
```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Then, create a function handle pointing to `myfun`.

```
fh = @myfun;
```

Plot the function from -20 to 20.

```
fplot(fh,[-20 20])
```



More About

Tips

`fplot` uses adaptive step control to produce a representative graph, concentrating its evaluation in regions where the function's rate of change is the greatest.

- Anonymous Functions

See Also

`eval` | `ezplot` | `feval` | `LineSpec` | `plot`

Introduced before R2006a

fprintf

Write data to text file

Syntax

```
fprintf(fileID,formatSpec,A1,...,An)
fprintf(formatSpec,A1,...,An)

nbytes = fprintf( ___ )
```

Description

`fprintf(fileID,formatSpec,A1,...,An)` applies the `formatSpec` to all elements of arrays `A1,...An` in column order, and writes the data to a text file. `fprintf` uses the encoding scheme specified in the call to `fopen`.

`fprintf(formatSpec,A1,...,An)` formats data and displays the results on the screen.

`nbytes = fprintf( ___ )` returns the number of bytes that `fprintf` writes, using any of the input arguments in the preceding syntaxes.

Examples

Print Literal Text and Array Values

Print multiple numeric values and literal text to the screen.

```
A1 = [9.9, 9900];
A2 = [8.8,  7.7 ; ...
      8800, 7700];
formatSpec = 'X is %4.2f meters or %8.3f mm\n';
fprintf(formatSpec,A1,A2)
```

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

`%4.2f` in the `formatSpec` input specifies that the first value in each line of output is a floating-point number with a field width of four digits, including two digits after the decimal point. `%8.3f` in the `formatSpec` input specifies that the second value in each line of output is a floating-point number with a field width of eight digits, including three digits after the decimal point. `\n` is a control character that starts a new line.

Print Double-Precision Values as Integers

Explicitly convert double-precision values with fractions to integer values.

```
a = [1.02, 3.04, 5.06];
fprintf('%d\n',round(a));
```

```
1
3
5
```

`%d` in the `formatSpec` input prints each value in the vector, `round(a)`, as a signed integer. `\n` is a control character that starts a new line.

Write Tabular Data to Text File

Write a short table of the exponential function to a text file called `exp.txt`.

```
x = 0:.1:1;
A = [x; exp(x)];

fileID = fopen('exp.txt','w');
fprintf(fileID,'%6s %12s\n','x','exp(x)');
fprintf(fileID,'%6.2f %12.8f\n',A);
fclose(fileID);
```

The first call to `fprintf` prints header text `x` and `exp(x)`, and the second call prints the values from variable `A`.

If you plan to read the file with Microsoft Notepad, use `'\r\n'` instead of `'\n'` to move to a new line. For example, replace the calls to `fprintf` with the following:

```
fprintf(fileID,'%6s %12s\r\n','x','exp(x)');
fprintf(fileID,'%6.2f %12.8f\r\n',A);
```

MATLAB import functions, all UNIX applications, and Microsoft Word and WordPad recognize `'\n'` as a newline indicator.

View the contents of the file with the `type` command.

```
type exp.txt
```

```
     x       exp(x)
  0.00   1.00000000
  0.10   1.10517092
  0.20   1.22140276
  0.30   1.34985881
  0.40   1.49182470
  0.50   1.64872127
  0.60   1.82211880
  0.70   2.01375271
  0.80   2.22554093
  0.90   2.45960311
  1.00   2.71828183
```

Get Number of Bytes Written to File

Write data to a file and return the number of bytes written.

Write an array of data, `A`, to a file and get the number of bytes that `fprintf` writes.

```
A = magic(4);

fileID = fopen('myfile.txt','w');
nbytes = fprintf(fileID,'%5d %5d %5d %5d\n',A)
```

```
nbytes =

    96
```

The `fprintf` function wrote 96 bytes to the file.

Close the file.

```
fclose(fileID);
```

View the contents of the file with the `type` command.

```
type('myfile.txt')
```

```
   16     5     9     4
    2    11     7    14
    3    10     6    15
```

```
    13     8    12     1
```

Display Hyperlinks in Command Window

Display a hyperlink (The MathWorks Web Site) on the screen.

```
site = 'http://www.mathworks.com';
title = 'The MathWorks Web Site';

fprintf('<a href = "%s">%s</a>\n',site,title)
```

`%s` in the `formatSpec` input indicates that the values of the variables `site` and `title`, should be printed as strings.

- "Export Cell Array to Text File"
- "Appending or Overwriting Existing Files"

Input Arguments

`fileID` — File identifier

1 (default) | 2 | scalar

File identifier, specified as one of the following:

- A file identifier obtained from `fopen`.
- `1` for standard output (the screen).
- `2` for standard error.

Data Types: `double`

`formatSpec` — Format of output fields

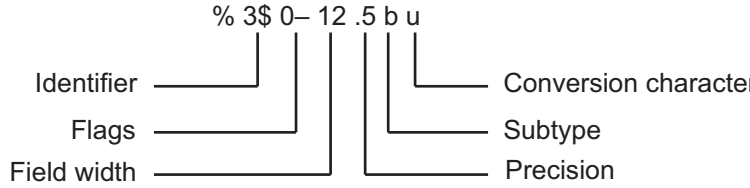
string containing formatting operators

Format of the output fields, specified as a string containing formatting operators. `formatSpec` also can include ordinary text and special characters.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier,

flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).



Conversion Character

This table shows conversion characters to format numeric and character data as strings.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a–f`
	`%X`	Same as `%x`, uppercase letters `A–F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)

Value Type	Conversion	Details
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters	`%c`	Single character
	`%s`	String of characters

Optional Operators

The optional identifier, flags, field width, precision, and subtype operators further define the format of the output string.

- **Identifier**

  Order for processing values from the input list. Use the syntax *n*`$`, where *n* represents the position of the value in the input list.

  Example: `'%3$s %2$s %1$s %2$s'` prints inputs `'A'`, `'B'`, `'C'` as follows: `C B A B`.

- **Flags**

  `'–'` Left-justify.
  Example: `%-5.2f`

  `'+'` Always print a sign character (+ or –) for any value.
  Example: `%+5.2f`

  `' '` Insert a space before the value.
  Example: `% 5.2f`

  `'0'` Pad to field width with zeros before the value.
  Example: `%05.2f`

  `'#'` Modify selected numeric conversions:

  - For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix.
  - For `%f`, `%e`, or `%E`, print decimal point even when precision is 0.
  - For `%g` or `%G`, do not remove trailing zeros or decimal point.

  Example: `%#5.0f`

- **Field Width**

Minimum number of characters to print. The field width operator can be a number, or an asterisk (`*`) to refer to an argument in the input list.

Example: The input list `('%12d',intmax)` is equivalent to `('%*d', 12, intmax)`.

The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `' 3.142'`

The precision operator can be a number, or an asterisk (`*`) to refer to an argument in the input list.

Example: The input list `('%6.4f', pi)` is equivalent to `('%*.*f', 6, 4, pi)`.

Note: If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

- **Subtypes**

Certain conversion characters can support a subtype. The subtype operator immediately precedes the conversion character. This table shows the conversions that can use subtypes.

Input Value Type	Subtype and Conversion Character	Output Value Type

Floating-point number	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

Input Value Type	Subtype and Conversion Character	Output Value Type

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Integer	`%ld` or `%li`
	`%lo`
	`%lu`
	`%lx` or `%lX`
		64-bit value

Integer	`%hd` or `%hi`
	`%ho`
	`%hu`
	`%hx` or `%hX`
		16-bit value

Text Before or After Formatting Operators

`formatSpec` can also include additional text before a percent sign, `%`, or after a conversion character. The text can be:

- Ordinary text to print.
- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in `formatSpec`.

Special Character	Representation
Single quotation mark	`''`
Percent character	`%%`
Backslash	`\\`
Alarm	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`

Special Character	Representation
Vertical tab	`\v`
Character whose ASCII code is the hexadecimal number, `N`	`\xN`
Character whose ASCII code is the octal number, `N`	`\N`

Notable Behavior of Conversions with Formatting Operators

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a string conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.

  Example: `'%s'` converts `pi` to `3.141593e+00`.
- If you apply a string conversion (`%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters.

  Example: `'%s'` converts `[65 66 67]` to `ABC`.

`A1,...,An` — Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Output Arguments

`nbytes` — Number of bytes

scalar

Number of bytes that `fprintf` writes, returned as a scalar. When writing to a file, `nbytes` is determined by the character encoding. When printing data to the screen, `nbytes` is the number of characters displayed on the screen.

More About

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.
- "Formatting Strings"

References

[1] Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: "Programming Language C," ANSI, 1430 Broadway, New York, NY 10018.

See Also

`disp` | `fclose` | `ferror` | `fopen` | `fread` | `fscanf` | `fseek` | `ftell` | `fwrite` | `sprintf`

Introduced before R2006a

fprintf (serial)

Write text to device

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the device connected to the serial port object, `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command line until execution completes.

`fprintf(obj,'format','cmd')` writes the string using the format specified by *`format`*.

`fprintf(obj,'cmd','mode')` writes the string with command-line access specified by *`mode`*. *`mode`* specifies if `cmd` is written synchronously or asynchronously.

`fprintf(obj,'format','cmd','mode')` writes the string using the specified format. *`format`* is a C language conversion specification.

You need an open connection from the serial port object, `obj`, to the device before performing read or write operations.

Use the `fopen` function to open a connection to the device. When `obj` has an open connection to the device, it has a `Status` property value of `open`. Refer to "Troubleshooting Common Errors" for `fprintf` errors.

To understand the use of `fprintf` refer to "Completing a Write Operation with fprintf" and "Rules for Writing the Terminator".

Input Arguments

`format`

ANSI C conversion specification includes these conversion characters.

Specifier	Description
`%c`	Single character
`%d` or `%i`	Decimal notation (signed)
`%e`	Exponential notation (using lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using uppercase `E` as in `3.1415E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`. Insignificant zeros do not print.
`%G`	Same as `%g`, but using uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`%X`	Hexadecimal notation (using uppercase letters `A`–`F`)

`mode`

Specifies whether the string `cmd` is written synchronously or asynchronously:

- `sync`: `cmd` is written synchronously and the command line is blocked.
- `async`: `cmd` is written asynchronously and the command line is not blocked.

If *`mode`* is not specified, the write operation is synchronous.

If you specify asynchronous *`mode`*, when the write operation occurs:

- The `BytesToOutput` property value continuously updates to reflect the number of bytes in the output buffer.
- The MATLAB file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

Use the `TransferStatus` property to determine whether an asynchronous write operation is in progress.

For more information on synchronous and asynchronous write operations, see Controlling Access to the MATLAB Command Line.

Examples

Create a serial port object `s` and connect it to a Tektronix TDS 210 oscilloscope. Write the `RS232?` command with `fprintf`. `RS232?` instructs the scope to return serial port communications settings. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
s = serial('COM1');
fopen(s)
fprintf(s,'%s','RS232?')
```

The default format for `fprintf` is `%s\n`. Therefore, the terminator specified by the `Terminator` property is automatically written. However, in some cases you might want to suppress writing the terminator.

Specify an array of formats and commands:

```
s = serial('COM1');
fopen(s)
fprintf(s,['ch:%d scale:%d'],[1 20e-3],'sync');
```

See Also

`fopen` | `fwrite` | `stopasync` | `BytesToOutput` | `OutputBufferSize` | `OutputEmptyFcn` | `Status` | `TransferStatus` | `ValuesSent`

Introduced before R2006a

frame2im

Return image data associated with movie frame

Syntax

```
[X,Map] = frame2im(F)
```

Description

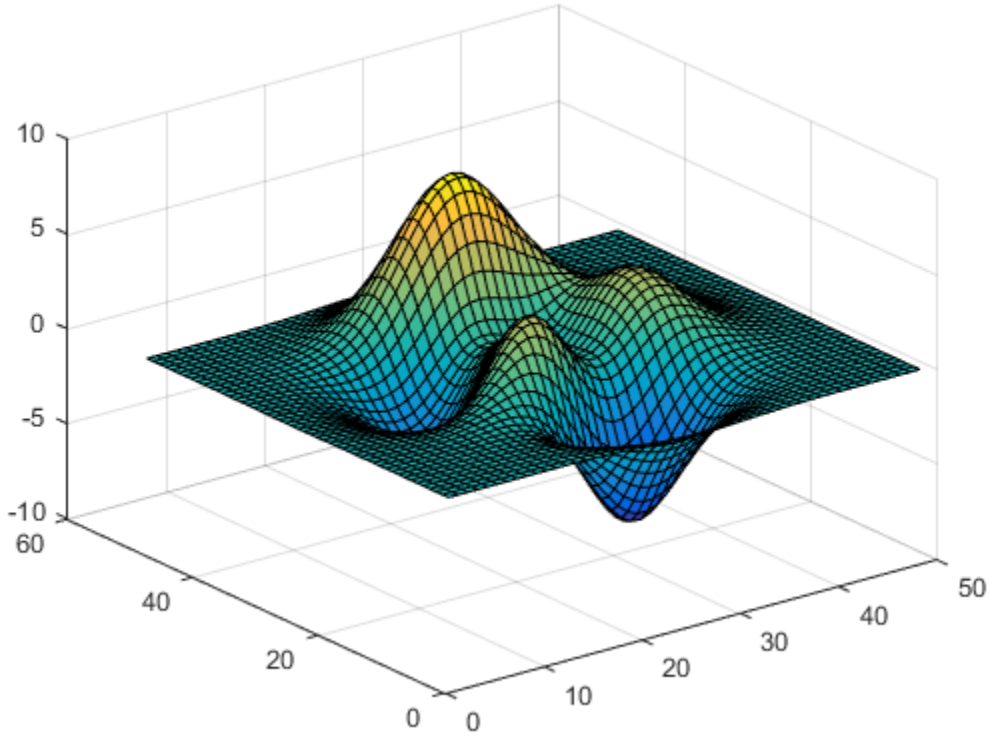
`[X,Map] = frame2im(F)` returns the indexed image data `X` and associated colormap `Map` from the single movie frame `F`. The output `Map` is a three-column matrix where each row of the matrix is an RGB triplet value that defines one color of the colormap. If the frame contains true-color data, then `Map` is empty. The functions `getframe` and `im2frame` create a movie frame.

Examples

Convert Movie Frame to Image Data

Create a surface plot.

```
surf(peaks)
```



Use `getframe` to capture the plot as a movie frame. Then, convert the captured movie frame to image data.

```
F = getframe;
[X,map] = frame2im(F);
```

See Also

`movie` | `getframe` | `im2frame` | `ind2rgb`

Introduced before R2006a

fread

Read data from binary file

Syntax

```
A = fread(fileID)
A = fread(fileID,sizeA)
A = fread(fileID,sizeA,precision)
A = fread(fileID,sizeA,precision,skip)
A = fread(fileID,sizeA,precision,skip,machinefmt)
[A,count] = fread( ___ )
```

Description

`A = fread(fileID)` reads data from an open binary file into column vector `A` and positions the file pointer at the end-of-file marker. The binary file is indicated by the file identifier, `fileID`. Use `fopen` to open the file and obtain the `fileID` value. When you finish reading, close the file by calling `fclose(fileID)`.

`A = fread(fileID,sizeA)` reads file data into an array, `A`, with dimensions, `sizeA`, and positions the file pointer after the last value read. `fread` populates `A` in column order.

`A = fread(fileID,sizeA,precision)` interprets values in the file according to the form and size described by `precision`. The `sizeA` argument is optional.

`A = fread(fileID,sizeA,precision,skip)` skips the number of bytes or bits specified by `skip` after reading each value in the file. The `sizeA` argument is optional.

`A = fread(fileID,sizeA,precision,skip,machinefmt)` additionally specifies the order for reading bytes or bits in the file. The `sizeA` and `skip` arguments are optional.

`[A,count] = fread( ___ )` additionally returns the number of characters that `fread` reads into `A`. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Read Entire File of uint8 Data

Write a nine-element vector to a sample file, `nine.bin`.

```
fileID = fopen('nine.bin','w');
fwrite(fileID,[1:9]);
fclose(fileID);
```

Read all the data in the file into a vector of class `double`. By default, `fread` reads a file 1 byte at a time, interprets each byte as an 8-bit unsigned integer (`uint8`), and returns a `double` array.

```
fileID = fopen('nine.bin');
A = fread(fileID)
```

```
A =

     1
     2
     3
     4
     5
     6
     7
     8
     9
```

`fread` returns a column vector, with one element for each byte in the file.

View information about `A`.

```
whos A
```

```
  Name      Size            Bytes  Class     Attributes

  A         9x1                72  double
```

Close the file.

```
fclose(fileID);
```

Read Entire File of Double-Precision Data

Create a file named `doubledata.bin`, containing nine double-precision values.

```
fileID = fopen('doubledata.bin','w');
fwrite(fileID,magic(3),'double');
fclose(fileID);
```

Open the file, `doubledata.bin`, and read the data in the file into a 3-by-3 array, `A`. Specify that the source data is class `double`.

```
fileID = fopen('doubledata.bin');
A = fread(fileID,[3 3],'double')
```

```
A =

     8     1     6
     3     5     7
     4     9     2
```

Close the file.

```
fclose(fileID);
```

Read Text File

Read the contents of the file, `fread.m`. Transpose the output array, `A` so that it is a row vector.

```
fileID = fopen('fread.m');
A = fread(fileID,'*char')';
fclose(fileID);
```

`fread` returns the character array, `A`.

Read Selected Rows or Columns from File

Create a file named `nine.bin`, containing the values from 1 to 9. Write the data as `uint16` values.

```
fileID = fopen('nine.bin','w');
fwrite(fileID,[1:9],'uint16');
```

```
fclose(fileID);
```

Read the first six values into a 3-by-2 array. Specify that the source data is class `uint16`.

```
fileID = fopen('nine.bin');
A = fread(fileID,[3,2],'uint16')
```

```
A =

     1     4
     2     5
     3     6
```

`fread` returns an array populated column-wise with the first six values from the file, `nine.bin`.

Return to the beginning of the file.

```
frewind(fileID)
```

Read two values at a time, and skip one value before reading the next values. Specify this format using the `precision` value, `'2*uint16'`. Because the data is class `uint16`, one value is represented by 2 bytes. Therefore, specify the `skip` argument as `2`.

```
precision = '2*uint16';
skip = 2;
B = fread(fileID,[2,3],precision,skip)
```

```
B =

     1     4     7
     2     5     8
```

`fread` returns a 2-by-3 array populated column-wise with the values from `nine.bin`.

Close the file.

```
fclose(fileID);
```

Read Digits of Binary Coded Decimal Values

Create a file with binary coded decimal (BCD) values.

```
str = ['AB'; 'CD'; 'EF'; 'FA'];

fileID = fopen('bcd.bin','w');
fwrite(fileID,hex2dec(str),'ubit8');
fclose(fileID);
```

Read 1 byte at a time.

```
fileID = fopen('bcd.bin');
onebyte = fread(fileID,4,'*ubit8');
```

Display the BCD values.

```
disp(dec2hex(onebyte))
```

```
AB
CD
EF
FA
```

Return to the beginning of the file using `frewind`. If you read 4 bits at a time on a little-endian system, your results appear in the wrong order.

```
frewind(fileID)

err = fread(fileID,8,'*ubit4');
disp(dec2hex(err))
```

```
B
A
D
C
F
E
A
F
```

Return to the beginning of the file using `frewind`. Read the data 4 bits at a time as before, but specify a big-endian ordering to display the correct results.

```
frewind(fileID)

correct = fread(fileID,8,'*ubit4','ieee-be');
disp(dec2hex(correct))
```

```
A
B
C
D
E
F
F
A
```

Close the file.

```
fclose(fileID);
```

Input Arguments

`fileID` — File identifier

integer

File identifier of an open binary file, specified as an integer. Before reading a file with `fread`, you must use `fopen` to open the file and obtain the `fileID`.

Data Types: `double`

`sizeA` — Dimensions of output array

`Inf` (default) | integer | two-element row vector

Dimensions of the output array, `A`, specified as `Inf`, an integer, or a two-element row vector.

Form of the `sizeA` Input	Dimensions of the output array, `A`
`Inf`	Column vector, with each element containing a value in the file.
`n`	Column vector with `n` elements.
`[m,n]`	`m`-by-`n` matrix, filled in column order. `n` can be `Inf`, but `m` cannot.

`precision` — Class and size of values to read

`'uint8=>double'` (default) | string

Class and size in bits of the values to read, specified as a string in one of the following forms. Optionally the input specifies the class of the output matrix, A.

Form of the `precision` Input	Description
`source`	Input values are of the class specified by *`source`*. Output matrix A is class `double`. Example: `'int16'`
`source=>output`	Input values are of the class specified by *`source`*. The class of the output matrix, A, is specified by *`output`*. Example: `'int8=>char'`
`*source`	The input values and the output matrix, A, are of the class specified by *`source`*. For `bit`*n* or `ubit`*n* precisions, the output has the smallest class that can contain the input. Example: `'*ubit18'` This is equivalent to `'ubit18=>uint32'`
`N*source` or *`N*source=>output`*	Read *N* values before skipping the number of bytes specified by the `skip` argument. Example: `'4*int8'`

The following table shows possible values for *`source`* and *`output`*.

Value Type	Precision	Bits (Bytes)
Integers, unsigned	`uint`	`32 (4)`
	`uint8`	`8 (1)`
	`uint16`	`16 (2)`
	`uint32`	`32 (4)`
	`uint64`	`64 (8)`
	`uchar`	`8 (1)`
	`unsigned char`	`8 (1)`
	`ushort`	`16 (2)`
	`ulong`	`32 (4)`
	`ubit`*n*	$1 \le n \le 64$

Value Type	Precision	Bits (Bytes)
Integers, signed	`int`	32 (4)
	`int8`	8 (1)
	`int16`	16 (2)
	`int32`	32 (4)
	`int64`	64 (8)
	`integer*1`	8 (1)
	`integer*2`	16 (2)
	`integer*4`	32 (4)
	`integer*8`	64 (8)
	`schar`	8 (1)
	`signed char`	8 (1)
	`short`	16 (2)
	`long`	32 (4)
	`bit`*`n`	$1 \le n \le 64$
Floating-point numbers	`single`	32 (4)
	`double`	64 (8)
	`float`	32 (4)
	`float32`	32 (4)
	`float64`	64 (8)
	`real*4`	32 (4)
	`real*8`	64 (8)
Characters	`char*1`	8 (1)
	`char`	Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

For most values of *`source`*, if `fread` reaches the end of the file before reading a complete value, it does not return a result for the final value. However, if *`source`* is `bit`*`n`* or `ubit`*`n`*, then `fread` returns a partial result for the final value.

Note: To preserve `NaN` and `Inf` values in MATLAB, read and write data of class `double` or `single`.

`skip` — Number of bytes to skip

0 (default) | scalar

Number of bytes to skip after reading each value, specified as a scalar. If you specify a `precision` of `bit`*n* or `ubit`*n*, specify `skip` in bits.

Use the `skip` argument to read data from noncontiguous fields in fixed-length records.

`machinefmt` — Order for reading bytes

`'n'` (default) | `'b'` | `'l'` | `'s'` | `'a'` | ...

Order for reading bytes in the file, specified as one of the strings in the table that follows. For `bit`*n* and `ubit`*n* precisions, `machinefmt` specifies the order for reading bits within a byte, but the order for reading bytes remains your system byte ordering.

`'n'` or `'native'`	Your system byte ordering (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big-endian or little-endian ordering.

Output Arguments

`A` — File data

column vector | matrix

File data, returned as a column vector. If you specified the `sizeA` argument, then `A` is a matrix of the specified size. Data in `A` is class `double` unless you specify a different class in the `precision` argument.

`count` — Number of characters read

scalar

Number of characters read, returned as a scalar value.

More About

- "Reading Portions of a File"
- "Reading Files Created on Other Systems"

See Also

`fclose` | `fgetl` | `fopen` | `fprintf` | `fscanf` | `fseek` | `ftell` | `fwrite`

Introduced before R2006a

fread (serial)

Read binary data from device

Syntax

```
A = fread(obj)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

Description

`A = fread(obj)` and `A = fread(obj,size)` read binary data from the device connected to the serial port object, `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *`precision`* (see below).

`A = fread(obj,size,'precision')` reads binary data with precision specified by *`precision`*.

`precision` controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *`precision`* are listed below in Tips.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.

More About

Tips

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB command line until:

- The specified number of values are read.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used for binary read operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer

Data Type	Precision	Interpretation
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

`fgetl` | `fopen` | `fscanf` | `fgets` | `BytesAvailable` | `BytesAvailableFcn` | `InputBufferSize` | `Status` | `Terminator` | `ValuesReceived`

Introduced before R2006a

freeBoundary

Class: TriRep

(Will be removed) Facets referenced by only one simplex

Note: `freeBoundary(TriRep)` will be removed in a future release. Use `freeBoundary(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
FF = freeBoundary(TR)
[FF XF] = freeBoundary(TR)
```

Description

`FF = freeBoundary(TR)` returns a matrix `FF` that represents the free boundary facets of the triangulation. A facet is on the free boundary if it is referenced by only one simplex (triangle/tetrahedron, etc). `FF` is of size `m`-by-`n`, where m is the number of boundary facets and n is the number of vertices per facet. The vertices of the facets index into the array of points representing the vertex coordinates `TR.X`. The array `FF` could be empty as in the case of a triangular mesh representing the surface of a sphere.

`[FF XF] = freeBoundary(TR)` returns a matrix of free boundary facets

Input Arguments

`TR`	Triangulation representation.

Output Arguments

`FF`	`FF` that has vertices defined in terms of a compact array of coordinates `XF`.

`XF`	`XF` is of size `m`-by-`ndim` where `m` is the number of free facets, and `ndim` is the dimension of the space where the triangulation resides

Definitions

A *simplex* is a triangle/tetrahedron or higher-dimensional equivalent. A *facet* is an edge of a triangle or a face of a tetrahedron.

Examples

Example 1

Use `TriRep` to compute the boundary triangulation of an imported triangulation.

Load a 3-D triangulation:

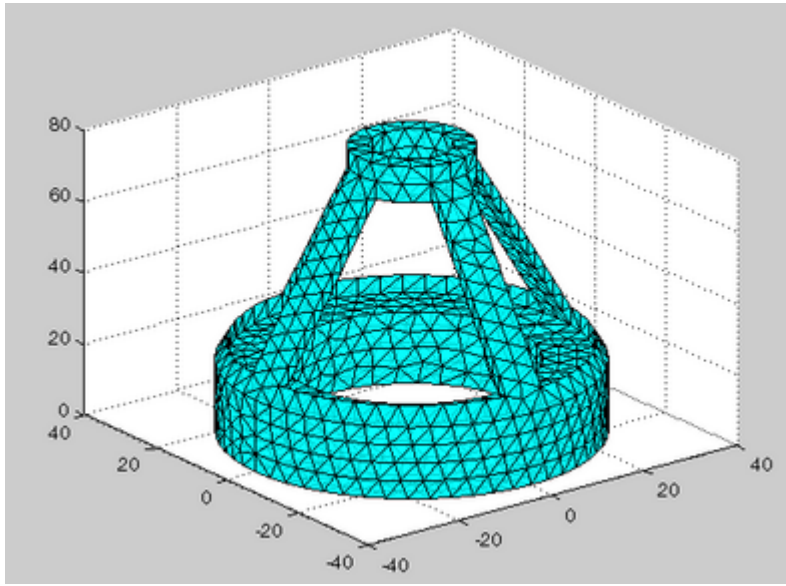
```
load tetmesh;
trep = TriRep(tet, X);
```

Compute the boundary triangulation:

```
 [tri xf] = freeBoundary(trep);
```

Plot the boundary triangulation:

```
trisurf(tri, xf(:,1),xf(:,2),xf(:,3), ...
     'FaceColor','cyan', 'FaceAlpha', 0.8);
```



Example 2

Perform a direct query of a 2-D triangulation created with `DelaunayTri`.

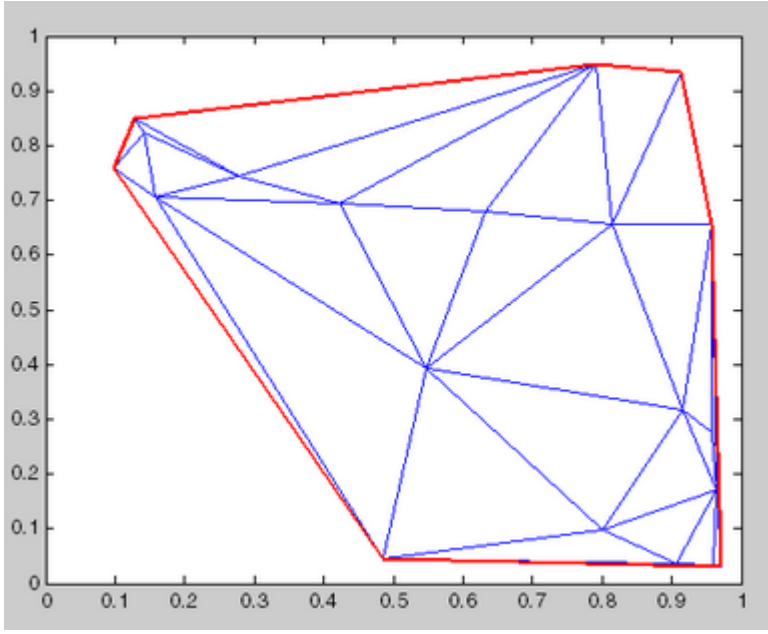
Plot the mesh:

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
fe = freeBoundary(dt)';
triplot(dt);
hold on;
```

Display the free boundary edges in red:

```
plot(x(fe), y(fe), '-r', 'LineWidth',2) ;
hold off;
```

In this instance the free edges correspond to the convex hull of `(x, y)`.



See Also

`convexHull` | `featureEdges` | `triangulation` | `delaunayTriangulation` | `faceNormal`

freqspace

Frequency spacing for frequency response

Syntax

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for various one- and two-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`f = freqspace(N)` returns the one-dimensional frequency vector `f` assuming `N` evenly spaced points around the unit circle. For `N` even or odd, `f` is `(0:2/N:1)`. For `N` even, `freqspace` therefore returns `(N+2)/2` points. For `N` odd, it returns `(N+1)/2` points.

`f = freqspace(N,'whole')` returns `N` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/N:2*(N-1)/N`.

See Also

`meshgrid`

Introduced before R2006a

frewind

Move file position indicator to beginning of open file

Syntax

```
frewind(fileID)
```

Description

`frewind(fileID)` sets the file position indicator to the beginning of a file. *`fileID`* is an integer file identifier obtained from `fopen`.

If the file is on a tape device and the rewind operation fails, `frewind` does not return an error message.

Alternatives

`frewind(fileID)` is equivalent to:

```
fseek(fileID, 0, 'bof');
```

See Also

`fclose` | `feof` | `ferror` | `fopen` | `fseek` | `ftell` | `fscanf` | `fprintf` | `fread` | `fwrite`

Introduced before R2006a

fscanf

Read data from text file

Syntax

```
A = fscanf(fileID,formatSpec)
A = fscanf(fileID,formatSpec,sizeA)
[A,count] = fscanf( ___ )
```

Description

`A = fscanf(fileID,formatSpec)` reads data from an open text file into column vector `A` and interprets values in the file according to the format specified by `formatSpec`. The `fscanf` function reapplies the format throughout the entire file and positions the file pointer at the end-of-file marker. If `fscanf` cannot match `formatSpec` to the data, it reads only the portion that matches and stops processing.

The text file is indicated by the file identifier, `fileID`. Use `fopen` to open the file, specify the character encoding, and obtain the `fileID` value. When you finish reading, close the file by calling `fclose(fileID)`.

`A = fscanf(fileID,formatSpec,sizeA)` reads file data into an array, `A`, with dimensions, `sizeA`, and positions the file pointer after the last value read. `fscanf` populates `A` in column order.

`[A,count] = fscanf( ___ )` additionally returns the number of fields that `fscanf` reads into `A`. For numeric data, this is the number of values read. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Read File Contents into Column Vector

Create a sample text file that contains floating-point numbers.

```
x = 100*rand(8,1);
```

```
fileID = fopen('nums1.txt','w');
fprintf(fileID,'%4.4f\n',x);
fclose(fileID);
```

View the contents of the file.

```
type nums1.txt
```

```
81.4724
90.5792
12.6987
91.3376
63.2359
9.7540
27.8498
54.6882
```

Open the file for reading, and obtain the file identifier, `fileID`.

```
fileID = fopen('nums1.txt','r');
```

Define the format of the data to read. Use the string, `'%f'`, to specify floating-point numbers.

```
formatSpec = '%f';
```

Read the file data, filling output array, `A`, in column order. `fscanf` reapplies the format, `formatSpec`, throughout the file.

```
A = fscanf(fileID,formatSpec)
```

```
A =

   81.4724
   90.5792
   12.6987
   91.3376
   63.2359
    9.7540
   27.8498
   54.6882
```

`A` is a column vector containing data from the file.

Close the file.

```
fclose(fileID);
```

Read File Contents into Array

Create a sample text file that contains integers and floating-point numbers.

```
x = 1:1:5;
y = [x;rand(1,5)];
fileID = fopen('nums2.txt','w');
fprintf(fileID,'%d %4.4f\n',y);
fclose(fileID);
```

View the contents of the file.

```
type nums2.txt
```

```
1 0.8147
2 0.9058
3 0.1270
4 0.9134
5 0.6324
```

Open the file for reading, and obtain the file identifier, `fileID`.

```
fileID = fopen('nums2.txt','r');
```

Define the format of the data to read and the shape of the output array.

```
formatSpec = '%d %f';
sizeA = [2 Inf];
```

Read the file data, filling output array, `A`, in column order. `fscanf` reuses the format, `formatSpec`, throughout the file.

```
A = fscanf(fileID,formatSpec,sizeA)
fclose(fileID);
```

```
A =

    1.0000    2.0000    3.0000    4.0000    5.0000
    0.8147    0.9058    0.1270    0.9134    0.6324
```

Transpose the array so that A matches the orientation of the data in the file.

```
A = A'
```

```
A =

    1.0000    0.8147
    2.0000    0.9058
    3.0000    0.1270
    4.0000    0.9134
    5.0000    0.6324
```

Skip Specific Characters in File

Skip specific characters in a sample file, and return only numeric data.

Create a sample text file containing temperature values.

```
str = '78°C 72°C 64°C 66°C 49°C';
fileID = fopen('temperature.dat','w');
fprintf(fileID,'%s',str);
fclose(fileID);
```

Read the numbers in the file, skipping the text, °C. Also return the number of values that `fscanf` reads. The extended ASCII code 176 represents the degree sign.

```
fileID = fopen('temperature.dat','r');
degrees = char(176);
[A,count] = fscanf(fileID, ['%d' degrees 'C'])
fclose(fileID);
```

```
A =

    78
    72
    64
    66
    49


count =

     5
```

A is a vector containing the numeric values in the file. `count` indicates that `fscanf` read five values.

Input Arguments

`fileID` — File identifier

integer

File identifier of an open text file, specified as an integer. Before reading a file with `fscanf`, you must use `fopen` to open the file and obtain the `fileID`.

Data Types: `double`

`formatSpec` — Format of data fields

string

Format of the data fields in the file, specified as a string of one or more conversion specifiers. When `fscanf` reads a file, it attempts to match the data to the `formatSpec` string.

Numeric Fields

This table lists available conversion specifiers for numeric inputs. `fscanf` converts values to their decimal (base 10) representation.

Numeric Field Type	Conversion Specifier	Details
Integer, signed	`%d`	Base 10
	`%i`	The values in the file determine the base: • The default is base 10. • If the initial digits are `0x` or `0X`, then the values are hexadecimal (base 16). • If the initial digit is `0`, then values are octal (base 8).
	`%ld` or `%li`	64-bit values, base 10, 8, or 16
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)

Numeric Field Type	Conversion Specifier	Details
	`%x`	Base 16 (hexadecimal)
	`%lu`, `%lo`, `%lx`	64-bit values, base 10, 8, or 16
Floating-point number	`%f`	Floating-point fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`.
	`%e`	
	`%g`	

Character Fields

This table lists available conversion specifiers for character inputs.

Character Field Type	Conversion Specifier	Description
Characters	`%s`	Read a string until `fscanf` encounters white space.
	`%c`	Read any single character, including white space. To read multiple characters at a time, specify field width.
Pattern-matching string	`%[...]`	Read only characters in the brackets up to the first nonmatching character or white space. Example: `%[mus]` reads `'summer '` as `'summ'`.

If `formatSpec` contains a combination of numeric and character specifiers, then `fscanf` converts each character to its numeric equivalent. This conversion occurs even when the format explicitly skips all numeric values (for example, `formatSpec` is `'%*d %s'`).

Optional Operators

- Fields and Characters to Ignore

  `fscanf` reads all numeric values and characters in your file in sequence, unless you tell it to ignore a particular field or a portion of a field. To skip fields, insert an asterisk (`*`) after the percent sign (`%`). For example, to skip integers, specify `%*d`.

- Field Width

  To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, `%10c` reads up to 10 characters at a time, including white space. `%4f` reads up to 4 digits at a time, including the decimal point.

- Literal Text to Ignore

  `fscanf` ignores specified text appended to the `formatSpec` string.

  Example: `Level%u` reads `'Level1'` as `1`.

  Example: `%uStep` reads `'2Step'` as `2`.

`sizeA` — Dimensions of output array

`Inf` (default) | integer | two-element row vector

Dimensions of the output array, `A`, specified as `Inf`, an integer, or a two-element row vector.

Form of the `sizeA` Input	Description
`Inf`	Read to the end of the file. For numeric data, the output, `A`, is a column vector. For text data, `A` is a string.
n	Read at most *n* numeric values or character fields. For numeric data, the output, `A`, is a column vector. For text data, `A`, is a string.
[*m*,*n*]	Read at most *m***n* numeric values or character fields. *n* can be `Inf`, but *m* cannot. The output, `A`, is `m`-by-`n`, filled in column order.

Output Arguments

`A` — File data

column vector | matrix | string | character array

File data, returned as a column vector, matrix, string, or character array. The class and size of `A` depend on the `formatSpec` input:

- If `formatSpec` contains only numeric specifiers, then `A` is numeric. If you specify the `sizeA` argument, then `A` is a matrix of the specified size. Otherwise, `A` is a column vector. If the input contains fewer than `sizeA` values, then `fscanf` pads `A` with zeros.
  - If `formatSpec` contains only 64-bit signed integer specifiers, then `A` is of class `int64`.
  - If `formatSpec` contains only 64-bit unsigned integer specifiers, then `A` is of class `uint64`.
  - Otherwise, `A` is of class `double`.
- If `formatSpec` contains only character or string specifiers (`%c` or `%s`), then `A` is a character array. If you specify `sizeA` and the input contains fewer characters, then `fscanf` pads `A` with `char(0)`.
- If `formatSpec` contains a combination of numeric and character specifiers, then `A` is numeric, of class `double`, and `fscanf` converts each text characters to its numeric equivalent. This occurs even when `formatSpec` explicitly skips all numeric fields (for example, `formatSpec` is `'%*d %s'`).
- If MATLAB cannot match the file data to `formatSpec`, then `A` can be numeric or a character array. The class of `A` depends on the values that `fscanf` reads before it stops processing.

`count` — Number of characters read

scalar

Number of characters read, returned as a scalar value.

More About

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

Algorithms

MATLAB reads characters using the encoding scheme associated with the file. You specify the encoding when you open the file using the `fopen` function.

- “Reading Data in a Formatted Pattern”
- “Opening Files with Different Character Encodings”

See Also

`fgetl` | `fgets` | `fopen` | `fprintf` | `fread` | `sscanf` | `textscan`

Introduced before R2006a

fscanf (serial)

Read ASCII data from device, and format as text

Syntax

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

Description

`A = fscanf(obj)` reads ASCII data from the device connected to the serial port object, `obj`, and returns it to `A`. The data is converted to text using the `%c` format. For binary data, use `fread`.

`A = fscanf(obj,'format')` reads data and converts it according to *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'format',size)` reads the number of values specified by `size`. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. If `size` is not of the form `[m,n]`, and a character conversion is specified, then `A` is returned as a row vector. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. An ASCII value is one byte.

`[A,count] = fscanf(...)` returns the number of values read to `count`.

`[A,count,msg] = fscanf(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Examples

Create the serial port object `s` and connect `s` to a Tektronix TDS 210 oscilloscope, which is displaying sine wave. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
```

Use the `fprintf` function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VALUE?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
```

```
ans =
    21
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
```

```
meas =
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',14)
```

```
pk2pk =
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

More About

Tips

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fscanf` is issued.

Rules for Completing a Read Operation with fscanf

A read operation with `fscanf` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The number of values specified by `size` is read.
- The input buffer is filled (unless `size` is specified)

See Also

`fgetl` | `fopen` | `fread` | `fgets` | `textscan` | `BytesAvailable` | `BytesAvailableFcn` | `InputBufferSize` | `Status` | `Terminator` | `Timeout`

Introduced before R2006a

fseek

Move to specified position in file

Syntax

```
fseek(fileID, offset, origin)
status = fseek(fileID, offset, origin)
```

Description

`fseek(fileID, offset, origin)` sets the file position indicator `offset` bytes from `origin` in the specified file.

`status = fseek(fileID, offset, origin)` returns `0` when the operation is successful. Otherwise, it returns `-1`.

Input Arguments

`fileID`

Integer file identifier obtained from `fopen`.

`offset`

Number of bytes to move from `origin`. Can be positive, negative, or zero. The `n` bytes of a given file are in positions `0` through `n-1`.

`origin`

Starting location in the file:

`'bof'` or `-1`	Beginning of file
`'cof'` or `0`	Current position in file
`'eof'` or `1`	End of file

Examples

Copy 5 bytes from the file `test1.dat`, starting at the tenth byte, and append to the end of `test2.dat`:

```
% Create files test1.dat and test2.dat
% Each character uses 8 bits (1 byte)

fid1 = fopen('test1.dat', 'w+');
fwrite(fid1, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');

fid2 = fopen('test2.dat', 'w+');
fwrite(fid2, 'Second File');

% Seek to the 10th byte ('J'), read 5
fseek(fid1, 9, 'bof');
A = fread(fid1, 5, 'uint8=>char');
fclose(fid1);

% Append to test2.dat
fseek(fid2, 0, 'eof');
fwrite(fid2, A);
fclose(fid2);
```

Alternatives

To move to the beginning of a file, call

```
frewind(fileID)
```

This call is identical to

```
fseek(fileID, 0, 'bof')
```

More About

- "Reading Portions of a File"

See Also

`fclose` | `feof` | `ferror` | `fopen` | `frewind` | `fread` | `fwrite` | `ftell` | `fscanf` | `fprintf`

Introduced before R2006a

ftell

Position in open file

Syntax

```
position = ftell(fileID)
```

Description

`position = ftell(fileID)` returns the current position in the specified file. `position` is a zero-based integer that indicates the number of bytes from the beginning of the file. If the query is unsuccessful, `position` is `-1`. `fileID` is an integer file identifier obtained from `fopen`.

See Also

`fclose` | `feof` | `ferror` | `fopen` | `frewind` | `fread` | `fwrite` | `fseek` | `fscanf` | `fprintf`

Introduced before R2006a

FTP class

Connect to FTP server

Description

Connect to an FTP server by calling the `ftp` function, which creates an FTP object. Perform file operations using methods on the FTP object, such as `mput` and `mget`. When you finish accessing the server, call the `close` method to close the connection.

Construction

`f = ftp(host,username,password)` connects to the FTP server host and creates FTP object f. If the host supports anonymous connections, you can use the `host` argument alone. To specify an alternate port, separate it from `host` with a colon (`:`).

Input Arguments

`host`

String enclosed in single quotation marks that specifies the FTP server.

`username`

String enclosed in single quotation marks that specifies your user name for the FTP server.

`password`

String enclosed in single quotation marks that specifies your password for the FTP server. Because FTP is not a secure protocol, others can see your user name and password.

Methods

ascii
: Set FTP transfer type to ASCII

binary	Set FTP transfer type to binary
cd	Change or view current folder on FTP server
close	Close connection to FTP server
delete	Remove file on FTP server
dir	View contents of folder on FTP server
mget	Download files from FTP server
mkdir	Create folder on FTP server
mput	Upload file or folder to FTP server
rename	Rename file on FTP server
rmdir	Remove folder on FTP server

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Modify the following code to connect to an FTP server, and display the FTP object:

```
mw = ftp('ftp.testsite.com');
disp(mw)
```

Connect to port 34 (not supported at `ftp.mathworks.com`, so this code returns an error):

```
mw = ftp('ftp.mathworks.com:34')
```

Modify the following code to connect to a host that requires a password:

```
test = ftp('ftp.testsite.com','myname','mypassword')
```

Algorithms

The `ftp` function is based on code from the Apache Jakarta Project.

Limitations

The `ftp` function does not support proxy server settings.

See Also

`urlwrite`

Introduced before R2006a

full

Convert sparse matrix to full matrix

Syntax

```
A = full(S)
```

Description

`A = full(S)` converts a sparse matrix `S` to full storage organization, such that `issparse(A)` returns logical `0` (`false`). If `S` is a full matrix, then `A` is identical to `S`.

Examples

Here is an example of a sparse matrix with a density of about two-thirds. `sparse(S)` and `full(S)` require about the same number of bytes of storage.

```
S = sparse(+(rand(200,200) < 2/3));
A = full(S);
whos
Name    Size    Bytes     Class
  A    200X200 320000 double array
  S    200X200 318432 double array (sparse)
```

More About

Tips

If `X` is an `m`-by-`n` matrix with `nz` nonzero elements then `full(X)` requires space to store `m*n` elements. On the other hand, `sparse(X)` requires space to store `nz` elements and `(nz+n+1)` integers.

The density of a matrix (`nnz(X)/numel(X)`) determines whether or not it is more efficient to store the matrix as sparse or full. The exact crossover point depends on the

matrix class as well as the platform. For example, in 32-bit MATLAB, a double sparse matrix with less than about 2/3 density will require less space than the same matrix in full storage. In 64-bit MATLAB, however, double matrices with less than half of their elements nonzero are more efficient to store as sparse matrices.

See Also

`issparse` | `sparse`

Introduced before R2006a

fullfile

Build full file name from parts

Syntax

```
f = fullfile(filepart1,...,filepartN)
```

Description

`f = fullfile(filepart1,...,filepartN)` builds a full file specification, `f`, from the folders and file names specified. `fullfile` inserts platform-dependent file separators where necessary, and does not add a trailing file separator. The output of `fullfile` is conceptually equivalent to

```
f = [filepart1 filesep filepart2 filesep ... filesep filepartN]
```

Examples

Create a Full File Name

```
f = fullfile('myfolder','mysubfolder','myfile.m')
```

```
f =
myfolder\mysubfolder\myfile.m
```

`fullfile` returns a string containing the full path to the file. On Windows platforms, the file separator character is a backslash, `\`. On other platforms, the file separator might be a different character.

Create Paths to Multiple Files

```
f = fullfile(toolboxdir('matlab'),'iofun',{'filesep.m';'fullfile.m'});
```

`fullfile` returns a cell array containing a path to the file `filesep.m`, and a path to the file `fullfile.m`.

Create a Path to a Folder

```
f = fullfile(matlabroot,'toolbox','matlab',filesep);
```

`fullfile` does not trim a leading or trailing `filesep`.

Input Arguments

`filepart1,...,filepartN` — Folder and file names

strings | cell arrays of strings

Folder and file names, specified as strings and cell arrays of strings. Any nonscalar cell arrays must be of the same size.

Example: `'folder1','folder2','myfile.m'`

Example: `{'folder1';'folder2'},{'subfolder1';'subfolder2'},'myfile.m'`

See Also

`fileparts` | `filesep` | `genpath` | `path` | `pathsep`

Introduced before R2006a

func2str

Construct function name string from function handle

Syntax

```
func2str(fhandle)
```

Description

`func2str(fhandle)` constructs a string `s` that holds the name of the function to which the function handle `fhandle` belongs.

When you need to perform a string operation, such as compare or display, on a function handle, you can use `func2str` to construct a string bearing the function name.

The `func2str` command does not operate on nonscalar function handles. Passing a nonscalar function handle to `func2str` results in an error.

Examples

Example 1

Convert a `sin` function handle to a string:

```
fhandle = @sin;

func2str(fhandle)
ans =
    sin
```

Example 2

The `catcherr` function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, `catcherr` uses `sprintf` to display an error message giving the name of the failing

function. The function name must be a string for `sprintf` to display it. The code derives the function name from the function handle using `func2str`:

```
function catcherr(func, data)
try
    ans = func(data);
    disp('Answer is:');
    ans
catch
    disp(sprintf('Error executing function ''%s''\n', ...
        func2str(func)))
end
```

The first call to `catcherr` passes a handle to the `round` function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, `round` fails, causing `catcherr` to display an error message that includes the failing function name:

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

More About

Tips

Any variables and their values originally stored in a function handle when it was created are lost if you convert the function handle to a string and back again using the `func2str` and `str2func` functions.

See Also

`function_handle` | `str2func` | `functions`

Introduced before R2006a

function

Declare function name, inputs, and outputs

Syntax

```
function [y1,...,yN] = myfun(x1,...,xM)
```

Description

`function [y1,...,yN] = myfun(x1,...,xM)` declares a function named `myfun` that accepts inputs `x1,...,xM` and returns outputs `y1,...,yN`. This declaration statement must be the first executable line of the function.

Save the function code in a text file with a `.m` extension. The name of the file should match the name of the first function in the file. Valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores.

Files can include multiple local functions or nested functions. Use the `end` keyword to indicate the end of each function in a file if:

- Any function in the file contains a nested function
- Any local function in the file uses the `end` keyword

Otherwise, the `end` keyword is optional.

Examples

Function with One Output

Define a function in a file named `average.m` that accepts an input vector, calculates the average of the values, and returns a single result.

```
function y = average(x)
if ~isvector(x)
    error('Input must be a vector')
end
```

```
y = sum(x)/length(x);
end
```

Call the function from the command line.

```
z = 1:99;
average(z)
```

```
ans =
    50
```

Function with Multiple Outputs

Define a function in a file named `stat.m` that returns the mean and standard deviation of an input vector.

```
function [m,s] = stat(x)
n = length(x);
m = sum(x)/n;
s = sqrt(sum((x-m).^2/n));
end
```

Call the function from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat(values)
```

```
ave =
   47.3400
stdev =
   29.4124
```

Multiple Functions in a File

Define two functions in a file named `stat2.m`, where the first function calls the second.

```
function [m,s] = stat2(x)
n = length(x);
m = avg(x,n);
s = sqrt(sum((x-m).^2/n));
end

function m = avg(x,n)
m = sum(x)/n;
end
```

Function `avg` is a *local function*. Local functions are only available to other functions within the same file.

Call function `stat2` from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat2(values)
```

```
ave =
   47.3400
stdev =
   29.4124
```

- "Create Functions in Files"

More About

- "Local Functions"
- "Nested Functions"
- "Base and Function Workspaces"
- "Function Precedence Order"

See Also

`nargin` | `nargout` | `pcode` | `return` | `varargin` | `varargout` | `what` | `which`

Introduced before R2006a

function_handle (@)

Handle used in calling functions indirectly

Syntax

```
handle = @functionname
handle = @(arglist)anonymous_function
```

Description

`handle = @functionname` returns a handle to the specified MATLAB function.

A function handle is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks). A function handle is one of the standard MATLAB data types.

At the time you create a function handle, the function you specify must be on the MATLAB path and in the current scope of the code creating the handle. For example, you can create a handle to a local function as long as you do so from within the file that defines that local function. This condition does not apply when you evaluate the function handle. You can, for example, execute a local function from a separate (out-of-scope) file using a function handle. This requires that the handle was created by the local function (in-scope).

`handle = @(arglist)anonymous_function` constructs an anonymous function and returns a `handle` to that function. The body of the function, to the right of the parentheses, is a single MATLAB statement or command. `arglist` is a comma-separated list of input arguments. Execute the function by calling it by means of the function handle, `handle`.

Examples

Example 1 — Constructing a Handle to a Named Function

The following example creates a function handle for the `humps` function and assigns it to the variable `fhandle`.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to `fminbnd`, which then minimizes over the interval `[0.3, 1]`.

```
x = fminbnd(fhandle, 0.3, 1)
x =
    0.6370
```

The `fminbnd` function evaluates the `@humps` function handle. A small portion of the `fminbnd` file is shown below. In line 1, the `funfcn` input parameter receives the function handle `@humps` that was passed in. The statement, in line 113, evaluates the handle.

```
1    function [xf,fval,exitflag,output] = ...
         fminbnd(funfcn,ax,bx,options,varargin)
             .
             .
             .
113  fx = funfcn(x,varargin{:});
```

Example 2 — Constructing a Handle to an Anonymous Function

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `integral` function to compute its integral from zero to one:

```
integral(sqr, 0, 1)
ans =
   0.3333
```

Example 3 — Using an Array of Function Handles

This example creates a structure array of function handles `S` and then applies each handle in the array to the output of a `linspace` calculation in one operation using `structfun`:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
structfun(@(x)x(linspace(1,4,3)), S, 'UniformOutput', false)
ans =
    a: [0.8415 0.5985 -0.7568]
    b: [0.5403 -0.8011 -0.6536]
    c: [1.5574 -0.7470 1.1578]
```

More About

Tips

The function handle is a standard MATLAB data type. As such, you can manipulate and operate on function handles in the same manner as on other MATLAB data types. This includes using function handles in structures and cell arrays:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
C = {@sin, @cos, @tan};
```

However, standard matrices or arrays of function handles are not supported:

```
A = [@sin, @cos, @tan];        % This is not supported
```

For nonoverloaded functions, local functions, and private functions, a function handle references just the one function specified in the `@functionname` syntax. When you evaluate an overloaded function by means of its handle, the arguments the handle is evaluated with determine the actual function that MATLAB dispatches to.

Use `isa(h, 'function_handle')` to see if variable `h` is a function handle.

See Also

`str2func` | `func2str` | `functions` | `isa`

Introduced before R2006a

functions

Information about function handle

Syntax

```
S = functions(funhandle)
```

Description

`S = functions(funhandle)` returns, in MATLAB structure `S`, the function name, type, filename, and other information for the function handle stored in the variable `funhandle`.

`functions` does not operate on nonscalar function handles. Passing a nonscalar function handle to `functions` results in an error.

Caution The `functions` function is provided for querying and debugging purposes. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

This table lists the standard fields of the return structure.

Field Name	Field Description
`function`	Function name
`type`	Function type (e.g., simple, overloaded)
`file`	The file to be executed when the function handle is evaluated with a nonoverloaded data type

Examples

Example 1

To obtain information on a function handle for the `poly` function, type

```
f = functions(@poly)
f =
    function: 'poly'
        type: 'simple'
        file:  '$matlabroot\toolbox\matlab\polyfun\poly.m'
```

(The term `$matlabroot` used in this example stands for the file specification of the directory in which MATLAB software is installed for your system. Your output will display this file specification.)

Access individual fields of the returned structure using dot selection notation:

```
f.type
ans =
   simple
```

Example 2

The function `get_handles` returns function handles for a local function and private function in output arguments `l` and `p` respectively:

```
function [l, p] = get_handles
l = @mylocfun;
p = @myprivatefun;
%
function mylocfun
disp 'Executing local function mylocfun'
```

Call `get_handles` to obtain the two function handles, and then pass each to the `functions` function. MATLAB returns information in a structure having the fields `function`, `type`, `file`, and `parentage`. The `file` field contains the file specification for the local or private function:

```
[floc fprv] = get_handles;

functions(floc)
ans =
     function: 'mylocfun'
         type: 'scopedfunction'
         file: 'c:\matlab\get_handles.m'
    parentage: {'mylocfun'  'get_handles'}

functions(fprv)
ans =
```

```
     function: 'myprivatefun'
         type: 'scopedfunction'
         file: 'c:\matlab\private\myprivatefun.m'
    parentage: {'myprivatefun'}
```

Example 3

In this example, the function `get_handles_nested.m` contains a nested function `nestfun`. This function has a single output which is a function handle to the nested function:

```
function handle = get_handles_nested(A)
nestfun(A);

   function y = nestfun(x)
   y = x + 1;
   end

handle = @nestfun;
end
```

Call this function to get the handle to the nested function. Use this handle as the input to `functions` to return the information shown here. Note that the `function` field of the return structure contains the names of the nested function and the function in which it is nested in the format. Also note that `functions` returns a `workspace` field containing the variables that are in context at the time you call this function by its handle:

```
fh = get_handles_nested(5);

fhinfo = functions(fh)
fhinfo =
     function: 'get_handles_nested/nestfun'
         type: 'nested'
         file: 'c:\matlab\get_handles_nested.m'
    workspace: [1x1 struct]

fhinfo.workspace
ans =
    handle: @get_handles_nested/nestfun
         A: 5
```

See Also

`func2str` | `function_handle` | `str2func`

Introduced before R2006a

functiontests

Create array of tests from handles to local functions

Syntax

```
tests = functiontests(f)
```

Description

`tests = functiontests(f)` creates an array of tests, `tests`, from a cell array of handles to local functions, `f`. To apply defined setup and teardown functions, include their function handles in `f`.

Local test functions must include 'test' at the beginning or end of the function name. `functiontests` must be called from within a test file.

Examples

Create Test Array

Create the file `exampleTest.m` in your MATLAB path. In the main function, create a test array. Use local functions to define setup, teardown, and two function tests. Your file should look like this.

```
function tests = exampleTest
tests = functiontests(localfunctions);

function setup(testCase)
function teardown(testCase)
function exampleOneTest(testCase)
function testExampleTwo(testCase)
```

From the command line, call the `exampleTest` function.

```
tests = exampleTest
```

```
tests =

  1x2 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Access the test suite to verify the names of the two function tests.

```
tests.Name
```

```
ans =

exampleTest/exampleOneTest


ans =

exampleTest/testExampleTwo
```

- "Write Function-Based Unit Tests"
- "Write Simple Test Case Using Functions"
- "Write Test Using Setup and Teardown Functions"

Input Arguments

`f` — Handles to local test functions

cell array of function handles

Handles to local test functions, specified as a cell array. Use `f=localfunctions` in your working file to automatically generate a cell array of function handles for that file. If you want explicit test enumeration, construct `f` by listing individual functions. `f` must include any setup or teardown functions necessary for your test.

Example: `f = localfunctions;`

Example: `f = {@setup,@exampleOneTest,@teardown};`

See Also

`matlab.unittest.Test` | `localfunctions` | `runtests`

funm

Evaluate general matrix function

Syntax

```
F = funm(A,fun)
F = funm(A,fun,options)
F = funm(A,fun,options,p1,p2,...)
[F,exitflag] = funm(...)
[F,exitflag,output] = funm(...)
```

Description

`F = funm(A,fun)` evaluates the user-defined function `fun` at the square matrix argument `A`. `F = fun(x,k)` must accept a vector `x` and an integer `k`, and return a vector `f` of the same size of `x`, where `f(i)` is the `k`th derivative of the function `fun` evaluated at `x(i)`. The function represented by fun must have a Taylor series with an infinite radius of convergence, except for `fun = @log`, which is treated as a special case.

You can also use `funm` to evaluate the special functions listed in the following table at the matrix `A`.

Function	Syntax for Evaluating Function at Matrix A
`exp`	`funm(A, @exp)`
`log`	`funm(A, @log)`
`sin`	`funm(A, @sin)`
`cos`	`funm(A, @cos)`
`sinh`	`funm(A, @sinh)`
`cosh`	`funm(A, @cosh)`

For matrix square roots, use `sqrtm(A)` instead. For matrix exponentials, which of `expm(A)` or `funm(A, @exp)` is the more accurate depends on the matrix `A`.

The function represented by `fun` must have a Taylor series with an infinite radius of convergence. The exception is `@log`, which is treated as a special case. “Parameterizing

Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`F = funm(A,fun,options)` sets the algorithm's parameters to the values in the structure `options`.

The following table lists the fields of `options`.

Field	Description	Values
`options.Display`	Level of display	`'off'` (default), `'on'`, `'verbose'`
`options.TolBlk`	Tolerance for blocking Schur form	Positive scalar. The default is `0.1`.
`options.TolTay`	Termination tolerance for evaluating the Taylor series of diagonal blocks	Positive scalar. The default is `eps`.
`options.MaxTerms`	Maximum number of Taylor series terms	Positive integer. The default is `250`.
`options.MaxSqrt`	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is `100`.
`options.Ord`	Specifies the ordering of the Schur form `T`.	A vector of length `length(A)`. `options.Ord(i)` is the index of the block into which `T(i,i)` is placed. The default is `[]`.

`F = funm(A,fun,options,p1,p2,...)` passes extra inputs `p1,p2,...` to the function.

`[F,exitflag] = funm(...)` returns a scalar `exitflag` that describes the exit condition of `funm`. `exitflag` can have the following values:

- `0` — The algorithm was successful.
- `1` — One or more Taylor series evaluations did not converge, or, in the case of a logarithm, too many square roots are needed. However, the computed value of `F` might still be accurate.

`[F,exitflag,output] = funm(...)` returns a structure `output` with the following fields:

Field	Description
`output.terms`	Vector for which `output.terms(i)` is the number of Taylor series terms used when evaluating the *i*th block, or, in the case of the logarithm, the number of square roots of matrices of dimension greater than 2.
`output.ind`	Cell array for which the `(i,j)` block of the reordered Schur factor `T` is `T(output.ind{i}, output.ind{j})`.
`output.ord`	Ordering of the Schur form, as passed to `ordschur`
`output.T`	Reordered Schur form

If the Schur form is diagonal then `output = struct('terms',ones(n,1),'ind',{1:n})`.

Examples

Example 1

The following command computes the matrix sine of the 3-by-3 magic matrix.

```
F=funm(magic(3), @sin)

F =

   -0.3850    1.0191    0.0162
    0.6179    0.2168   -0.1844
    0.4173   -0.5856    0.8185
```

Example 2

The statements

```
S = funm(X,@sin);
C = funm(X,@cos);
```

produce the same results to within roundoff error as

```
E = expm(i*X);
C = real(E);
S = imag(E);
```

In either case, the results satisfy `S*S+C*C = I`, where `I = eye(size(X))`.

Example 3

To compute the function `exp(x) + cos(x)` at `A` with one call to `funm`, use

```
F = funm(A,@fun_expcos)
```

where `fun_expcos` is the following function.

```
function f = fun_expcos(x, k)
% Return kth derivative of exp + cos at X.
        g = mod(ceil(k/2),2);
        if mod(k,2)
           f = exp(x) + sin(x)*(-1)^g;
        else
           f = exp(x) + cos(x)*(-1)^g;
        end
```

More About

Algorithms

The algorithm `funm` uses is described in [1].

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Third Edition, Johns Hopkins University Press, 1996, p. 384.

[3] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later" *SIAM Review 20*, Vol. 45, Number 1, pp. 1-47, 2003.

See Also

`expm` | `logm` | `sqrtm` | `function_handle`

Introduced before R2006a

fwrite

Write data to binary file

Syntax

```
fwrite(fileID,A)
fwrite(fileID,A,precision)
fwrite(fileID,A,precision,skip)
fwrite(fileID,A,precision,skip,machinefmt)

count = fwrite( ___ )
```

Description

`fwrite(fileID,A)` write the elements of array `A` as 8-bit unsigned integers to a binary file in column order. The binary file is indicated by the file identifier, `fileID`. Use `fopen` to open the file and obtain the `fileID` value. When you finish reading, close the file by calling `fclose(fileID)`.

`fwrite(fileID,A,precision)` writes the values in `A` in the form and size described by `precision`.

`fwrite(fileID,A,precision,skip)` skips the number of bytes or bits specified by `skip` before writing each value.

`fwrite(fileID,A,precision,skip,machinefmt)` additionally specifies the order for writing bytes or bits to the file. The `skip` argument is optional.

`count = fwrite( ___ )` returns the number of elements of `A` that `fwrite` successfully writes to the file. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Write uint8 Data to Binary File

Open a file named `nine.bin` for writing. Specify write access using `'w'` in the call to `fopen`.

```
fileID = fopen('nine.bin','w');
```

`fopen` returns a file identifier, `fileID`.

Write the integers from 1 to 9 as 8-bit unsigned integers.

```
fwrite(fileID,[1:9]);
```

Close the file.

```
fclose(fileID);
```

Write 4-byte Integers to Binary File

Open a file named `magic5.bin` for writing.

```
fileID = fopen('magic5.bin','w');
```

Write the 25 elements of the 5-by-5 magic square. Use the `precision` argument, `'integer*4'`, to write 4-byte integers.

```
fwrite(fileID,magic(5),'integer*4');
```

Close the file.

```
fclose(fileID);
```

Append Data to Binary File

Write a binary file containing the elements of the 4-by-4 magic square, stored as double-precision floating-point numbers.

```
fileID = fopen('magic4.bin','w');
fwrite(fileID,magic(4),'double');
fclose(fileID);
```

Open the file, `magic4.bin`, with write-access that enables appending to the file. Specify the file-access type, `'a'`, in the call to `fopen`.

```
fileID = fopen('magic4.bin','a');
```

Append a 4-by-4 matrix of zeros to the file. Then, close the file.

```
fwrite(fileID,zeros(4),'double');
fclose(fileID);
```

Write Binary File with Big-Endian Byte Ordering

Write random double-precision numbers to a file named `myfile.bin` for use on a big-endian system. Specify a `machinefmt` value of `'ieee-be'` in the call to `fwrite`, to indicate big-endian byte ordering.

```
fileID = fopen('myfile.bin','w');
fwrite(fileID,rand(4),'double','ieee-be');
fclose(fileID);
```

- "Writing and Reading Complex Numbers"

Input Arguments

`fileID` — File identifier

integer | 1 | 2

File identifier, specified as an integer obtained from `fopen`, `1` for standard output (the screen), or `2` for standard error.

A — Data to write

numeric array | character array

Data to write, specified as a numeric or character array.

Example: `[1,2,3;4,5,6]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

`precision` — Class and size of values to write

`uint8` (default) | string

Class and size in bits of the values to write, specified as one of the following strings in the Precision column.

Value Type	Precision	Bits (Bytes)
Integers, unsigned	`uint`	32 (4)
	`uint8`	8 (1)
	`uint16`	16 (2)
	`uint32`	32 (4)
	`uint64`	64 (8)
	`uchar`	8 (1)
	`unsigned char`	8 (1)
	`ushort`	16 (2)
	`ulong`	32 (4)
	`ubit`*n*	$1 \le n \le 64$
Integers, signed	`int`	32 (4)
	`int8`	8 (1)
	`int16`	16 (2)
	`int32`	32 (4)
	`int64`	64 (8)
	`integer*1`	8 (1)
	`integer*2`	16 (2)
	`integer*4`	32 (4)
	`integer*8`	64 (8)
	`schar`	8 (1)
	`signed char`	8 (1)
	`short`	16 (2)
	`long`	32 (4)
	`bit`*n*	$1 \le n \le 64$
Floating-point numbers	`single`	32 (4)
	`double`	64 (8)

Value Type	Precision	Bits (Bytes)
	`float`	`32 (4)`
	`float32`	`32 (4)`
	`float64`	`64 (8)`
	`real*4`	`32 (4)`
	`real*8`	`64 (8)`
Characters	`char*1`	`8 (1)`
	`char`	Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

If you specify a precision of `bit`*n* or `ubit`*n*, then `fwrite` saturates for all values outside the range.

Note: To preserve `NaN` and `Inf` values in MATLAB, read and write data of class `double` or `single`.

`skip` — Number of bytes to skip

0 (default) | scalar

Number of bytes to skip before writing each value, specified as a scalar. If you specify a `precision` of `bit`*n* or `ubit`*n*, specify `skip` in bits.

Use the `skip` argument to insert data into noncontiguous fields in fixed-length records.

`machinefmt` — Order for writing bytes

`'n'` (default) | `'b'` | `'l'` | `'s'` | `'a'`

Order for writing bytes within the file, specified as one of the strings in the table that follows. For `bit`*n* and `ubit`*n* precisions, `machinefmt` specifies the order for writing bits within a byte, but the order for writing bytes remains your system byte ordering.

`'n'` or `'native'`	Your system byte ordering (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering

`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big-endian or little-endian ordering.

More About

- "Overwriting or Appending to an Existing File"
- "Creating a File for Use on a Different System"

See Also

`fclose` | `ferror` | `fopen` | `fprintf` | `fread` | `fscanf` | `fseek` | `ftell`

Introduced before R2006a

fwrite (serial)

Write binary data to device

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Description

`fwrite(obj,A)` writes the binary data `A` to the device connected to the serial port object, `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *`precision`*.

`precision` controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. The supported values for *`precision`* are listed in Tips.

`fwrite(obj,A,'mode')` writes binary data with command-line access specified by *`mode`*. If *`mode`* is `sync`, `A` is written synchronously and the command line is blocked. If *`mode`* is `async`, `A` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fwrite(obj,A,'precision','mode')` writes binary data with precision specified by *`precision`* and command-line access specified by *`mode`*.

More About

Tips

Before you can write data to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error

is returned if you attempt to perform a write operation while `obj` is not connected to the device.

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

If you set the `FlowControl` property to `hardware` on a serial object, and a hardware connection is not detected, `fwrite` returns an error message. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check the remote device status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note: If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device, check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fwrite`.

Synchronous Versus Asynchronous Write Operations

By default, data is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in Writing Data.

Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used with binary write operations.

Supported Precisions

The following table shows the supported values for *`precision`*.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point

Data Type	Precision	Interpretation
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

`fopen` | `fprintf` | `BytesToOutput` | `OutputBufferSize` | `OutputEmptyFcn` | `Status` | `Timeout` | `TransferStatus` | `ValuesSent`

Introduced before R2006a

fzero

Root of nonlinear function

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)

x = fzero(problem)

[x,fval,exitflag,output] = fzero( ___ )
```

Description

`x = fzero(fun,x0)` tries to find a point `x` where `fun(x) = 0`. This solution is where `fun(x)` changes sign—`fzero` cannot find a root of a function such as `x^2`.

`x = fzero(fun,x0,options)` uses `options` to modify the solution process.

`x = fzero(problem)` solves a root-finding problem specified by `problem`.

`[x,fval,exitflag,output] = fzero( ___ )` returns `fun(x)` in the `fval` output, `exitflag` encoding the reason `fzero` stopped, and an output structure containing information on the solution process.

Examples

Root Starting From One Point

Calculate π by finding the zero of the sine function near `3`.

```
fun = @sin; % function
x0 = 3; % initial point
x = fzero(fun,x0)
```

```
x =

    3.1416
```

Root Starting From an Interval

Find the zero of cosine between 1 and 2.

```
fun = @cos; % function
x0 = [1 2]; % initial interval
x = fzero(fun,x0)
```

```
x =

    1.5708
```

Note that $\cos(1)$ and $\cos(2)$ differ in sign.

Root of a Function Defined by a File

Find a zero of the function $f(x) = x^3 - 2x - 5$.

First, write a file called `f.m`.

```
function y = f(x)
y = x.^3 - 2*x - 5;
```

Save `f.m` on your MATLAB path.

Find the zero of $f(x)$ near 2.

```
fun = @f; % function
x0 = 2; % initial point
z = fzero(fun,x0)
```

```
z =
    2.0946
```

Since `f(x)` is a polynomial, you can find the same real zero, and a complex conjugate pair of zeros, using the `roots` command.

```
roots([1 0 -2 -5])

   ans =
   2.0946
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

Root of Function with Extra Parameter

Find the root of a function that has an extra parameter.

```
myfun = @(x,c) cos(c*x);  % parameterized function
c = 2;                    % parameter
fun = @(x) myfun(x,c);    % function of x alone
x = fzero(fun,0.1)


x =

    0.7854
```

Nondefault Options

Plot the solution process by setting some plot functions.

Define the function and initial point.

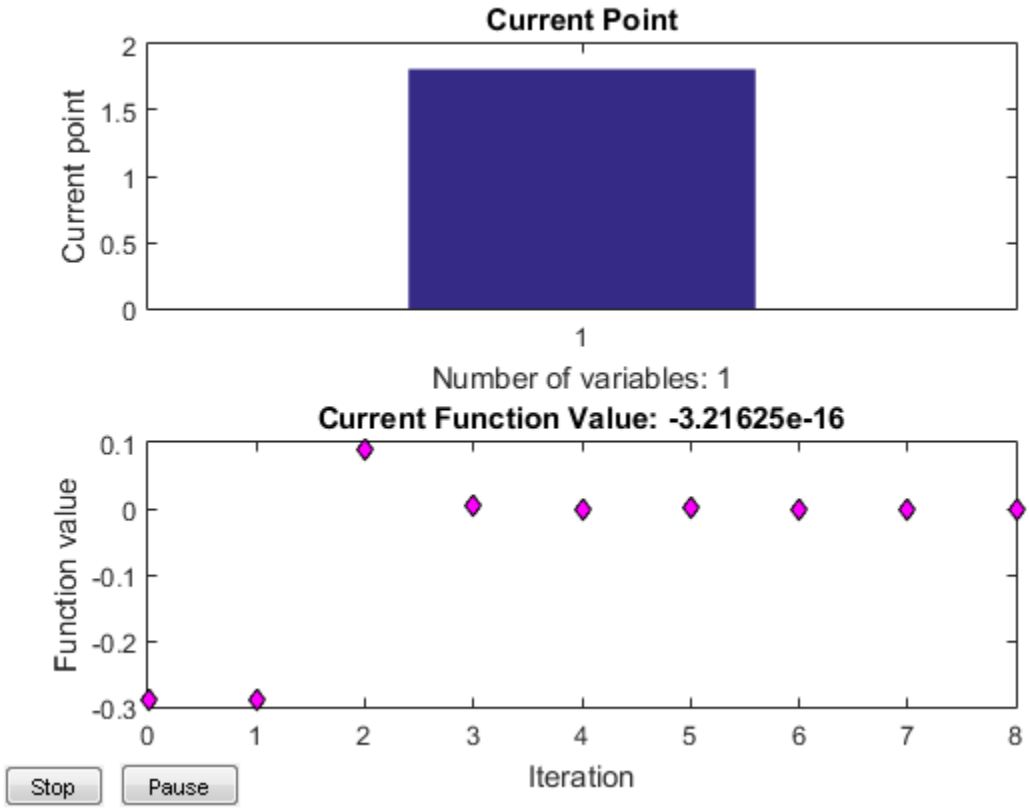
```
fun = @(x)sin(cosh(x));
x0 = 1;
```

Examine the solution process by setting options that include plot functions.

```
options = optimset('PlotFcns',{@optimplotx,@optimplotfval});
```

Run `fzero` including `options`.

```
x = fzero(fun,x0,options)


x =

    1.8115
```



Solve Problem Structure

Solve a problem that is defined by a problem structure.

Define a structure that encodes a root-finding problem.

```
problem.objective = @(x)sin(cosh(x));
problem.x0 = 1;
problem.solver = 'fzero'; % a required part of the structure
problem.options = optimset(@fzero); % default options
```

Solve the problem.

```
x = fzero(problem)
```

```
x =

    1.8115
```

More Information from Solution

Find the point where `exp(-exp(-x)) = x`, and display information about the solution process.

```
fun = @(x) exp(-exp(-x)) - x; % function
x0 = [0 1]; % initial interval
options = optimset('Display','iter'); % show iterations
[x fval exitflag output] = fzero(fun,x0,options)
```

```
 Func-count    x          f(x)             Procedure
    2              1     -0.307799        initial
    3        0.544459     0.0153522        interpolation
    4        0.566101    0.00070708        interpolation
    5        0.567143  -1.40255e-08        interpolation
    6        0.567143   1.50013e-12        interpolation
    7        0.567143             0        interpolation

Zero found in the interval [0, 1]

x =

    0.5671


fval =

     0


exitflag =

     1


output =

    intervaliterations: 0
```

```
       iterations: 5
        funcCount: 7
        algorithm: 'bisection, interpolation'
          message: 'Zero found in the interval [0, 1]'
```

`fval` = 0 means `fun(x) = 0`, as desired.

- "Roots of Scalar Functions"
- "Parameterizing Functions"

Input Arguments

`fun` — Function to solve

function handle

Function to solve, specified as a handle to a scalar-valued function. `fun` accepts a scalar `x` and returns a scalar `fun(x)`.

`fzero` solves `fun(x) = 0`. To solve an equation `fun(x) = c(x)`, instead solve `fun2(x) = fun(x) - c(x) = 0`.

To include extra parameters in your function, see the example "Root of Function with Extra Parameter" on page 1-2858 and the section "Parameterizing Functions".

Example: `@sin`

Example: `@myFunction`

Example: `@(x)(x-a)^5 - 3*x + a - 1`

Data Types: `function_handle`

`x0` — Initial value

scalar | 2-element vector

Initial value, specified as a real scalar or a 2-element real vector.

- Scalar — `fzero` begins at `x0` and tries to locate a point `x1` where `fun(x1)` has the opposite sign of `fun(x0)`. Then `fzero` iteratively shrinks the interval where `fun` changes sign to reach a solution.
- 2-element vector — `fzero` checks that `fun(x0(1))` and `fun(x0(2))` have opposite signs, and errors if they do not. It then iteratively shrinks the interval where `fun`

changes sign to reach a solution. An interval `x0` must be finite; it cannot contain `±Inf`.

Tip Calling `fzero` with an interval (`x0` with two elements) is often faster than calling it with a scalar `x0`.

Example: 3

Example: [2,17]

Data Types: `double`

`options` — Options for solution process

structure, typically created using `optimset`

Options for solution process, specified as a structure. Create or modify the `options` structure using `optimset`. `fzero` uses these `options` structure fields.

`Display`	Level of display: • `'off'` displays no output. • `'iter'` displays output at each iteration. • `'final'` displays just the final output. • `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. • `'on'` displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`. • The default, `'off'`, displays no error.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See “Output Functions”.
`PlotFcns`	Plot various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`).

	• `@optimplotx` plots the current point. • `@optimplotfval` plots the function value. For information on writing a custom plot function, see "Plot Functions".
`TolX`	Termination tolerance on `x`, a positive scalar. The default is `eps`, 2.2204e–16.

Example: `options = optimset('FunValCheck','on')`

Data Types: `struct`

`problem` — Root-finding problem

structure

Root-finding problem, specified as a structure with all of the following fields.

`objective`	Objective function
`x0`	Initial point for `x`, real scalar or 2-element vector
`solver`	`'fzero'`
`options`	Options structure, typically created using `optimset`

For an example, see "Solve Problem Structure" on page 1-2859.

Data Types: `struct`

Output Arguments

`x` — Location of root or sign change

real scalar

Location of root or sign change, returned as a scalar.

`fval` — Function value at `x`

real scalar

Function value at `x`, returned as a scalar.

`exitflag` — Integer encoding the exit condition

integer

Integer encoding the exit condition, meaning the reason `fsolve` stopped its iterations.

`1`	Function converged to a solution `x`.
`-1`	Algorithm was terminated by the output function or plot function.
`-3`	`NaN` or `Inf` function value was encountered while searching for an interval containing a sign change.
`-4`	Complex function value was encountered while searching for an interval containing a sign change.
`-5`	Algorithm might have converged to a singular point.
`-6`	`fzero` did not detect a sign change.

`output` — Information about root-finding process

structure

Information about root-finding process, returned as a structure. The fields of the structure are:

`intervaliterations`	Number of iterations taken to find an interval containing a root
`iterations`	Number of zero-finding iterations
`funcCount`	Number of function evaluations
`algorithm`	`'bisection, interpolation'`
`message`	Exit message

More About

Algorithms

The `fzero` command is a function file. The algorithm, created by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which `fzero` is based, is in [2].

References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

See Also

`fminbnd` | `optimset` | `roots`

Introduced before R2006a

matlab.unittest.FunctionTestCase class

Package: matlab.unittest
Superclasses: matlab.unittest.TestCase

TestCase used for function-based tests

Description

The `FunctionTestCase` class is a subclass of `TestCase` that allows function-based tests to use qualification functions in the `matlab.unittest.qualifications` package. For each test function, MATLAB creates an instance of the `FunctionTestCase` class and passes it to the test function.

The `functiontests` function constructs `FunctionTestCase` instances, so there is no need for test authors to construct this class directly.

See Also

`matlab.unittest.TestCase` | `functiontests` | `runtests`

Introduced in R2013b

gallery

Test matrices

Syntax

```
[A,B,C,...] = gallery(matname,P1,P2,...)
[A,B,C,...] = gallery(matname,P1,P2,...,classname)
gallery(3)
gallery(5)
```

Description

`[A,B,C,...] = gallery(matname,P1,P2,...)` returns the test matrices specified by the quoted string `matname`. The `matname` input is the name of a matrix family selected from the table below. `P1,P2,...` are input parameters required by the individual matrix family. The number of optional parameters `P1,P2,...` used in the calling syntax varies from matrix to matrix. The exact calling syntaxes are detailed in the individual matrix descriptions below.

`[A,B,C,...] = gallery(matname,P1,P2,...,classname)` produces a matrix of class `classname`. The `classname` input is a quoted string that must be either `'single'` or `'double'` (unless `matname` is `'integerdata'`, in which case `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, and `'uint32'` are also allowed). If `classname` is not specified, then the class of the matrix is determined from those arguments among `P1,P2,...` that do not specify dimensions or select an option. If any of these arguments is of class `single` then the matrix is `single`; otherwise the matrix is `double`.

`gallery(3)` is a badly conditioned 3-by-3 matrix and `gallery(5)` is an interesting eigenvalue problem.

The gallery holds over fifty different test matrix functions useful for testing algorithms and other purposes.

binomial — Multiple of involutory matrix

`A = gallery('binomial',n)` returns an `n`-by-`n` matrix,with integer entries such that `A^2 = 2^(n-1)*eye(n)`.

Thus, `B = A*2^((1-n)/2)` is involutory, that is, `B^2 = eye(n)`.

cauchy — Cauchy matrix

`C = gallery('cauchy',x,y)` returns an `n`-by-`n` matrix, `C(i,j) = 1/(x(i)+y(j))`. Arguments `x` and `y` are vectors of length `n`. If you pass in scalars for `x` and `y`, they are interpreted as vectors `1:x` and `1:y`.

`C = gallery('cauchy',x)` returns the same as above with `y = x`. That is, the command returns `C(i,j) = 1/(x(i)+x(j))`.

Explicit formulas are known for the inverse and determinant of a Cauchy matrix. The determinant `det(C)` is nonzero if `x` and `y` both have distinct elements. `C` is totally positive if `0 < x(1) <... < x(n)` and `0 < y(1) < ... < y(n)`.

chebspec — Chebyshev spectral differentiation matrix

`C = gallery('chebspec',n,switch)` returns a Chebyshev spectral differentiation matrix of order `n`. Argument `switch` is a variable that determines the character of the output matrix. By default, `switch` = `0`.

For `switch` = `0` (“no boundary conditions”), `C` is nilpotent (`C^n = 0`) and has the null vector `ones(n,1)`. The matrix `C` is similar to a Jordan block of size `n` with eigenvalue zero.

For `switch` = `1`, `C` is nonsingular and well-conditioned, and its eigenvalues have negative real parts.

The eigenvector matrix of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand — Vandermonde-like matrix for the Chebyshev polynomials

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where `m` is scalar, produces a rectangular version of the above, with `m` rows.

If p is a vector, then $C(i,j) = T_{i-1}(p(j))$ where T_{i-1} is the Chebyshev polynomial of degree $i-1$. If p is a scalar, then p equally spaced points on the interval [0,1] are used to calculate C.

chow — Singular Toeplitz lower Hessenberg matrix

A = gallery('chow',n,alpha,delta) returns A such that A = H(alpha) + delta*eye(n), where $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$ and argument n is the order of the Chow matrix. Default value for scalars alpha and delta are 1 and 0, respectively.

H(alpha) has p = floor(n/2) eigenvalues that are equal to zero. The rest of the eigenvalues are equal to 4*alpha*cos(k*pi/(n+2))^2, k=1:n-p.

circul — Circulant matrix

C = gallery('circul',v) returns the circulant matrix whose first row is the vector v.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals "wrap around."

If v is a scalar, then C = gallery('circul',1:v).

The eigensystem of C (n-by-n) is known explicitly: If t is an nth root of unity, then the inner product of v and $w = [1\ t\ t^2\ \ldots\ t^{(n-1)}]$ is an eigenvalue of C and w(n:-1:1) is an eigenvector.

clement — Tridiagonal matrix with zero diagonal entries

A = gallery('clement',n,k) returns an n-by-n tridiagonal matrix with zeros on its main diagonal and known eigenvalues. It is singular if n is odd. About 64 percent of the entries of the inverse are zero. The eigenvalues include plus and minus the numbers n-1, n-3, n-5, ..., (1 or 0).

For k=0 (the default), A is nonsymmetric. For k=1, A is symmetric.

gallery('clement',n,1) is diagonally similar to gallery('clement',n).

For odd N = 2*M+1, M+1 of the singular values are the integers sqrt((2*M+1)^2 - (2*K+1).^2), K = 0:M.

Note: Similar properties hold for `gallery('tridiag',x,y,z)` where `y = zeros(n,1)`. The eigenvalues still come in plus/minus pairs but they are not known explicitly.

compar — Comparison matrices

`A = gallery('compar',A,1)` returns `A` with each diagonal element replaced by its absolute value, and each off-diagonal element replaced by minus the absolute value of the largest element in absolute value in its row. However, if `A` is triangular `compar(A,1)` is too.

`gallery('compar',A)` is `diag(B) - tril(B,-1) - triu(B,1)`, where `B = abs(A)`. `compar(A)` is often denoted by M(*A*) in the literature.

`gallery('compar',A,0)` is the same as `gallery('compar',A)`.

condex — Counter-examples to matrix condition number estimators

`A = gallery('condex',n,k,theta)` returns a "counter-example" matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k`:

`k = 1`	4-by-4	LINPACK
`k = 2`	3-by-3	LINPACK
`k = 3`	arbitrary	LINPACK (`rcond`) (independent of `theta`)
`k = 4`	`n >= 4`	LAPACK (RCOND) (default). It is the inverse of this matrix that is a counter-example.

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

cycol — Matrix whose columns repeat cyclically

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one "cycle" consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`, and `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol', [n n],k)`.

dorr — Diagonally dominant, ill-conditioned, tridiagonal matrix

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining an `n`-by-`n`, row diagonally dominant, tridiagonal matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is `0.01`. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

dramadah — Matrix of zeros and ones whose inverse has large integer entries

`A = gallery('dramadah',n,k)` returns an `n`-by-`n` matrix of `0`'s and `1`'s for which `mu(A) = norm(inv(A),'fro')` is relatively large, although not necessarily maximal. An anti-Hadamard matrix `A` is a matrix with elements `0` or `1` for which `mu(A)` is maximal.

`n` and `k` must both be scalars. Argument `k` determines the character of the output matrix:

k = 1	Default. `A` is Toeplitz, with `abs(det(A)) = 1`, and `mu(A) > c(1.75)^n`, where `c` is a constant. The inverse of `A` has integer entries.
k = 2	`A` is upper triangular and Toeplitz. The inverse of `A` has integer entries.
k = 3	`A` has maximal determinant among lower Hessenberg (0,1) matrices. `det(A)` = the `n`th Fibonacci number. `A` is Toeplitz. The eigenvalues have an interesting distribution in the complex plane.

fiedler — Symmetric matrix

`A = gallery('fiedler',c)`, where `c` is a length `n` vector, returns the `n`-by-`n` symmetric matrix with elements `abs(n(i)-n(j))`. For scalar `c`, `A = gallery('fiedler',1:c)`.

Matrix `A` has a dominant positive eigenvalue and all the other eigenvalues are negative.

Explicit formulas for `inv(A)` and `det(A)` are given in [Todd, J., *Basic Numerical Mathematics*, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New

York, 1977, p. 159] and attributed to Fiedler. These indicate that `inv(A)` is tridiagonal except for nonzero `(1,n)` and `(n,1)` elements.

forsythe — Perturbed Jordan block

`A = gallery('forsythe',n,alpha,lambda)` returns the `n`-by-`n` matrix equal to the Jordan block with eigenvalue `lambda`, excepting that `A(n,1) = alpha`. The default values of scalars `alpha` and `lambda` are `sqrt(eps)` and `0`, respectively.

The characteristic polynomial of `A` is given by:

```
det(A-t*I) = (lambda-t)^N - alpha*(-1)^n.
```

frank — Matrix with ill-conditioned eigenvalues

`F = gallery('frank',n,k)` returns the Frank matrix of order `n`. It is upper Hessenberg with determinant `1`. If `k = 1`, the elements are reflected about the anti-diagonal `(1,n)` — `(n,1)`. The eigenvalues of `F` may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if `n` is odd, `1` is an eigenvalue. `F` has `floor(n/2)` ill-conditioned eigenvalues — the smaller ones.

gcdmat — Greatest common divisor matrix

`A = gallery('gcdmat',n)` returns the `n`-by-`n` matrix with `(i,j)` entry `gcd(i,j)`. Matrix`A` is symmetric positive definite, and `A.^r` is symmetric positive semidefinite for all nonnegative `r`.

gearmat — Gear matrix

`A = gallery('gearmat',n,i,j)` returns the `n`-by-`n` matrix with ones on the sub- and super-diagonals, `sign(i)` in the `(1,abs(i))` position, `sign(j)` in the `(n,n+1-abs(j))` position, and zeros everywhere else. Arguments `i` and `j` default to `n` and `-n`, respectively.

Matrix `A` is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form `2*cos(a)` and the eigenvectors are of the form `[sin(w+a), sin(w+2*a), ..., sin(w+n*a)]`, where `a` and `w` are given in Gear, C. W., "A Simple Set of Test Matrices for Eigenvalue Programs," *Math. Comp.*, Vol. 23 (1969), pp. 119-125.

grcar — Toeplitz matrix with sensitive eigenvalues

`A = gallery('grcar',n,k)` returns an `n`-by-`n` Toeplitz matrix with `-1`s on the subdiagonal, `1`s on the diagonal, and `k` superdiagonals of `1`s. The default is `k = 3`. The eigenvalues are sensitive.

hanowa — Matrix whose eigenvalues lie on a vertical line in the complex plane

`A = gallery('hanowa',n,d)` returns an `n`-by-`n` block 2-by-2 matrix of the form:

```
[d*eye(m) -diag(1:m)
diag(1:m) d*eye(m)]
```

Argument `n` is an even integer `n=2*m`. Matrix `A` has complex eigenvalues of the form `d ± k*i`, for `1 <= k <= m`. The default value of `d` is `-1`.

house — Householder matrix

`[v,beta,s] = gallery('house',x,k)` takes `x`, an `n`-element column vector, and returns `V` and `beta` such that `H*x = s*e1`. In this expression, `e1` is the first column of `eye(n)`, `abs(s) = norm(x)`, and `H = eye(n) - beta*V*V'` is a Householder matrix.

`k` determines the sign of `s`:

`k = 0`	`sign(s) = -sign(x(1))` (default)
`k = 1`	`sign(s) = sign(x(1))`
`k = 2`	`sign(s) = 1` (`x` must be real)

If `x` is complex, then `sign(x) = x./abs(x)` when `x` is nonzero.

If `x = 0`, or if `x = alpha*e1` (`alpha >= 0`) and either `k = 1` or `k = 2`, then `V = 0`, `beta = 1`, and `s = x(1)`. In this case, `H` is the identity matrix, which is not strictly a Householder matrix.

`[v, beta] = gallery('house',x)` takes `x`, a scalar or `n`-element column vector, and returns `v` and `beta` such that `eye(n,n) - beta*v*v'` is a Householder matrix. A Householder matrix `H` satisfies the relationship

```
H*x = -sign(x(1))*norm(x)*e1
```

where `e1` is the first column of `eye(n,n)`. Note that if `x` is complex, then `sign(x)` `exp(i*arg(x))` (which equals `x./abs(x)` when `x` is nonzero).

If `x = 0`, then `v = 0` and `beta = 1`.

integerdata — Array of arbitrary data from uniform distribution on specified range of integers

`A = gallery('integerdata',imax,[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `1:imax`. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('integerdata', ...)` with different values of `J` will return different arrays. Repeated calls to `gallery('integerdata',...)` with the same `imax`, size vector and `j` inputs will always return the same array.

In any call to `gallery('integerdata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('integerdata',7,[1,2,3,4],5)` is equivalent to `gallery('integerdata',7,1,2,3,4,5)`.

`A = gallery('integerdata',[imin imax],[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `imin:imax`.

`[A,B,...] = gallery('integerdata',[imin imax],[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('integerdata',[imin imax],[m,n,...],j,classname)` produces an array of class `classname`. `classname` must be `'uint8'`, `'uint16'`, `'uint32'`, `'int8'`, `'int16'`, `int32'`, `'single'` or `'double'`.

invhess — Inverse of an upper Hessenberg matrix

`A = gallery('invhess',x,y)`, where `x` is a length `n` vector and `y` is a length `n-1` vector, returns the matrix whose lower triangle agrees with that of `ones(n,1)*x'` and whose strict upper triangle agrees with that of `[1 y]*ones(1,n)`.

The matrix is nonsingular if `x(1) ~= 0` and `x(i+1) ~= y(i)` for all `i`, and its inverse is an upper Hessenberg matrix. Argument `y` defaults to `-x(1:n-1)`.

If `x` is a scalar, `invhess(x)` is the same as `invhess(1:x)`.

invol — Involutory matrix

`A = gallery('invol',n)` returns an `n`-by-`n` involutory (`A*A = eye(n)`) and ill-conditioned matrix. It is a diagonally scaled version of `hilb(n)`.

`B = (eye(n)-A)/2` and `B = (eye(n)+A)/2` are idempotent (`B*B = B`).

ipjfact — Hankel matrix with factorial elements

`[A,d] = gallery('ipjfact',n,k)` returns `A`, an `n`-by-`n` Hankel matrix, and `d`, the determinant of `A`, which is known explicitly. If `k = 0` (the default), then the elements of `A` are `A(i,j) = (i+j)!` If `k = 1`, then the elements of `A` are `A(i,j) 1/(i+j)`.

Note that the inverse of `A` is also known explicitly.

jordbloc — Jordan block

`A = gallery('jordbloc',n,lambda)` returns the `n`-by-`n` Jordan block with eigenvalue `lambda`. The default value for `lambda` is `1`.

kahan — Upper trapezoidal matrix

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is `n`-by-`n`. The useful range of `theta` is `0 < theta < pi`, with a default value of `1.2`.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is `25`, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

kms — Kac-Murdock-Szego Toeplitz matrix

`A = gallery('kms',n,rho)` returns the `n`-by-`n` Kac-Murdock-Szego Toeplitz matrix such that `A(i,j) = rho^(abs(i-j))`, for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L inv(gallery('triw',n,-rho,1))'`, and `D(i,i) (1-abs(rho)^2)*eye(n)`, except `D(1,1) = 1`.
- Positive definite if and only if `0 < abs(rho) < 1`.
- The inverse `inv(A)` is tridiagonal.

krylov — Krylov matrix

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

```
[x, Ax, A^2x, ..., A^(j-1)x]
```

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',randn(n))`.

lauchli — Rectangular matrix

`A = gallery('lauchli',n,mu)` returns the `(n+1)`-by-`n` matrix

```
[ones(1,n); mu*eye(n)]
```

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming `A'*A`. Argument `mu` defaults to `sqrt(eps)`.

lehmer — Symmetric positive definite matrix

`A = gallery('lehmer',n)` returns the symmetric positive definite `n`-by-`n` matrix such that `A(i,j) = i/j` for `j >= i`.

The Lehmer matrix `A` has these properties:

- `A` is totally nonnegative.
- The inverse `inv(A)` is tridiagonal and explicitly known.

- The order `n <= cond(A) <= 4*n*n`.

leslie — Matrix of birth numbers and survival rates

`L = gallery('leslie',a,b)` is the `n`-by-`n` matrix from the Leslie population model with average birth numbers `a(1:n)` and survival rates `b(1:n-1)`. It is zero, apart from the first row (which contains the `a(i)`) and the first subdiagonal (which contains the `b(i)`). For a valid model, the `a(i)` are nonnegative and the `b(i)` are positive and bounded by 1, i.e., `0 < b(i) <= 1`.

`L = gallery('leslie',n)` generates the Leslie matrix with `a = ones(n,1)`, `b = ones(n-1,1)`.

lesp — Tridiagonal matrix with real, sensitive eigenvalues

`A = gallery('lesp',n)` returns an `n`-by-`n` matrix whose eigenvalues are real and smoothly distributed in the interval approximately `[-2*N-3.5, -4.5]`.

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with `D = diag(1!,2!,...,n!)`.

lotkin — Lotkin matrix

`A = gallery('lotkin',n)` returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix `A` is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

minij — Symmetric positive definite matrix

`A = gallery('minij',n)` returns the `n`-by-`n` symmetric positive definite matrix with `A(i,j) = min(i,j)`.

The `minij` matrix has these properties:

- The inverse `inv(A)` is tridiagonal and equal to `-1` times the second difference matrix, except its `(n,n)` element is `1`.

- Givens' matrix, `2*A-ones(size(A))`, has tridiagonal inverse and eigenvalues `0.5*sec((2*r-1)*pi/(4*n))^2`, where `r=1:n`.
- `(n+1)*ones(size(A))-A` has elements that are `max(i,j)` and a tridiagonal inverse.

moler — Symmetric positive definite matrix

`A = gallery('moler',n,alpha)` returns the symmetric positive definite `n`-by-`n` matrix `U'*U`, where `U = gallery('triw',n,alpha)`.

For the default `alpha` = `-1`, `A(i,j) = min(i,j)-2`, and `A(i,i) = i`. One of the eigenvalues of `A` is small.

neumann — Singular matrix from the discrete Neumann problem (sparse)

`C = gallery('neumann',n)` returns the sparse `n`-by-`n` singular, row diagonally dominant matrix resulting from discretizing the Neumann problem with the usual five-point operator on a regular mesh. Argument `n` is a perfect square integer $n = m^2$ or a two-element vector. `C` is sparse and has a one-dimensional null space with null vector `ones(n,1)`.

normaldata — Array of arbitrary data from standard normal distribution

`A = gallery('normaldata',[m,n,...],j)` returns an `m`-by-`n`-by-... array `A`. The values of `A` are a random sample from the standard normal distribution. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('normaldata', ...)` with different values of `j` will return different arrays. Repeated calls to `gallery('normaldata',...)` with the same size vector and `j` inputs will always return the same array.

In any call to `gallery('normaldata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('normaldata', [1,2,3,4],5)` is equivalent to `gallery('normaldata',1,2,3,4,5)`.

`[A,B,...] = gallery('normaldata',[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('normaldata',[m,n,...],j, classname)` produces a matrix of class `classname`. `classname` must be either `'single'` or `'double'`.

Generate the arbitrary 6-by-4 matrix of data from the standard normal distribution `N(0, 1)` corresponding to `j = 2`:.

```
x = gallery('normaldata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the standard normal distribution `N(0, 1)` corresponding to `j = 17`:.

```
y = gallery('normaldata', 1, 2, 3, 17, 'single');
```

orthog — Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the kth type of matrix of order `n`, where `k > 0` selects exactly orthogonal matrices, and `k < 0` selects diagonal scalings of orthogonal matrices. Available types are:

`k = 1` `Q(i,j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))`

Symmetric eigenvector matrix for second difference matrix. This is the default.

`k = 2` `Q(i,j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))`

Symmetric.

`k = 3` `Q(r,s) = exp(2*pi*i*(r-1)*(s-1)/n) / sqrt(n)`

Unitary, the Fourier matrix. Q^4 is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)!`

`k = 4` Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.

`k = 5` `Q(i,j) = sin(2*pi*(i-1)*(j-1)/n) + cos(2*pi*(i-1)*(j-1)/n)`

Symmetric matrix arising in the Hartley transform.

`k = 6` `Q(i,j) = sqrt(2/n)*cos((i-1/2)*(j-1/2)*pi/n)`

Symmetric matrix arising as a discrete cosine transform.

`k = -1` `Q(i,j) = cos((i-1)*(j-1)*pi/(n-1))`

Chebyshev Vandermonde-like matrix, based on extrema of `T(n-1)`.

`k = -2` `Q(i,j) = cos((i-1)*(j-1/2)*pi/n)`

Chebyshev Vandermonde-like matrix, based on zeros of `T(n)`.

parter — Toeplitz matrix with singular values near pi

`C = gallery('parter',n)` returns the matrix `C` such that `C(i,j) = 1/(i-j+0.5)`.

`C` is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of `C` are very close to `pi`.

pei — Pei matrix

`A = gallery('pei',n,alpha)`, where `alpha` is a scalar, returns the symmetric matrix `alpha*eye(n) + ones(n)`. The default for `alpha` is 1. The matrix is singular for `alpha` equal to either `0` or `-n`.

poisson — Block tridiagonal matrix from Poisson's equation (sparse)

`A = gallery('poisson',n)` returns the block tridiagonal (sparse) matrix of order `n^2` resulting from discretizing Poisson's equation with the 5-point operator on an `n`-by-`n` mesh.

prolate — Symmetric, ill-conditioned Toeplitz matrix

`A = gallery('prolate',n,w)` returns the `n`-by-`n` prolate matrix with parameter `w`. It is a symmetric Toeplitz matrix.

If `0 < w < 0.5` then `A` is positive definite

- The eigenvalues of `A` are distinct, lie in `(0,1)`, and tend to cluster around `0` and `1`.
- The default value of `w` is 0.25.

randcolu — Random matrix with normalized cols and specified singular values

`A = gallery('randcolu',n)` is a random `n`-by-`n` matrix with columns of unit 2-norm, with random singular values whose squares are from a uniform distribution.

`A'*A` is a correlation matrix of the form produced by `gallery('randcorr',n)`.

`gallery('randcolu',x)` where `x` is an `n`-vector (`n` > 1), produces a random `n`-by-`n` matrix having singular values given by the vector `x`. The vector `x` must have nonnegative elements whose sum of squares is `n`.

`gallery('randcolu',x,m)` where `m >= n`, produces an `m`-by-`n` matrix.

`gallery('randcolu',x,m,k)` provides a further option:

`k = 0`	`diag(x)` is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have zero entries.

For more information, see [4].

randcorr — Random correlation matrix with specified eigenvalues

`gallery('randcorr',n)` is a random `n`-by-`n` correlation matrix with random eigenvalues from a uniform distribution. A correlation matrix is a symmetric positive semidefinite matrix with `1`s on the diagonal (see `corrcoef`).

`gallery('randcorr',x)` produces a random correlation matrix having eigenvalues given by the vector `x`, where `length(x) > 1`. The vector `x` must have nonnegative elements summing to `length(x)`.

`gallery('randcorr',x,k)` provides a further option:

`k = 0`	The diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have some zero entries.

For more information, see [3] and [4].

randhess — Random, orthogonal upper Hessenberg matrix

`H = gallery('randhess',n)` returns an `n`-by-`n` real, random, orthogonal upper Hessenberg matrix.

`H = gallery('randhess',x)` if `x` is an arbitrary, real, length `n` vector with `n > 1`, constructs `H` nonrandomly using the elements of `x` as parameters.

Matrix `H` is constructed via a product of `n-1` Givens rotations.

randjorth — Random J-orthogonal matrix

`A = gallery('randjorth', n)`, for a positive integer `n`, produces a random `n`-by-`n` J-orthogonal matrix `A`, where

- `J = blkdiag(eye(ceil(n/2)),-eye(floor(n/2)))`
- `cond(A) = sqrt(1/eps)`

J-orthogonality means that A'*J*A = J. Such matrices are sometimes called *hyperbolic*.

`A = gallery('randjorth', n, m)`, for positive integers `n` and `m`, produces a random (`n`+`m`)-by-(`n`+`m`) J-orthogonal matrix `A`, where

- `J = blkdiag(eye(n),-eye(m))`
- `cond(A) = sqrt(1/eps)`

```
A = gallery('randjorth',n,m,c,symm,method)
```

uses the following optional input arguments:

- `c` — Specifies `cond(A)` to be the scalar `c`.
- `symm` — Enforces symmetry if the scalar `symm` is nonzero.
- `method` — calls `qr` to perform the underlying orthogonal transformations if the scalar `method` is nonzero. A call to `qr` is much faster than the default method for large dimensions

rando — Random matrix composed of elements -1, 0 or 1

`A = gallery('rando',n,k)` returns a random `n`-by-`n` matrix with elements from one of the following discrete distributions:

`k = 1`	`A(i,j)` = 0 or 1 with equal probability (default).
`k = 2`	`A(i,j)` = -1 or 1 with equal probability.

`k = 3`	`A(i,j)` = `-1`, `0` or `1` with equal probability.

Argument `n` may be a two-element vector, in which case the matrix is `n(1)`-by-`n(2)`.

randsvd — Random matrix with preassigned singular values

`A = gallery('randsvd',n,kappa,mode,kl,ku)` returns a banded (multidiagonal) random matrix of order `n` with `cond(A) = kappa` and singular values from the distribution mode. If `n` is a two-element vector, `A` is `n(1)`-by-`n(2)`.

Arguments `kl` and `ku` specify the number of lower and upper off-diagonals, respectively, in `A`. If they are omitted, a full matrix is produced. If only `kl` is present, `ku` defaults to `kl`.

Distribution `mode` can be:

`1`	One large singular value.
`2`	One small singular value.
`3`	Geometrically distributed singular values (default).
`4`	Arithmetically distributed singular values.
`5`	Random singular values with uniformly distributed logarithm.
`< 0`	If `mode` is `-1`, `-2`, `-3`, `-4`, or `-5`, then `randsvd` treats `mode` as `abs(mode)`, except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number `kappa` defaults to `sqrt(1/eps)`. In the special case where `kappa < 0`, `A` is a random, full, symmetric, positive definite matrix with `cond(A) = -kappa` and eigenvalues distributed according to `mode`. Arguments `kl` and `ku`, if present, are ignored.

`A = gallery('randsvd',n,kappa,mode,kl,ku,method)` specifies how the computations are carried out. `method = 0` is the default, while `method = 1` uses an alternative method that is much faster for large dimensions, even though it uses more flops.

redheff — Redheffer's matrix of 1s and 0s

`A = gallery('redheff',n)` returns an `n`-by-`n` matrix of `0`'s and `1`'s defined by `A(i,j) = 1`, if `j = 1` or if `i` divides `j`, and `A(i,j) = 0` otherwise.

The Redheffer matrix has these properties:

- `(n-floor(log2(n)))-1` eigenvalues equal to `1`
- A real eigenvalue (the spectral radius) approximately `sqrt(n)`
- A negative eigenvalue approximately `-sqrt(n)`
- The remaining eigenvalues are provably "small."
- The Riemann hypothesis is true if and only if $\det(A) = O(n^{1/2+\varepsilon})$ for every $\varepsilon >$ `0`.

Barrett and Jarvis conjecture that "the small eigenvalues all lie inside the unit circle `abs(Z) = 1`," and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as `n` tends to infinity, would yield a new proof of the prime number theorem.

riemann — Matrix associated with the Riemann hypothesis

`A = gallery('riemann',n)` returns an `n`-by-`n` matrix for which the Riemann hypothesis is true if and only if

$$\det(A) = O(n!n^{-1/2+\varepsilon})$$

for every $\varepsilon > 0$.

The Riemann matrix is defined by:

```
A = B(2:n+1,2:n+1)
```

where `B(i,j) = i-1` if `i` divides `j`, and `B(i,j) = -1` otherwise.

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `abs(e(i)) <= m-1/m`, where `m = n+1`.
- `i <= e(i) <= i+1` with at most `m-sqrt(m)` exceptions.
- All integers in the interval `(m/3, m/2]` are eigenvalues.

ris — Symmetric Hankel matrix

`A = gallery('ris',n)` returns a symmetric `n`-by-`n` Hankel matrix with elements

```
A(i,j) = 0.5/(n-i-j+1.5)
```

The eigenvalues of `A` cluster around $\pi/2$ and $-\pi/2$. This matrix was invented by F.N. Ris.

sampling — Nonsymmetric matrix with ill-conditioned integer eigenvalues.

`A = gallery('sampling',x)`, where `x` is an `n`-vector, is the `n`-by-`n` matrix with `A(i,j) = X(i)/(X(i)-X(j))` for `i ~= j` and `A(j,j)` the sum of the off-diagonal elements in column `j`. A has eigenvalues `0:n-1`. For the eigenvalues 0 and `n`–1, corresponding eigenvectors are `X` and `ones(n,1)`, respectively.

The eigenvalues are ill-conditioned. `A` has the property that `A(i,j) + A(j,i) = 1` for `i ~= j`.

Explicit formulas are available for the left eigenvectors of `A`. For scalar `n`, `sampling(n)` is the same as `sampling(1:n)`. A special case of this matrix arises in sampling theory.

smoke — Complex matrix with a 'smoke ring' pseudospectrum

`A = gallery('smoke',n)` returns an `n`-by-`n` matrix with `1`'s on the superdiagonal, `1` in the `(n,1)` position, and powers of roots of unity along the diagonal.

`A = gallery('smoke',n,1)` returns the same except that element `A(n,1)` is zero.

The eigenvalues of `gallery('smoke',n,1)` are the `n`th roots of unity; those of `gallery('smoke',n)` are the `n`th roots of unity times `2^(1/n)`.

toeppd — Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd',n,m,w,theta)` returns an `n`-by-`n` symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of `m` rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

```
T = w(1)*T(theta(1)) + ... + w(m)*T(theta(m))
```

where `T(theta(k))` has `(i,j)` element `cos(2*pi*theta(k)*(i-j))`.

By default: `m = n`, `w = rand(m,1)`, and `theta = rand(m,1)`.

toeppen — Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen',n,a,b,c,d,e)` returns the `n`-by-`n` sparse, pentadiagonal Toeplitz matrix with the diagonals: `P(3,1) = a`, `P(2,1) = b`, `P(1,1) = c`, `P(1,2) = d`, and `P(1,3) = e`, where `a`, `b`, `c`, `d`, and `e` are scalars.

By default, `(a,b,c,d,e) = (1,-10,0,10,1)`, yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment `2*cos(2*t) + 20*i*sin(t)`.

tridiag — Tridiagonal matrix (sparse)

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)-1`.

`A = gallery('tridiag',n,c,d,e)`, where `c`, `d`, and `e` are all scalars, yields the Toeplitz tridiagonal matrix of order `n` with subdiagonal elements `c`, diagonal elements `d`, and superdiagonal elements `e`. This matrix has eigenvalues

```
d + 2*sqrt(c*e)*cos(k*pi/(n+1))
```

where `k = 1:n`. (see [1].)

`A = gallery('tridiag',n)` is the same as `A = gallery('tridiag',n,-1,2,-1)`, which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

triw — Upper triangular matrix discussed by Wilkinson and others

`A = gallery('triw',n,alpha,k)` returns the upper triangular matrix with ones on the diagonal and `alpha`s on the first `k >= 0` superdiagonals.

Order `n` may be a 2-element vector, in which case the matrix is `n(1)`-by-`n(2)` and upper trapezoidal.

Ostrowski ["On the Spectrum of a One-parametric Family of Matrices," *J. Reine Angew. Math.,* 1954] shows that

```
cond(gallery('triw',n,2)) = cot(pi/(4*n))^2,
```

and, for large `abs(alpha)`, `cond(gallery('triw',n,alpha))` is approximately `abs(alpha)^n*sin(pi/(4*n-2))`.

Adding `-2^(2-n)` to the `(n,1)` element makes `triw(n)` singular, as does adding `-2^(1-n)` to all the elements in the first column.

uniformdata — Array of arbitrary data from standard uniform distribution

`A = gallery('uniformdata',[m,n,...],j)` returns an `m`-by-`n`-by-... array `A`. The values of `A` are a random sample from the standard uniform distribution. j must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('uniformdata', ...)` with different values of `j` will return different arrays. Repeated calls to `gallery('uniformdata',...)` with the same size vector and `j` inputs will always return the same array.

In any call to `gallery('uniformdata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('uniformdata', [1,2,3,4],5)` is equivalent to `gallery('uniformdata',1,2,3,4,5)`.

`[A,B,...] = gallery('uniformdata',[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('uniformdata',[m,n,...],j, classname)` produces a matrix of class `classname`. `classname` must be either `'single'` or `'double'`.

Generate the arbitrary 6-by-4 matrix of data from the uniform distribution on [0, 1] corresponding to `j = 2`.

```
x = gallery('uniformdata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the uniform distribution on [0, 1] corresponding to `j = 17`.

```
y = gallery('uniformdata', 1, 2, 3, 17, 'single');
```

wathen — Finite element matrix (sparse, random entries)

`A = gallery('wathen',nx,ny)` returns a sparse, random, `n`-by-`n` finite element matrix where `n = 3*nx*ny + 2*nx + 2*ny + 1`.

Matrix `A` is precisely the "consistent mass matrix" for a regular `nx`-by-`ny` grid of 8-node (serendipity) elements in two dimensions. `A` is symmetric, positive definite for any (positive) values of the "density," `rho(nx,ny)`, which is chosen randomly in this routine.

`A = gallery('wathen',nx,ny,1)` returns a diagonally scaled matrix such that

```
0.25 <= eig(inv(D)*A) <= 4.5
```

where `D = diag(diag(A))` for any positive integers `nx` and `ny` and any densities `rho(nx,ny)`.

wilk — Various matrices devised or discussed by Wilkinson

`gallery('wilk',n)` returns a different matrix or linear system depending on the value of `n`.

`n = 3`	Upper triangular system `Ux=b` illustrating inaccurate solution.
`n = 4`	Lower triangular system `Lx=b`, ill-conditioned.
`n = 5`	`hilb(6)(1:5,2:6)*1.8144`. A symmetric positive definite matrix.
`n = 21`	`W21+`, a tridiagonal matrix. eigenvalue problem. For more detail, see [2].

References

[1] The *MATLAB* gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of Manchester, Manchester, England. Further background can be found in the books *MATLAB Guide, Second Edition*, Desmond J. Higham and Nicholas J. Higham, SIAM, 2005, and *Accuracy and Stability of Numerical Algorithms,* Nicholas J. Higham, SIAM, 1996.

[2] Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965, p.308.

[3] Bendel, R. B. and M. R. Mickey, "Population Correlation Matrices for Sampling Experiments," *Commun. Statist. Simulation Comput.*, B7, 1978, pp. 163-182.

[4] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

See Also

`hadamard` | `magic` | `hilb` | `invhilb` | `wilkinson`

Introduced before R2006a

gamma

Gamma function

Syntax

```
Y = gamma(X)
```

Definitions

The `gamma` function is defined by the integral:

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

The `gamma` function interpolates the `factorial` function. For integer `n`:

```
gamma(n+1) = n! = prod(1:n)
```

Description

`Y = gamma(X)` returns the `gamma` function at the elements of `X`. `X` must be real.

More About

Algorithms

The computation of `gamma` is based on algorithms outlined in [1]. Several different minimax rational approximations are used depending upon the value of `A`.

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

See Also

`gammainc` | `gammaincinv` | `gammaln` | `psi`

Introduced before R2006a

gammainc

Incomplete gamma function

Syntax

```
Y = gammainc(X,A)
Y = gammainc(X,A,tail)
Y = gammainc(X,A,'scaledlower')
Y = gammainc(X,A,'scaledupper')
```

Definitions

The incomplete gamma function is

$$P(a,x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt,$$

where $\Gamma(a)$ is the gamma function, `gamma(a)`.

Note: The syntax `gammainc(X,A)` is equivalent to the function `P(A,X)` defined above, where `X` is the limit of integration in each case.

For any `A` ≥ 0, `gammainc(X,A)` approaches 1 as `X` approaches infinity. For small `X` and `A`, `gammainc(X,A)` is approximately equal to `X^A`, so `gammainc(0,0) = 1`.

Description

`Y = gammainc(X,A)` returns the incomplete `gamma` function of corresponding elements of `X` and `A`. The elements of `A` must be nonnegative. Furthermore, `X` and `A` must be real and the same size (or either can be scalar).

`Y = gammainc(X,A,tail)` specifies the tail of the incomplete `gamma` function. The choices for `tail` are `'lower'` (the default) and `'upper'`. The upper incomplete `gamma` function is defined as:

$$Q(a,x) = \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{a-1} dt = 1 - P(a,x).$$

When the upper tail value is close to 0, the `'upper'` option provides a way to compute that value more accurately than by subtracting the lower tail value from 1.

`Y = gammainc(X,A,'scaledlower')` and `Y = gammainc(X,A,'scaledupper')` return the incomplete gamma function, scaled by

$$\Gamma(a+1)\left(\frac{e^x}{x^a}\right).$$

These functions are unbounded above, but are useful for values of `X` and `A` where `gammainc(X,A,'lower')` or `gammainc(X,A,'upper')` underflow to zero.

Note: When `X` is negative, `Y` can be inaccurate for `abs(X)>A+1`. This applies to all syntaxes.

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

See Also

`gamma` | `gammaincinv` | `psi` | `gammaln`

Introduced before R2006a

gammaincinv

Inverse incomplete gamma function

Syntax

```
x = gammaincinv(y,a)
x = gammaincinv(y,a,tail)
```

Description

`x = gammaincinv(y,a)` evaluates the inverse incomplete gamma function for corresponding elements of `y` and `a`, such that `y = gammainc(x,a)`. The elements of `y` must be in the closed interval `[0,1]`, and those of `a` must be nonnegative. `y` and `a` must be real and the same size (or either can be a scalar).

`x = gammaincinv(y,a,tail)` specifies the tail of the incomplete gamma function. Choices are `'lower'` (the default) to use the integral from 0 to `x`, or `'upper'` to use the integral from `x` to infinity.

These two choices are related as:

`gammaincinv(y,a,'upper') = gammaincinv(1-y,a,'lower')`.

When `y` is close to 0, the `'upper'` option provides a way to compute `x` more accurately than by subtracting `y` from 1.

Definitions

The lower incomplete gamma function is defined as:

$$\text{gammainc}(x,a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{(a-1)} dt$$

The upper incomplete gamma function is defined as:

$$\text{gammainc}(x,a) = \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{(a-1)} dt$$

`gammaincinv` computes the inverse of the incomplete gamma function with respect to the integration limit `x` using Newton's method.

For any `a>0`, as `y` approaches 1, `gammaincinv(y,a)` approaches infinity. For small `x` and `a`, `gammainc(x,a)` $\cong x^a$, so `gammaincinv(1,0) = 0`.

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

See Also

`gamma` | `gammainc` | `psi` | `gammaln`

gammaln

Logarithm of gamma function

Syntax

```
Y = gammaln(A)
```

Description

`Y = gammaln(A)` returns the logarithm of the `gamma` function, `gammaln(A) = log(gamma(A))`. Input `A` must be nonnegative and real. The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using `log(gamma(A))`.

See Also

`gammainc` | `gamma` | `psi` | `gammaincinv`

Introduced before R2006a

gca

Current axes handle

Syntax

```
ax = gca
```

Description

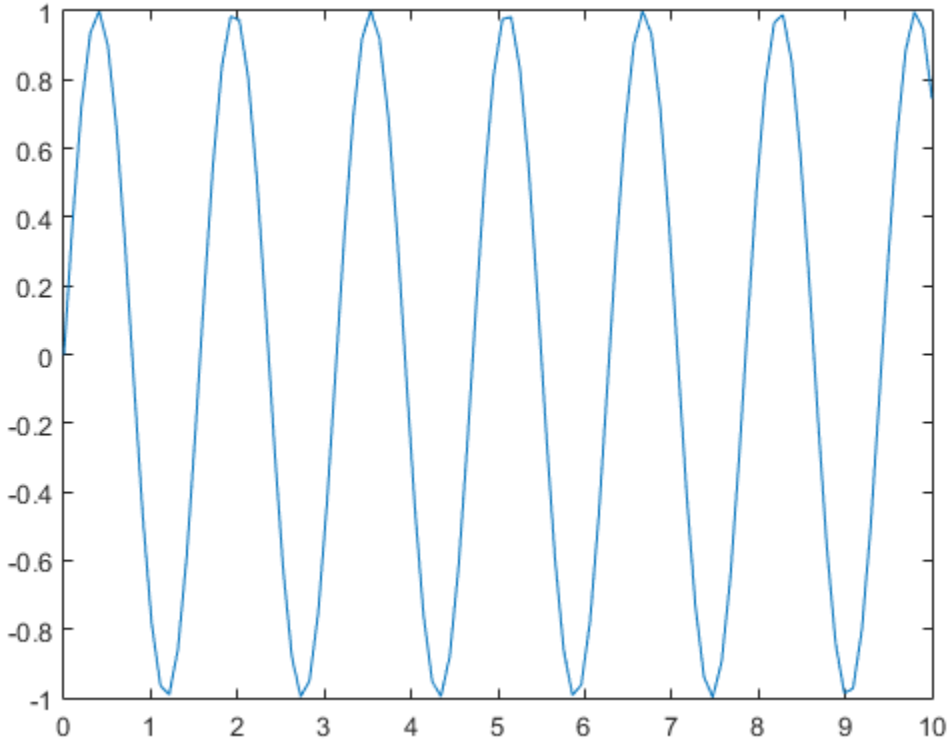
`ax = gca` returns the handle to the current axes for the current figure. If an axes does not exist, then `gca` creates an axes and returns its handle. You can use the axes handle to query and modify axes properties. For more information, see Axes Properties.

Examples

Specify Properties for Current Axes

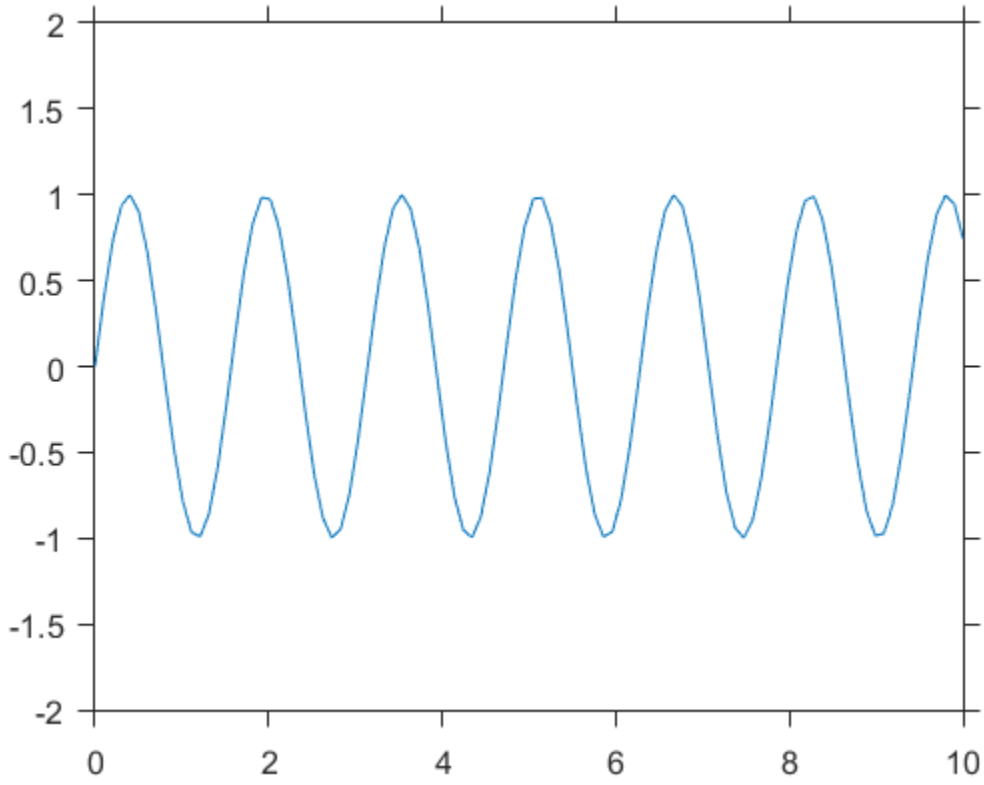
Plot a sine wave.

```
x = linspace(0,10);
y = sin(4*x);
plot(x,y)
```



Set the font size, tick direction, tick length, and *y*-axis limits for the current axes. Use `gca` to refer to the current axes. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
ax = gca; % current axes
ax.FontSize = 12;
ax.TickDir = 'out';
ax.TickLength = [0.02 0.02];
ax.YLim = [-2 2];
```



More About

Current Axes

The current axes is the target for graphics output. It is the axes in which graphics commands such as `plot`, `text`, and `surf` draw their results. It is typically the last axes created or the last axes clicked with the mouse. Changing the current figure also changes the current axes.

User interaction can change the current axes. If you need to access a specific axes, store the axes handle in your program code when you create the axes and use this handle instead of `gca`.

Tips

- To get the handle of the current axes without forcing the creation of an axes if one does not exist, query the figure `CurrentAxes` property.

  ```
  fig = gcf;
  ax = fig.CurrentAxes;
  ```
  MATLAB returns `ax` as an empty array if there is no current axes.
- Set axes properties after plotting since some plotting functions reset axes properties.

See Also

Functions
`axes` | `cla` | `findobj` | `gcf` | `get` | `set`

Properties
Axes Properties

Introduced before R2006a

gcbf

Handle of figure containing object whose callback is executing

Syntax

```
fig = gcbf
```

Description

`fig = gcbf` returns the handle of the figure that contains the object whose callback is currently executing. This object can be the figure itself, in which case, `gcbf` returns the figure's handle.

When no callback is executing, `gcbf` returns the empty matrix, `[]`.

The value returned by `gcbf` is identical to the `figure` output argument returned by `gcbo`.

See Also

`gcbo` | `gco` | `gcf` | `gca`

Introduced before R2006a

gcbo

Handle of object whose callback is executing

Syntax

```
h = gcbo
[h,figure] = gcbo
```

Description

`h = gcbo` returns the handle of the graphics object whose callback is executing.

`[h,figure] = gcbo` returns the handle of the current callback object and the handle of the figure containing this object.

More About

Tips

The MATLAB software stores the handle of the object whose callback is executing in the root `CallbackObject` property. If a callback interrupts another callback, MATLAB replaces the `CallbackObject` value with the handle of the object whose callback is interrupting. When that callback completes, MATLAB restores the handle of the object whose callback was interrupted.

The root `CallbackObject` property is read only, so its value is always valid at any time during callback execution. The root `CurrentFigure` property, and the figure `CurrentAxes` and `CurrentObject` properties (returned by `gcf`, `gca`, and `gco`, respectively) are user settable, so they can change during the execution of a callback, especially if that callback is interrupted by another callback. Therefore, those functions are not reliable indicators of which object's callback is executing.

When you write callback routines for the `CreateFcn` and `DeleteFcn` of any object and the figure `ResizeFcn`, you must use `gcbo` since those callbacks do not update the root's `CurrentFigure` property, or the figure's `CurrentObject` or `CurrentAxes` properties; they only update the root's `CallbackObject` property.

When no callbacks are executing, `gcbo` returns `[]` (an empty matrix).

See Also

`gca` | `gcf` | `gco` | `groot`

Introduced before R2006a

gcd

Greatest common divisor

Syntax

```
G = gcd(A,B)
[G,U,V] = gcd(A,B)
```

Description

`G = gcd(A,B)` returns the greatest common divisors of the elements of `A` and `B`. The elements in `G` are always nonnegative, and `gcd(0,0)` returns `0`. This syntax supports inputs of any numeric type.

`[G,U,V] = gcd(A,B)` also returns the Bézout coefficients, `U` and `V`, which satisfy: `A.*U + B.*V = G`. The Bézout coefficients are useful for solving Diophantine equations. This syntax supports double, single, and signed integer inputs.

Examples

Greatest Common Divisors of Double Values

```
A = [-5 17; 10 0];
B = [-15 3; 100 0];
G = gcd(A,B)
```

```
G =

     5     1
    10     0
```

`gcd` returns positive values, even when the inputs are negative.

Greatest Common Divisors of Unsigned Integers

```
A = uint16([255 511 15]);
B = uint16([15 127 1023]);
```

```
G = gcd(A,B)

G =

     15      1      3
```

Solution to Diophantine Equation

Solve the Diophantine equation, $30x + 56y = 8$ for x and y.

Find the greatest common divisor and a pair of Bézout coefficients for `30` and `56`.

```
[g,u,v] = gcd(30,56)

g =

     2

u =

   -13

v =

     7
```

`u` and `v` satisfy the Bézout's identity, `(30*u) + (56*v) = g`.

Rewrite Bézout's identity so that it looks more like the original equation. Do this by multiplying by `4`. Use `==` to verify that both sides of the equation are equal.

```
(30*u*4) + (56*v*4) == g*4

ans =

   1
```

Calculate the values of x and y that solve the problem.

```
x = u*4
y = v*4

x =

   -52
```

```
y =

    28
```

Input Arguments

A,B — Input values

scalars, vectors, or arrays of real integer values

Input values, specified as scalars, vectors, or arrays of real integer values. A and B can be any numeric type, and they can be of different types within certain limitations:

- If A or B is of type `single`, then the other can be of type `single` or `double`.
- If A or B belongs to an integer class, then the other must belong to the same class or it must be a `double` scalar value.

A and B must be the same size or one must be a scalar.

Example: `[20 -3 13],[10 6 7]`

Example: `int16([100 -30 200]),int16([20 15 9])`

Example: `int16([100 -30 200]),20`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Output Arguments

G — Greatest common divisor

real, nonnegative integer values

Greatest common divisor, returned as an array of real nonnegative integer values. G is the same size as A and B, and the values in G are always real and nonnegative. G is returned as the same type as A and B. If A and B are of different types, then G is returned as the nondouble type.

U,V — Bézout coefficients

real integer values

Bézout coefficients, returned as arrays of real integer values that satisfy the equation, `A.*U + B.*V = G`. The data type of `U` and `V` is the same type as that of `A` and `B`. If `A` and `B` are of different types, then `U` and `V` are returned as the nondouble type.

More About

Algorithms

`g = gcd(A,B)` is calculated using the Euclidian algorithm.[1]

`[g,u,v] = gcd(A,B)` is calculated using the extended Euclidian algorithm.[1]

References

[1] Knuth, D. "Algorithms A and X." *The Art of Computer Programming, Vol. 2*, Section 4.5.2. Reading, MA: Addison-Wesley, 1973.

See Also

`lcm`

Introduced before R2006a

gcf

Current figure handle

Syntax

```
fig = gcf
```

Description

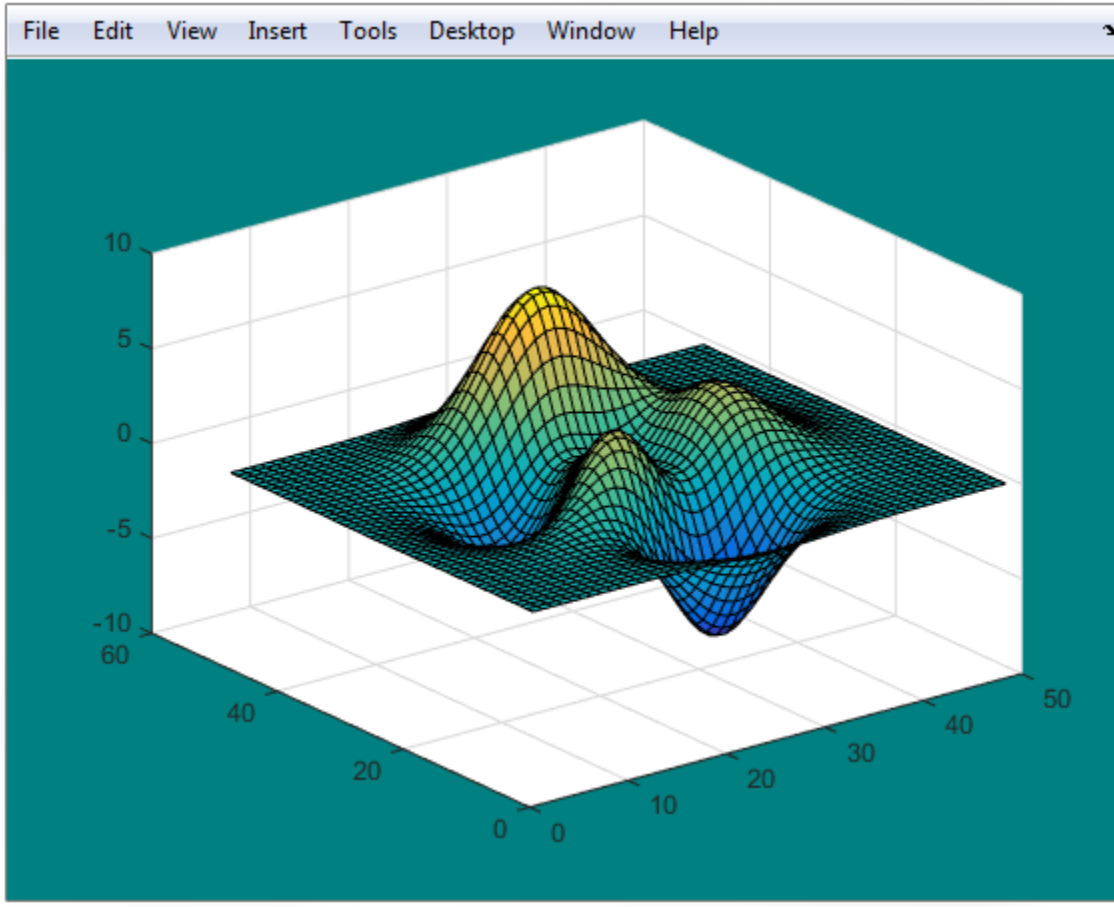
`fig = gcf` returns the current figure handle. If a figure does not exist, then `gcf` creates a figure and returns its handle. You can use the figure handle to query and modify figure properties. For more information, see Figure Properties.

Examples

Specify Properties for Current Figure

Set the background color and remove the toolbar for the current figure. Use the `gcf` command to get the current figure handle.

```
surf(peaks)
fig = gcf; % current figure handle
fig.Color = [0 0.5 0.5];
fig.ToolBar = 'none';
```



More About

Current Figure

The current figure is the target for graphics output. It is the figure window in which graphics commands such as `plot`, `title`, and `surf` draw their results. It is typically the last figure created or the last figure clicked with the mouse.

User interaction can change the current figure. If you need to access a specific figure, store the figure handle in your program code when you create the figure and use this handle instead of `gcf`.

Tips

- To get the handle of the current figure without forcing the creation of a figure if one does not exist, query the `CurrentFigure` property on the root object.

  ```
  fig = get(groot,'CurrentFigure');
  ```
  MATLAB returns `fig` as an empty array if there is no current figure.

See Also

`clf` | Figure Properties | `figure` | `gca` | `get` | `set`

Introduced before R2006a

gcmr

Get current mapreducer configuration

The `gcmr` function defines the global execution environment for `mapreduce`, and is most likely used with Parallel Computing Toolbox, MATLAB Distributed Computing Server™, or MATLAB Compiler™.

If you do not specify a configuration to use in your call to `mapreduce`, then by default `mapreduce` uses the configuration returned (or generated) by `gcmr`.

Syntax

```
mr = gcmr
mr = gcmr('nocreate')
```

Description

`mr = gcmr` returns an object representing the current global execution environment for `mapreduce`.

- If no global execution environment exists, then `gcmr` calls `mapreducer` to set the global execution environment to be the default.
- If a global execution environment currently exists, then `gcmr` returns the last visible MapReducer object created.

  When you create a MapReducer object using `mapreducer`, the object sets the global execution environment. The global execution environment persists even if the object representing it is later deleted.
- If the global execution environment is deleted or invalid, then `gcmr` returns the next visible MapReducer object available. For example, `delete(gcmr)` deletes the current global execution environment.

`mr = gcmr('nocreate')` returns the current global execution environment for `mapreduce`, if one already exists. If no global execution environment exists, then `gcmr` returns `[]`.

More About

Tips

- If you have Parallel Computing Toolbox, see the `mapreducer` function reference page for related information.
- If you have MATLAB Compiler, see the `mapreducer` function reference page for related information.
- "Speed Up and Deploy MapReduce Using Other Products"

See Also

`mapreduce` | `mapreducer`

Introduced in R2014b

gco

Handle of current object

Syntax

```
h = gco
h = gco(figure_handle)
```

Description

`h = gco` returns the handle of the current object.

`h = gco(figure_handle)` returns the handle of the current object in the figure specified by `figure_handle`.

More About

Tips

The current object is the last object clicked or selected via keyboard interaction, excluding `uimenu`s. If the mouse click did not occur over a figure child object, the figure becomes the current object. The MATLAB software stores the handle of the current object in the figure's `CurrentObject` property.

An object can become the current object as a result of pressing the space bar to invoke a callback in a dialog when a uicontrol in that dialog has focus (usually the result of using the **Tab** key to change focus).

The `CurrentObject` of the `CurrentFigure` does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the `CurrentObject` or even the `CurrentFigure`. Some callbacks, such as `CreateFcn` and `DeleteFcn`, and uimenu `Callback`, intentionally do not update `CurrentFigure` or `CurrentObject`.

`gcbo` provides the only completely reliable way to retrieve the handle to the object whose callback is executing, at any point in the `callback` function, regardless of the type of callback or of any previous interruptions.

See Also

gca | gcbo | gcf

Introduced before R2006a

ge, >=

Determine greater than or equal to

Syntax

```
A >= B
ge(A,B)
```

Description

`A >= B` returns a logical array with elements set to logical 1 (`true`) where `A` is greater than or equal to `B`; otherwise, it returns logical `0` (`false`).

The test compares only the real part of numeric arrays. `ge` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`ge(A,B)` is an alternate way to execute `A >= B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Find which vector elements are greater than or equal to a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are greater than or equal to `11`.

```
A >= 11
```

```
ans =

     0     1     1     0     0     1     0     1
```

The result is a vector with values of logical 1 (`true`) where the elements of `A` satisfy the expression.

Use the vector of logical values as an index to view the values in `A` that are greater than or equal to `11`.

```
A(A >= 11)
```

```
ans =

    12    18    11    15
```

The result is a subset of the elements in `A`.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values greater than or equal to `9` with the value `10`.

```
A(A >= 9) = 10
```

```
A =

    10     2     3    10
     5    10    10     8
    10     7     6    10
     4    10    10     1
```

The result is a new matrix whose largest element is `10`.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
```

```
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)

A =

     large       medium      small
     medium      small       large
```

The array has three categories: `'small'`, `'medium'`, and `'large'`.

Find all values greater than or equal to the category `'medium'`.

```
A >= 'medium'

ans =

     1     1     0
     1     0     1
```

A value of logical `1` (`true`) indicates a value greater than or equal to the category `'medium'`.

Compare the rows of `A`.

```
A(1,:) >= A(2,:)

ans =

     1     1     0
```

The function returns logical `1` (`true`) where the first row has a category value greater than or equal to the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are greater than or equal to `2`.

```
A(A >= 2)

ans =

   2.0000 - 2.0000i   5.0000 - 1.0000i
```

`ge` compares only the real part of the elements in `A`.

Use `abs` to find which elements are outside a radius of `2` from the origin.

```
A(abs(A) >= 2)
```

```
ans =

   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i   5.0000 - 1.0000i
```

The result has more elements since `abs` accounts for the imaginary part of the numbers.

Test Duration Values

Create a `duration` array.

```
d = hours(21:25) + minutes(75)
```

```
d =

   22.25 hrs   23.25 hrs   24.25 hrs   25.25 hrs   26.25 hrs
```

Test the array for elements that are greater than or equal to one standard day.

```
d >= 1
```

```
ans =

     0     0     1     1     1
```

Input Arguments

A — Left array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Left array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

`B` — Right array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Right array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

More About

Tips

- Some floating-point numbers cannot be represented exactly in binary form. This leads to small differences in results that the `>=` operator reflects. For more information, see "Avoiding Common Problems with Floating-Point Arithmetic".
- "Ordinal Categorical Arrays"

See Also

`eq` | `gt` | `le` | `lt` | `ne`

Introduced before R2006a

genpath

Generate path string

Syntax

```
p = genpath
p = genpath(folderName)
```

Description

`p = genpath` returns a path string, `p`, that includes all the folders and subfolders below *`matlabroot`*`/toolbox`, including empty subfolders.

`p = genpath(folderName)` returns a path string that includes `folderName` and multiple levels of subfolders below `folderName`. The path string does not include folders named `private`, folders that begin with the `@` character (class folders), folders that begin with the `+` character (package folders), or subfolders within any of these.

Examples

Add Folder and Subfolders to Search Path

Use `genpath` in conjunction with `addpath` to add a folder and its subfolders to the search path.

Generate a path that includes *`matlabroot`*`/toolbox/images/colorspaces` and all folders below it.

```
folderName = fullfile(matlabroot,'toolbox','images','colorspaces');
p = genpath(folderName);
```

Add the folder and its subfolders to the search path.

```
addpath(p)
```

Input Arguments

`folderName` — Folder name

string

Folder name, specified as a string.

Example: `'c:/matlab/myfiles'`

More About

- "What Is the MATLAB Search Path?"

See Also

`addpath` | `path` | `rmpath`

Introduced before R2006a

genvarname

Construct valid variable name from string

Note: `genvarname` will be removed in a future release. Use `matlab.lang.makeValidName` and `matlab.lang.makeUniqueStrings` instead.

Syntax

```
varname = genvarname(str)
varname = genvarname(str, exclusions)
```

Description

`varname = genvarname(str)` constructs a string `varname` that is similar to or the same as the `str` input, and can be used as a valid variable name. `str` can be a single character array or a cell array of strings. If `str` is a cell array of strings, `genvarname` returns a cell array of strings in `varname`. The strings in a cell array returned by `genvarname` are guaranteed to be different from each other.

`varname = genvarname(str, exclusions)` returns a valid variable name that is different from any name listed in the `exclusions` input. The `exclusions` input can be a single character array or a cell array of strings. Specify the function `who` in the `exclusions` character array to create a variable name that will be unique in the current MATLAB workspace (see “Example 4” on page 1-2926, below).

Note `genvarname` returns a string that can be used as a variable name. It does not create a variable in the MATLAB workspace. You cannot, therefore, assign a value to the output of `genvarname`.

Examples

Example 1

Create four similar variable name strings that do not conflict with each other:

```
v = genvarname({'A', 'A', 'A', 'A'})
v =
    'A'    'A1'    'A2'    'A3'
```

Example 2

Read a column header `hdr` from worksheet `trial2` in Excel spreadsheet myproj_apr23:

```
[data hdr] = xlsread('myproj_apr23.xls', 'trial2');
```

Make a variable name from the text of the column header that will not conflict with other names:

```
v = genvarname(['Column ' hdr{1,3}]);
```

Assign data taken from the spreadsheet to the variable in the MATLAB workspace:

```
eval([v '= data(1:7, 3);']);
```

Example 3

Collect readings from an instrument once every minute over the period of an hour into different fields of a structure. Simulate instrument readings using a random number. `genvarname` not only generates unique fieldname strings, but also creates the structure and fields in the MATLAB workspace:

```
for k = 1:60
record.(genvarname(['reading' datestr(clock, 'HHMMSS')])) = rand(1);
pause(60)
end
```

After the program ends, display the recorded data from the workspace:

```
record
```

```
record =

    reading092610: 0.6541
    reading092710: 0.6892
    reading092811: 0.7482
    reading092911: 0.4505
    reading093011: 0.0838
          .
          .
          .
```

Example 4

Generate variable names that are unique in the MATLAB workspace by putting the output from the `who` function in the `exclusions` list.

```
for k = 1:5
   t = clock;
   pause(uint8(rand * 10));
   v = genvarname('time_elapsed', who);
   eval([v ' = etime(clock,t)'])
   end
```

As this code runs, you can see that the variables created by `genvarname` are unique in the workspace:

```
time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010
time_elapsed3 =
    8.0010
time_elapsed4 =
    3.0040
```

After the program completes, use the `who` function to view the workspace variables:

```
who

k           time_elapsed   time_elapsed2  time_elapsed4
t           time_elapsed1  time_elapsed3  v
```

Example 5

If you try to make a variable name from a MATLAB keyword, `genvarname` creates a variable name string that capitalizes the keyword and precedes it with the letter `x`:

```
v = genvarname('global')
v =
    xGlobal
```

Example 6

If you enter a string that is longer than the value returned by the `namelengthmax` function, `genvarname` truncates the resulting variable name string:

```
namelengthmax
ans =
    63

vstr = genvarname(sprintf('%s%s', ...
   'This name truncates because it contains ', ...
   'more than the maximum number of characters'))
vstr =
ThisNameTruncatesBecauseItContainsMoreThanTheMaximumNumberOfCha
```

More About

Tips

A valid MATLAB variable name is a character string of letters, digits, and underscores, such that the first character is a letter, and the length of the string is less than or equal to the value returned by the `namelengthmax` function. Any string that exceeds `namelengthmax` is truncated in the `varname` output. See “Example 6” on page 1-2927, below.

The variable name returned by `genvarname` is not guaranteed to be different from other variable names currently in the MATLAB workspace unless you use the `exclusions` input in the manner shown in “Example 4” on page 1-2926, below.

If you use `genvarname` to generate a field name for a structure, MATLAB does create a variable for the structure and field in the MATLAB workspace. See “Example 3” on page 1-2925, below.

If the `str` input contains any whitespace characters, `genvarname` removes then and capitalizes the next alphabetic character in `str`. If `str` contains any nonalphanumeric characters, `genvarname` translates these characters into their hexadecimal value.

See Also

`isvarname` | `iskeyword` | `isletter` | `namelengthmax` | `who` | `regexp`

Introduced before R2006a

get

Query graphics object properties

Syntax

```
v = get(h)
v = get(h,propertyName)
v = get(h,propertyArray)
v = get(h,'default')
v = get(h,defaultTypeProperty)
v = get(groot,'factory')
v = get(groot,factoryTypeProperty)
```

Description

Note: Do not use the `get` function on Java objects as it will cause a memory leak. For more information, see "Accessing Private and Public Data"

`v = get(h)` returns all properties and property values for the graphics object identified by `h`. `v` is a structure whose field names are the property names and whose values are the corresponding property values. `h` can be a single object or an `m`-by-`n` array of objects. If `h` is a single object and you do not specify an output argument, then MATLAB displays the information on the screen.

`v = get(h,propertyName)` returns the value for the specific property, `propertyName`. Use single quotes around the property name, for example, `get(h,'Color')`. If you do not specify an output argument, then MATLAB displays the information on the screen.

`v = get(h,propertyArray)` returns an *m*-by-*n* cell array, where *m* is equal to `length(h)` and *n* is equal to the number of property names contained in `propertyArray`.

`v = get(h,'default')` returns all default values currently defined on object `h` in a structure array. The field names are the object property names and the field values are

the corresponding property values. If you do not specify an output argument, MATLAB displays the information on the screen.

`v = get(h,defaultTypeProperty)` returns the current default value for a specific property. The argument `defaultTypeProperty` is the word `default` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`) in single quotes. For example, `get(groot,'defaultFigureColor')`.

`v = get(groot,'factory')` returns the factory-defined values of all user-settable properties in a structure array. The field names are the object property names and the field values are the corresponding property values. If you do not specify an output argument, MATLAB displays the information on the screen.

`v = get(groot,factoryTypeProperty)` returns the factory-defined value for a specific property. The argument `factoryTypeProperty` is the word `factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`) in single quotes. For example, `get(groot,'factoryFigureColor')`.

Examples

List All Property Values for Specific Object

Create a line plot and return the chart line object as `p`. List all the properties of the line and the current property values.

```
p = plot(1:10);
get(p)
```

```
    AlignVertexCenters: 'off'
            Annotation: [1x1 matlab.graphics.eventdata.Annotation]
          BeingDeleted: 'off'
            BusyAction: 'queue'
          ButtonDownFcn: ''
               Children: []
               Clipping: 'on'
                  Color: [0.9290 0.6940 0.1250]
             CreateFcn: ''
             DeleteFcn: ''
           DisplayName: ''
      HandleVisibility: 'on'
               HitTest: 'on'
```

```
         Interruptible: 'on'
             LineStyle: '-'
             LineWidth: 0.5000
                Marker: 'none'
       MarkerEdgeColor: 'auto'
       MarkerFaceColor: 'none'
            MarkerSize: 6
                Parent: [1x1 Axes]
         PickableParts: 'visible'
              Selected: 'off'
    SelectionHighlight: 'on'
                   Tag: ''
                  Type: 'line'
         UIContextMenu: []
              UserData: []
               Visible: 'on'
                 XData: [1 2 3 4 5 6 7 8 9 10]
             XDataMode: 'auto'
           XDataSource: ''
                 YData: [1 2 3 4 5 6 7 8 9 10]
           YDataSource: ''
                 ZData: [1x0 double]
           ZDataSource: ''
```

Query Specific Property of Specific Object

Create a line plot and return the chart line object as `p`. Use `get` to return the current value of the `LineWidth` property.

```
p = plot(1:10);
get(p,'LineWidth')
```

```
ans =

    0.5000
```

Query Set of Properties for Specific Object

Create a line plot with circle markers and return the chart line object as `p`. Use `get` to return the current values of the `LineWidth`, `Marker`, and `MarkerSize` properties for the object.

```
p = plot(1:10,'ro-');
```

```
props = {'LineWidth','Marker','MarkerSize'};
get(p,props)
```

```
ans =

    [0.5000]    'o'    [6]
```

Query Default Property Value on Root

Return the default value of the `LineWidth` property defined on the root for all line graphics objects.

```
get(groot,'DefaultLineLineWidth')
```

```
ans =

    0.5000
```

More About

- "Graphics Object Properties"

See Also

`findobj` | `gca` | `gcf` | `gco` | `set`

Introduced before R2006a

get

Query property values for `audioplayer` object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
Values = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the specified property for object `obj`.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-`N` cell array.

`Values = get(obj)` returns a scalar structure that contains the values of all properties of `obj`. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Examples

Create an `audioplayer` object from the example file `handel.mat` and query the object properties:

```
load handel.mat;
handelObj = audioplayer(y, Fs);

% Display all properties.
get(handelObj)

% Display only the SampleRate property.
get(handelObj, 'SampleRate')
```

```
% Create a cell array that contains
% values for two properties.
info = get(handelObj, {'BitsPerSample', 'NumberOfChannels'});
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `TotalSamples` property for an object named `handelObj` (as created in the Example):

```
numSamples = handelObj.TotalSamples;
```

This command is exactly equivalent to:

```
numSamples = get(handelObj, 'TotalSamples');
```

See Also

`audioplayer` | `set`

get

Query property values for `audiorecorder` object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
Values = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the specified property for object `obj`.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-`N` cell array.

`Values = get(obj)` returns a scalar structure that contains the values of all properties of `obj`. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Examples

Create an `audiorecorder` object and query the object properties:

```
recorderObj = audiorecorder;

% Display all properties.
get(recorderObj)

% Display only the SampleRate property.
get(recorderObj, 'SampleRate')

% Create a cell array that contains
% values for two properties.
```

```
info = get(recorderObj, {'BitsPerSample', 'NumberOfChannels'});
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `TotalSamples` property for an object named `recorderObj` (as created in the Example):

```
numSamples = recorderObj.TotalSamples;
```

This command is exactly equivalent to:

```
numSamples = get(recorderObj, 'TotalSamples');
```

See Also

`audiorecorder` | `set`

get (COM)

Get property value from interface, or display properties

Syntax

```
V = get(h)
V = get(h,'name')
```

Description

`V = get(h)` returns a list of all properties and their values for the object or interface, `h`.

If `V` is empty, either there are no properties in the object, or MATLAB cannot read the object's type library. Refer to the COM vendors documentation.

`V = get(h,'name')` returns value of property specified in string, `name`.

COM functions are available on Microsoft Windows systems only.

Examples

Create a COM server running Microsoft Excel software:

```
e = actxserver('Excel.Application');
```

Retrieve a single property value.

```
e.Path
```

```
ans =
   C:\Program Files\MSOffice\OFFICE11
```

MATLAB displays the path to your program.

List of all properties for the `CommandBars` interface.

```
c = CommandBars.get(e)
```

MATLAB displays information for your version of Excel.

More About

Tips

The meaning and type of the return value is dependent upon the specific property being retrieved. The object's documentation should describe the specific meaning of the return value. MATLAB may convert the data type of the return value. For a description of how MATLAB converts COM data types, see “Handling COM Data in MATLAB Software”.

See Also

`set (COM)` | `inspect` | `isprop` | `addproperty` | `deleteproperty`

Introduced before R2006a

get

Class: VideoReader

Query property values for video reader object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
allValues = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the property with the specified Name for object obj.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-N cell array.

`allValues = get(obj)` returns a scalar structure that contains the values of all properties of obj. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Input Arguments

`obj`

`VideoReader` object created by the `VideoReader` function.

`Name`

String enclosed in single quotation marks that specifies a `VideoReader` property.

Output Arguments

`Value`

String containing the value associated with the specified `VideoReader` property.

`Values`

Cell array containing the values associated with the specified `VideoReader` properties. Values is a row vector, with one column for each property.

`allValues`

Scalar (1-by-1) structure array that contains the values of all properties of `VideoReader` object obj.

Examples

Display All Object Properties

Display all properties of an object associated with the example file, `xylophone.mp4`.

```
xyloObj = VideoReader('xylophone.mp4');
get(xyloObj)
```

```
obj =

  VideoReader with properties:

   General Properties:
           Name: 'xylophone.mp4'
           Path: 'matlabroot\toolbox\matlab\audiovideo'
       Duration: 4.7000
    CurrentTime: 0
            Tag: ''
       UserData: []

   Video Properties:
          Width: 320
         Height: 240
       FrameRate: 30
    BitsPerPixel: 24
```

```
    VideoFormat: 'RGB24'
```

Store Specific Object Properties

Create a 1-by-3 cell array that contains the values of the `Height`, `Width`, and `Duration` properties of an object associated with `xylophone.mp4`.

```
xyloObj = VideoReader('xylophone.mp4');
xyloProp = get(xyloObj,{'Height','Width','Duration'})
```

```
xyloProp =

    [240]    [320]    [4.7000]
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `Duration` property for object `xyloObj` (as created in the Examples):

```
D = xyloObj.Duration;
```

This command is equivalent to:

```
D = get(xyloObj,'Duration');
```

See Also

`VideoReader` | `set`

get (RandStream)

Random stream properties

Class

`@RandStream`

Syntax

```
get(s)
P = get(s)
P = get(s,'PropertyName')
```

Description

`get(s)` prints the list of properties for the random stream `s`.

`P = get(s)` returns all properties of `s` in a scalar structure.

`P = get(s,'PropertyName')` returns the property `'PropertyName'`.

See Also

`RandStream` | `set (RandStream)`

get (serial)

Serial port object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Description

`get(obj)` returns all property names and their current values to the command line for the serial port object, `obj`.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *`PropertyName`* for `obj`. If *`PropertyName`* is replaced by a 1-by-n or n-by-1 cell array of strings containing property names, then `get` returns a 1-by-n cell array of values to `out`. If `obj` is an array of serial port objects, then `out` will be a m-by-n cell array of property values where m is equal to the length of `obj` and n is equal to the number of properties specified.

Examples

This example illustrates some of the ways you can use `get` to return property values for the serial port object `s` on a Windows platform.

```
s = serial('COM1');
out1 = get(s);
out2 = get(s,{'BaudRate','DataBits'});
s.Parity

ans =
none
```

More About

Tips

Refer to Displaying Property Names and Property Values for a list of serial port object properties that you can return with `get`.

When you specify a property name, you can do so without regard to case. For example, if `s` is a serial port object, then these commands are equivalent.

```
out = s.BaudRate;
out = s.baudrate;
```

See Also

`set`

Introduced before R2006a

get (tscollection)

Query `tscollection` object property values

Syntax

```
value = get(tsc,'PropertyName')
```

Description

`value = get(tsc,'PropertyName')` returns the value of the specified property of the `tscollection` object tsc. The following syntax is equivalent:

```
value = tsc.PropertyName
```

`get(tsc)` displays all properties and values of the `tscollection` object tsc.

See Also

`set (tscollection)` | `timeseries`

Introduced before R2006a

getabstime (tscollection)

Extract date-string time vector into cell array

Syntax

```
getabstime(tsc)
```

Description

`getabstime(tsc)` extracts the time vector from the `tscollection` object tsc as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the time-series collection. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Examples

1. Create a `tscollection` object.

   ```
   tsc = tscollection(timeseries([3 6 8 0 10]));
   ```

2. Set the `StartDate` property.

   ```
   tsc.TimeInfo.StartDate = '10/27/2005 07:05:36';
   ```

3. Extract a vector of absolute time values.

   ```
   getabstime(tsc)

   ans =

       '27-Oct-2005 07:05:36'
       '27-Oct-2005 07:05:37'
       '27-Oct-2005 07:05:38'
       '27-Oct-2005 07:05:39'
       '27-Oct-2005 07:05:40'
   ```

4. Change the date-string format of the time vector.

```
tsc.TimeInfo.Format = 'mm/dd/yy';
```

5 Extract the time vector with the new date-string format.

```
getabstime(tsc)

ans =

    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
```

See Also

datestr | setabstime (tscollection) | tscollection

Introduced before R2006a

getappdata

Retrieve application-defined data

Use the `getappdata` function to retrieve data stored using the `setappdata` function. Both of these functions provide a convenient way to share data between callbacks or between separate UIs.

Syntax

```
val = getappdata(obj,name)
vals = getappdata(obj)
```

Description

`val = getappdata(obj,name)` returns a value stored in the graphics object, obj. The name identifier, name, uniquely identifies the value to retrieve.

`vals = getappdata(obj)` returns all values stored in the graphics object with their name identifiers.

Examples

Store and Retrieve Date and Time

Create a figure window.

```
f = figure;
```

Get the current date and time as separate variables.

```
dt = fix(clock);
currdate = dt(1:3);
currtime = dt(4:6);
```

Store `currdate` and `currtime` using the `setappdata` function.

```
setappdata(f,'todaysdate',currdate);
```

```
setappdata(f,'presenttime',currtime);
```

Retrieve the date information.

```
getappdata(f,'todaysdate')
```

```
ans =

    2014  12    23
```

Retrieve all data associated with figure `f`.

```
getappdata(f)
```

```
ans =

     todaysdate: [2014 12 23]
    presenttime: [16 51 5]
```

Input Arguments

`obj` — Graphics object containing the value

figure | uipanel | uibuttongroup | uicontrol | ...

Graphics object containing the value, specified as any graphics object (except an ActiveX component). This is the same graphics object passed to `setappdata` during the storage operation.

`name` — Name identifier

`string`

Name identifier, specified as a `string` value. This is the same name identifier passed to `setappdata` during the storage operation.

Data Types: `char`

Output Arguments

`val` — Stored value

any MATLAB data type

Stored value, returned as the same value and data type that was originally stored.

`vals` — All values stored in the graphics object with name identifiers

`struct`

All values stored in the graphics object with name identifiers, returned as a `struct`. Each field in the `struct` corresponds to a stored value. The field names of the `struct` correspond to the name identifiers assigned when each value was stored.

More About

- "Share Data Among Callbacks"

See Also

`guidata` | `isappdata` | `rmappdata` | `setappdata`

Introduced before R2006a

getaudiodata

Store recorded audio signal in numeric array

Syntax

```
y = getaudiodata(recorder)
y = getaudiodata(recorder, dataType)
```

Description

`y = getaudiodata(recorder)` returns recorded audio data associated with `audiorecorder` object `recorder` to `double` array `y`.

`y = getaudiodata(recorder, dataType)` converts the signal data to the specified data type: `'double'`, `'single'`, `'int16'`, `'int8'`, or `'uint8'`.

Output Arguments

y

Audio signal data `y` contains the same number of columns as the number of channels in the recording: one for mono, two for stereo. The range of values depends on the data type, as shown in the following table.

Data Type	Sample Value Range
`int8`	-128 to 127
`uint8`	0 to 255
`int16`	-32768 to 32767
`single`	-1 to 1
`double`	-1 to 1

Examples

Collect a sample of your speech with a microphone, and plot the signal data:

```
% Record your voice for 5 seconds.
recObj = audiorecorder;
disp('Start speaking.')
recordblocking(recObj, 5);
disp('End of Recording.');

% Play back the recording.
play(recObj);

% Store data in double-precision array.
myRecording = getaudiodata(recObj);

% Plot the waveform.
plot(myRecording);
```

See Also

`audiorecorder`

How To

- “Characteristics of Audio Files”
- “Record Audio”

GetCharArray

Character array from Automation server

Syntax

IDL Method Signature

```
HRESULT GetCharArray([in] BSTR varName, [in] BSTR Workspace,
    [out, retval] BSTR *mlString)
```

Microsoft Visual Basic Client

```
GetCharArray(varname As String, workspace As String) As String
```

MATLAB Client

```
str = GetCharArray(h,'varname','workspace')
```

Description

`str = GetCharArray(h,'varname','workspace')` gets the character array stored in `varname` from the specified *`workspace`* of the server attached to handle `h` and returns it in `str`. The values for *`workspace`* are `base` or `global`.

Examples

This example uses a Visual Basic .NET client.

1 Create the Visual Basic application. Use the `MsgBox` command to control flow between MATLAB and the application.

```
Dim Matlab As Object
Dim S As String
Matlab = CreateObject("matlab.application")
```

```
MsgBox("In MATLAB, type" & vbCrLf _
    & "str='new string';")
```

2 Open the MATLAB window, then type:

```
str='new string';
```

3 Click **Ok**.

```
Try
    S = Matlab.GetCharArray("str", "base")
    MsgBox("str = " & S)
Catch ex As Exception
    MsgBox("You did not set 'str' in MATLAB")
End Try
```

The Visual Basic `MsgBox` displays what you typed in MATLAB.

See Also

`PutCharArray` | `GetWorkspaceData` | `GetVariable`

Introduced before R2006a

getenv

Environment variable

Syntax

```
getenv 'name'
N = getenv('name')
```

Description

`getenv 'name'` searches the underlying operating system environment list for a string of the form `name=value`, where `name` is the input string. If found, MATLAB returns the string `value`. If the specified name cannot be found, an empty matrix is returned.

On UNIX platforms, the shell you use to start MATLAB determines the operating system environment. For example, starting MATLAB on a Mac platform from the `Applications` folder creates a different shell environment from launching MATLAB from Terminal.

`N = getenv('name')` returns `value` to the variable `N`.

Examples

```
os = getenv('OS')
```

```
os =
Windows_NT
```

See Also

`setenv` | `computer` | `pwd` | `ver` | `path`

Introduced before R2006a

getfield

Field of structure array

Syntax

```
value = getfield(struct, 'field')
value = getfield(struct,{sIndx1,...,sIndxM},'field',
{fIndx1,...,fIndxN})
```

Description

`value = getfield(struct, 'field')`, where `struct` is a 1-by-1 structure, returns the contents of the specified field, equivalent to `value = struct.field`. Pass field references as strings.

`value = getfield(struct,{sIndx1,...,sIndxM},'field', {fIndx1,...,fIndxN})` returns the contents of the specified field, equivalent to `value = struct(sIndx1,...,sIndxM).field(fIndx1,...,fIndxN)`. The `getfield` function supports multiple sets of `field` and `fIndx` inputs, and all `Indx` inputs are optional. If structure `struct` or any of the fields is a nonscalar structure, and you do not specify an `Indx`, the `getfield` function returns the values associated with the first index. If you specify a single colon operator for an `fIndx` input, enclose it in single quotation marks: `':'`.

Examples

The `what` function returns a structure array that describes the MATLAB files in the current folder. Find the files with the `.m` extension:

```
files = getfield(what, 'm');
```

To perform the same task by indexing requires that you create a temporary variable:

```
templist = what;
files = templist.m;
```

Find values within a structure that contains nested fields:

```
level = 5;
semester = 'Fall';
subject = 'Math';
student = 'John_Doe';
fieldnames = {semester subject student};

% Add data to a structure named grades.
grades(level).(semester).(subject).(student)(10,21:30) = ...
             [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];

% Retrieve the data added.
getfield(grades, {level}, fieldnames{:}, {10,21:30})
```

Using the structure defined in the previous example, find all values in the tenth row of the specified field:

```
getfield(grades, {level}, fieldnames{:}, {10,':'})
```

More About

Tips

- For most cases, retrieve data from a structure array by indexing rather than using the `getfield` function. For more information, see “Access Data in a Structure Array” and “Generate Field Names from Variables”.
- Call `getfield` to simplify references to structure arrays with nested fields, or to avoid creating unnecessary temporary variables, as shown in the Examples section.

- “Generate Field Names from Variables”
- “Access Data in a Structure Array”

See Also

`setfield` | `fieldnames` | `isfield` | `orderfields` | `rmfield`

Introduced before R2006a

VideoReader.getFileFormats

Class: VideoReader

File formats that `VideoReader` supports

Syntax

```
formats = VideoReader.getFileFormats()
```

Description

`formats = VideoReader.getFileFormats()` returns an array of `audiovideo.FileFormatInfo` objects that indicate which formats `VideoReader` can read on the current system. Each object has the following properties: `Extension`, `Description`, `ContainsVideo`, and `ContainsAudio`.

Tips

- On Windows and UNIX systems, the list of file formats does not always contain all the formats that `VideoReader` can read on your system. `getFileFormats` returns a platform-dependent, static list of formats that `VideoReader` can read on most systems.
- On all systems, `VideoReader` cannot always read a particular video file even if `getFileFormats` lists its format. For more information, see Supported Video File Formats.

Output Arguments

`formats`

Array of `audiovideo.FileFormatInfo` objects, which have the following properties:

`Extension`	File extension.

`Description`	Text description of the file format.
`ContainsVideo`	Whether `VideoReader` can read video from this format.
`ContainsAudio`	Whether `VideoReader` can read audio from this format.

To convert this array to a filter list for dialog boxes generated with `uigetfile`, use the `getFilterSpec` method with the following syntax:

```
filterSpec = getFilterSpec(formats)
```

The filter list includes `'All Video Files'` in the first row of the cell array, and `'All Files (*.*)'` in the last row.

Examples

View the list of file formats that `VideoReader` supports on your system:

```
VideoReader.getFileFormats()
```

On a Windows system, this list appears as follows:

```
Video File Formats:
    .asf - ASF File
    .asx - ASX File
    .avi - AVI File
    .mj2 - Motion JPEG2000
    .mpg - MPEG-1
    .wmv - Windows Media Video
```

Create a dialog box to select a video file:

```
% Get the supported file formats.
formats = VideoReader.getFileFormats();

% Convert to a filter list.
filterSpec = getFilterSpec(formats);

% Create the dialog box.
[filename, pathname] = uigetfile(filterSpec);
```

Check whether `VideoReader` can read an `.avi` file on the current system:

```
fmtList = VideoReader.getFileFormats();

if any(ismember({fmtList.Extension},'avi'))
     disp('VideoReader can read AVI files on this system.');
else
     disp('VideoReader cannot read AVI files on this system.');
end
```

See Also

VideoReader

getframe

Capture axes or figure as movie frame

Syntax

```
F = getframe
F = getframe(ax)
F = getframe(fig)
F = getframe( ___ ,rect)
```

Description

`F = getframe` captures the current axes as it appears on the screen as a movie frame. `F` is a structure containing the image data. `getframe` captures the axes at the same size that it appears on the screen. It does not capture tick labels or other content outside the axes outline.

`F = getframe(ax)` captures the axes identified by `ax` instead of the current axes.

`F = getframe(fig)` captures the figure identified by `fig`. Specify a figure if you want to capture the entire interior of the figure window, including the axes title, labels, and tick marks. The captured movie frame does not include the figure menu and tool bars.

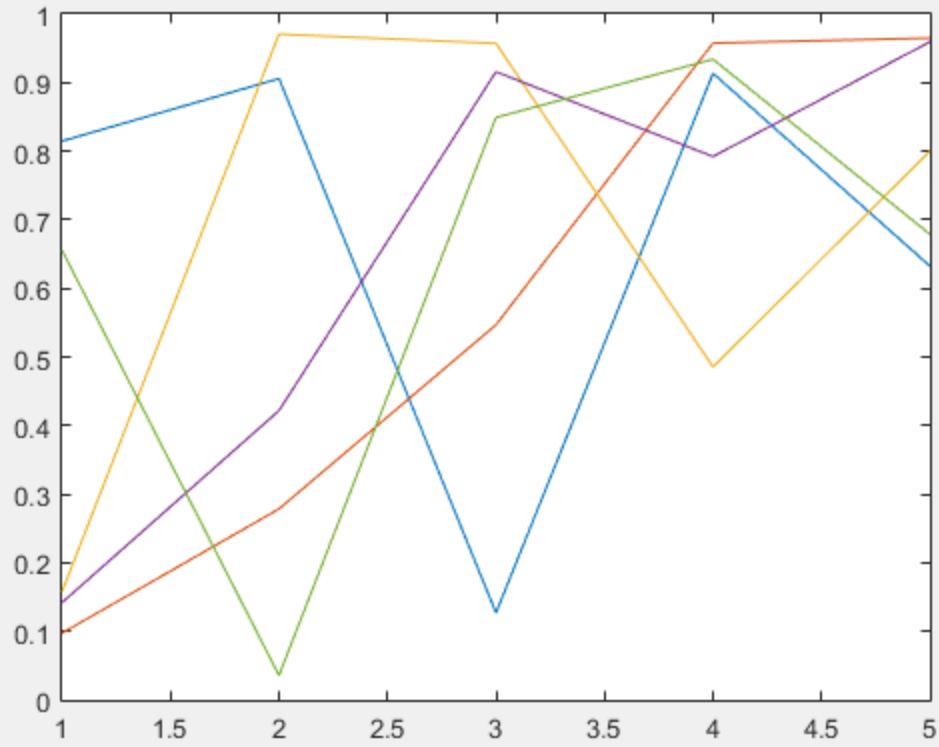
`F = getframe( ___ ,rect)` captures the area within the rectangle defined by `rect`. Specify `rect` as a four-element vector of the form `[left bottom width height]`. Use this option with either the `ax` or `fig` input arguments in the previous syntaxes.

Examples

Capture Contents of Current Axes

Create a plot of random data. Capture the axes and return the image data. `getframe` captures the interior of the axes and the axes outline. It does not capture content that extends beyond the axes outline.

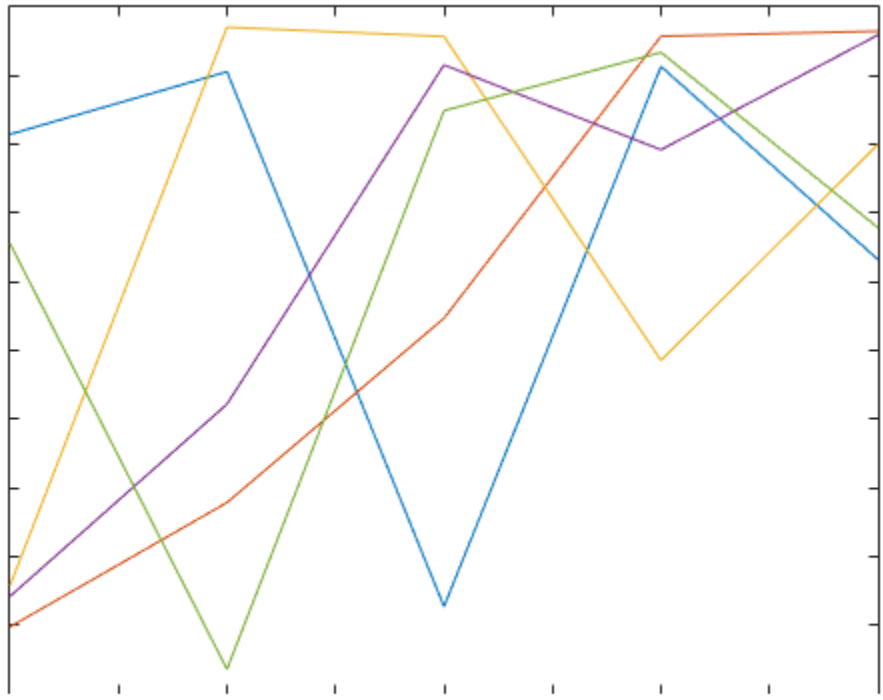
```
plot(rand(5))
F = getframe;
```



`F` is a structure with the field `cdata` that contains the captured image data.

Use `imshow` to display the captured image data.

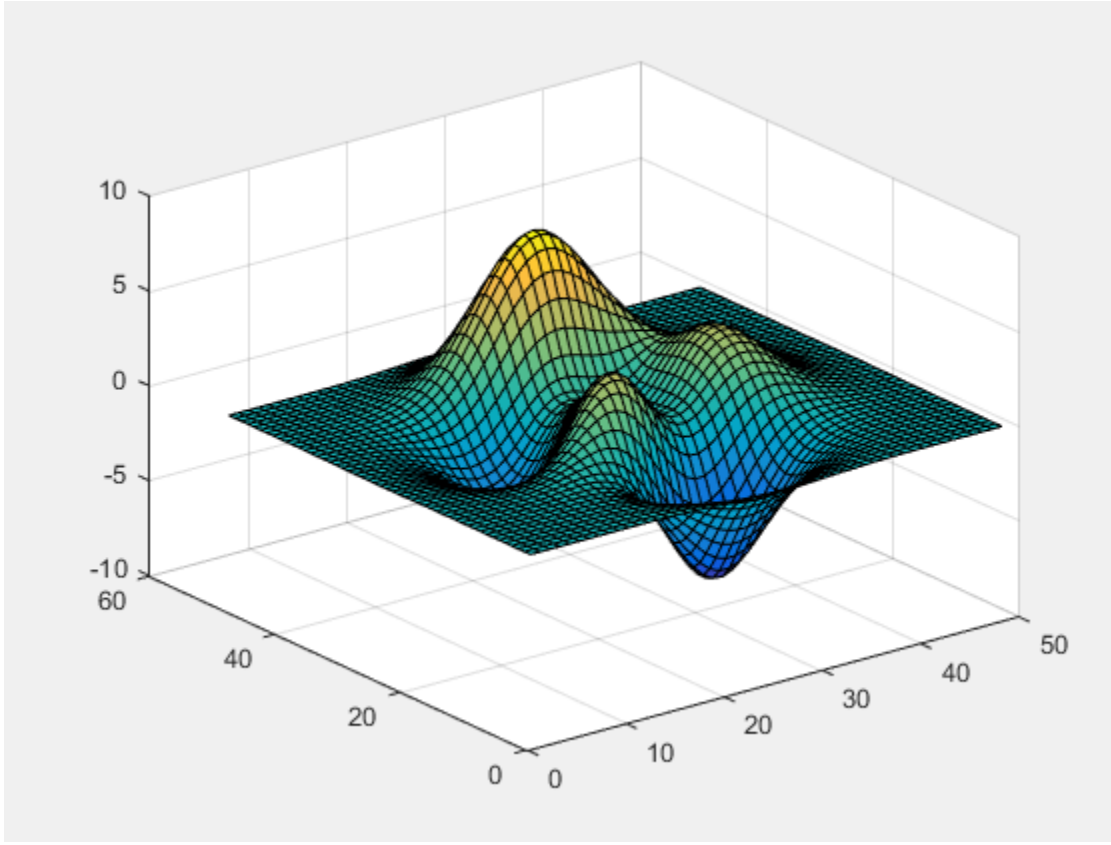
```
figure
imshow(F.cdata)
```



Capture Contents of Figure

Create a surface plot. Capture the interior of the figure window, excluding the menu and tool bars.

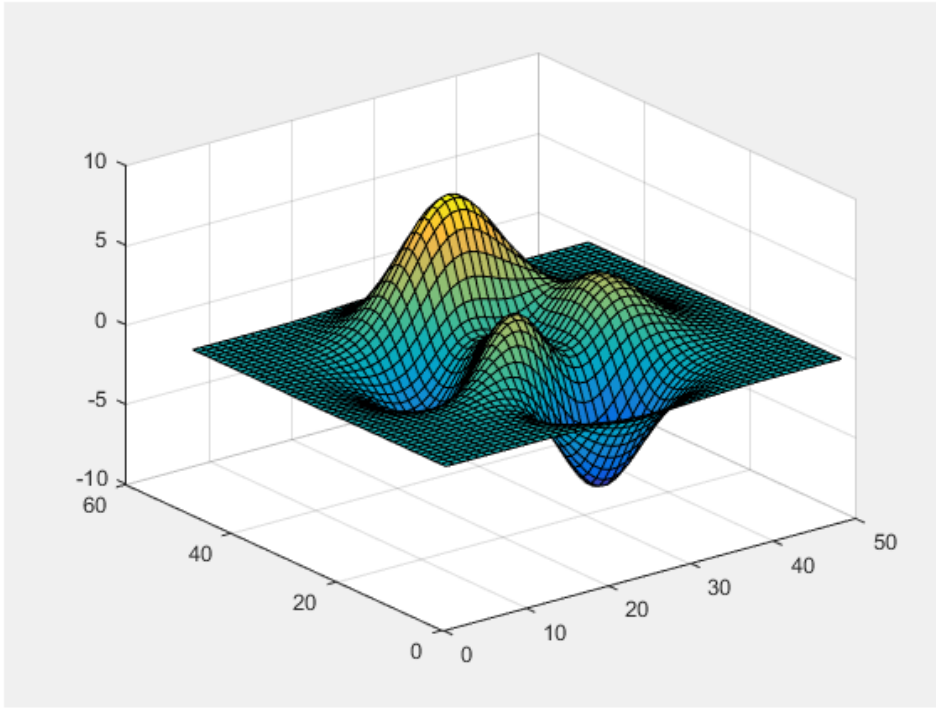
```
surf(peaks)
F = getframe(gcf);
```



`F` is a structure with the field `cdata` that contains the captured image data.

Use `imshow` to display the captured image data.

```
figure
imshow(F.cdata)
```

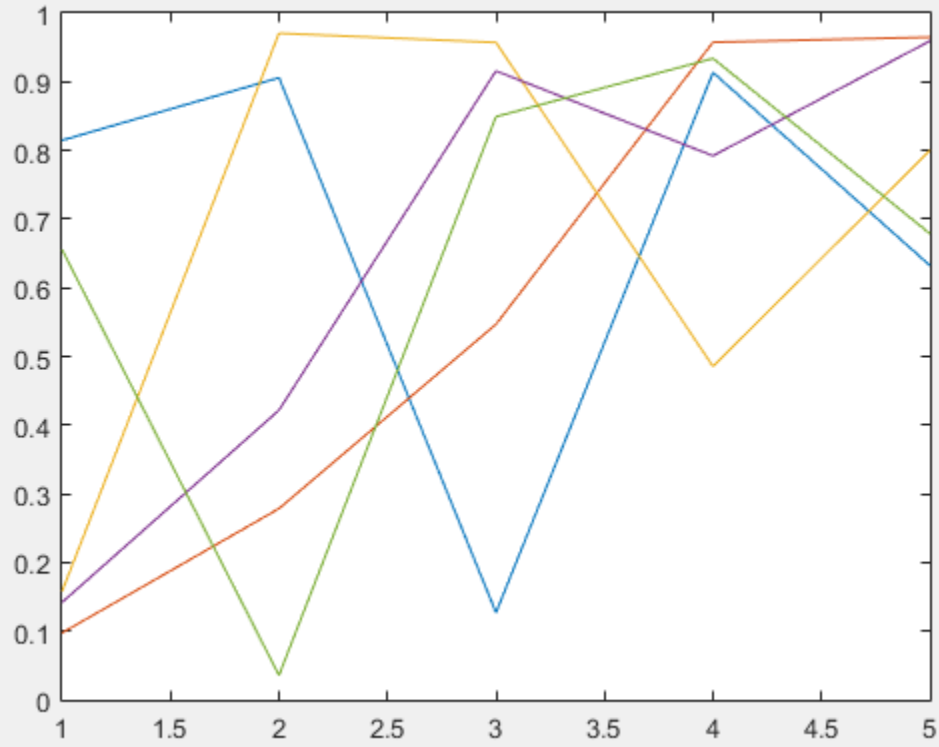


Capture Axes Plus 30 Pixel Border

Capture the interior of an axes plus a margin of 30 pixels in each direction. The added margin is necessary to include the tick labels in the capture frame. Depending on the size of the tick labels, the margin might need to be adjusted.

Create a plot of random data.

```
plot(rand(5))
```



Change the axes units to pixels and return the current axes position. The third and fourth elements of the position vector specify the axes width and height in pixels.

```
drawnow
ax = gca;
ax.Units = 'pixels';
pos = ax.Position
```

```
pos =

   73.8000   47.2000  434.0000  342.3000
```

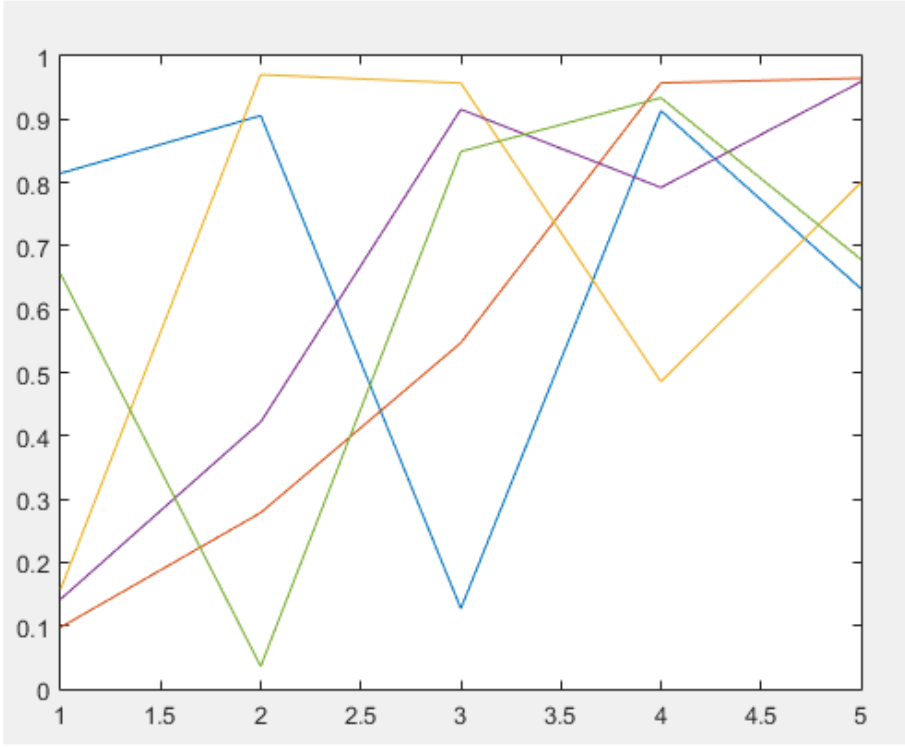
Create a four-element vector, `rect`, that defines a rectangular area covering the axes plus the desired margin. The first two elements of `rect` specify the lower left corner of

the rectangle relative to the lower left corner of the axes. The last two elements of `rect` specify the width and height of the rectangle. Reset the axes units to the default value of `'normalized'`.

```
marg = 30;
rect = [-marg, -marg, pos(3)+2*marg, pos(4)+2*marg];
F = getframe(gca,rect);
ax.Units = 'normalized';
```

Use `imshow` to display the captured image data.

```
figure
imshow(F.cdata)
```

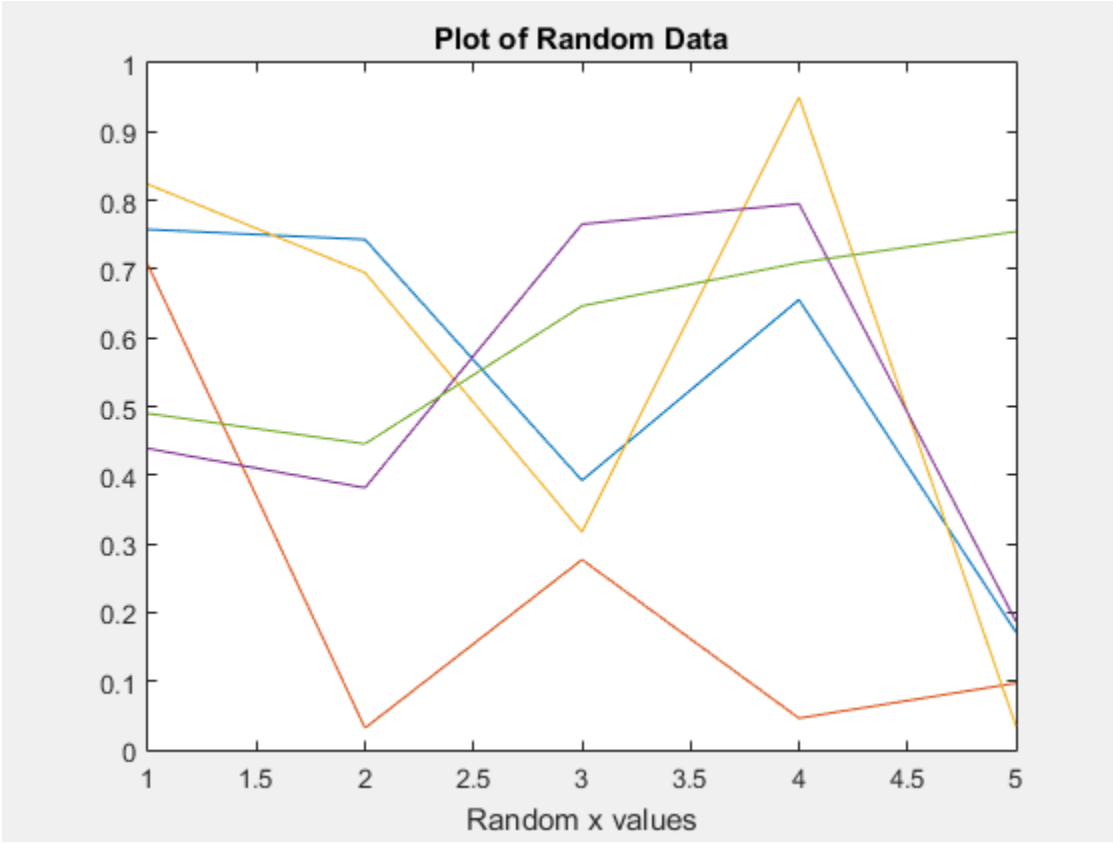


Automatically Calculate Margin to Include Title and Labels

Automatically calculate a margin around the axes so that the captured image data includes the title, axis labels, and tick labels.

Create a plot with a title and an *x*-axis label.

```
plot(rand(5))
xlabel('Random x values')
title('Plot of Random Data')
```



Change the axes units to pixels and store the `Position` and `TightInset` property values for the axes. The `TighInset` property is a four-element vector of the form `[left bottom right top]`. The values are the margins used around the axes for the tick values and text labels.

```
drawnow
ax = gca;
ax.Units = 'pixels';
pos = ax.Position;
ti = ax.TightInset;
```

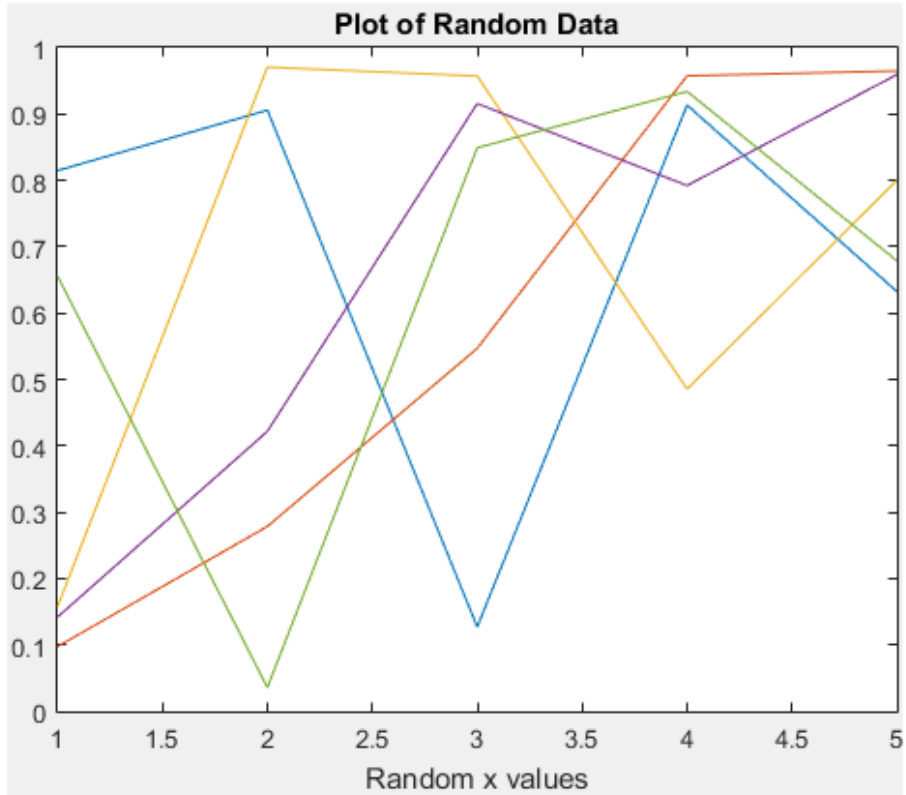
Create a four-element vector, `rect`, that defines a rectangular area covering the axes plus the automatically calculated margin. The first two elements of `rect` specify the

lower left corner of the rectangle relative to the lower left corner of the axes. The last two elements of `rect` specify the width and height of the rectangle.

```
rect = [-ti(1), -ti(2), pos(3)+ti(1)+ti(3), pos(4)+ti(2)+ti(4)];
F = getframe(ax,rect);
```

Use `imshow` to display the captured image data.

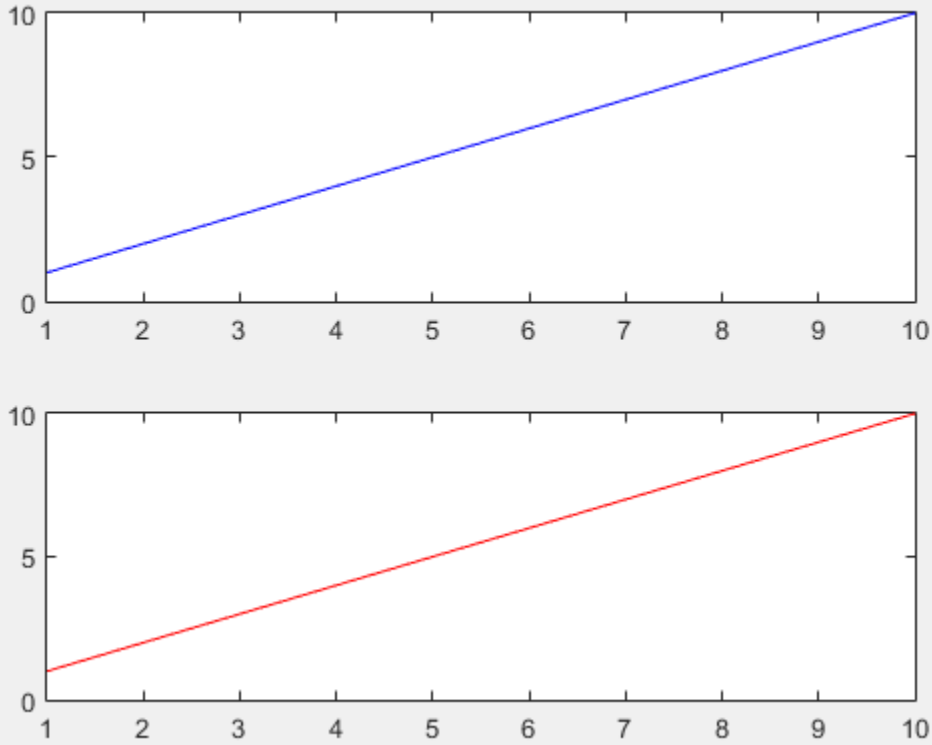
```
figure
imshow(F.cdata)
```



Capture Specific Subplot Axes

Create a figure with two subplots. In the upper subplot, plot a blue line. In the lower subplot, plot a red line.

```
ax1 = subplot(2,1,1);
plot(1:10,'b')
ax2 = subplot(2,1,2);
plot(1:10,'r')
```

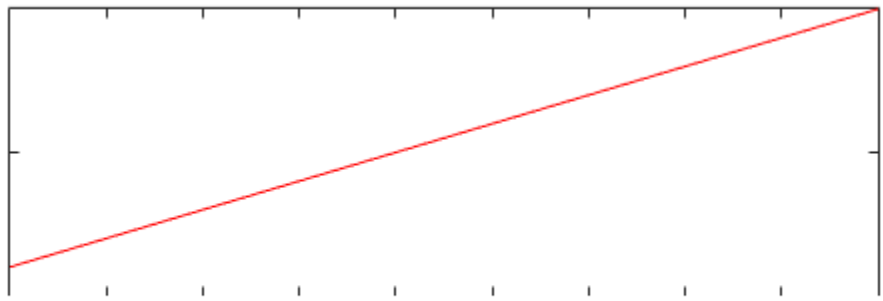


Capture the contents of the lower subplot. `getframe` captures the interior and border of the subplot. It does not capture tick values or labels that extend beyond the outline of the subplot.

```
F = getframe(ax2);
```

Use `imshow` to display the capture image data.

```
figure
imshow(F.cdata)
```



Record Frames and Play Movie

Use the `getframe` function in a loop to record frames of the peaks function vibrating. Preallocate an array to store the movie frames.

```
Z = peaks;
surf(Z)
axis tight manual
ax = gca;
ax.NextPlot = 'replaceChildren';

loops = 40;
F(loops) = struct('cdata',[],'colormap',[]);
for j = 1:loops
    X = sin(j*pi/10)*Z;
    surf(X,Z)
    drawnow
    F(j) = getframe(gcf);
end
```

Play back the movie two times.

```
fig = figure;
```

```
movie(fig,F,2)
```

Input Arguments

`ax` — Axes to capture

axes object

Axes to capture, specified as an axes object. Use this option if you want to capture an axes that is not the current axes.

`getframe` captures the content within the smallest rectangle that encloses the axes outline. If you want to capture all the tick values and labels, then use the `fig` input argument instead.

Example: `F = getframe(ax);`

`fig` — Figure to capture

figure object

Figure to capture, specified as a figure object. Use `gcf` to capture the current figure.

Example: `F = getframe(gcf);`

`rect` — Rectangular area to capture

four-element vector of the form `[left bottom width height]`

Rectangular area to capture, specified as a four-element vector of the form `[left bottom width height]`. The `left` and `bottom` elements define the position of the lower left corner of the rectangle. The position is relative to the figure or axes that is specified as the first input argument to `getframe`. The `width` and `height` elements define the dimensions of the rectangle. The values are in pixel units.

Specify a rectangle that is fully contained within the figure window.

Example: `F = getframe(gcf,[0 0 560 420]);`

Output Arguments

`F` — Movie frame

structure

Movie frame, returned as a structure with two fields:

- `cdata` — The image data stored as an array of `uint8` values.
- `colormap` — The colormap. On true color systems, this field is empty.

More About

Tips

- For the fastest performance when using `getframe`, make sure that the figure is visible on the screen. If the figure is not visible, `getframe` can still capture the figure, but performance can be slower.
- For more control over the resolution of the image data, use the `print` function instead. The `cdata` output argument with print returns the image data. The `resolution` input argument controls the resolution of the image.

See Also

`frame2im` | `im2frame` | `image` | `imshow` | `movie` | `print`

Introduced before R2006a

GetFullMatrix

Matrix from Automation server workspace

Syntax

IDL Method Signature

```
GetFullMatrix([in] BSTR varname, [in] BSTR workspace,
    [in, out] SAFEARRAY(double) *pr, [in, out] SAFEARRAY(double) *pi)
```

Microsoft Visual Basic Client

```
GetFullMatrix(varname As String, workspace As String,
    [out] XReal As Double, [out] XImag As Double
```

MATLAB Client

```
[xreal ximag] = GetFullMatrix(h,'varname','workspace',zreal,zimag)
```

Description

`[xreal ximag] = GetFullMatrix(h,'varname','workspace',zreal,zimag)` gets matrix stored in variable `varname` from the specified *`workspace`* of the server attached to handle `h`. The function returns the real part in `xreal` and the imaginary part in `ximag`. The values for *`workspace`* are `base` or `global`.

The `zreal` and `zimag` arguments are matrices of the same size as the real and imaginary matrices (`xreal` and `ximag`) returned from the server. The `zreal` and `zimag` matrices are commonly set to zero.

Use `GetFullMatrix` for values of type `double` only. Use `GetVariable` or `GetWorkspaceData` for other types.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the

`variant` data type instead of the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

This example uses a Visual Basic .NET client to read data from a MATLAB Automation server:

1. Create the Visual Basic application. To control flow between MATLAB and the application, use the `MsgBox` command.

```
Dim MatLab As Object
Dim Result As String
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim i, j As Integer

MatLab = CreateObject("matlab.application")
Result = MatLab.Execute("M = rand(5);")
MsgBox("In MATLAB, type" & vbCrLf _
    & "M(3,4)")
```

2. Open the MATLAB window and type:

```
M(3,4)
```

3. Click **Ok**.

4. In the Visual Basic application:

```
MatLab.GetFullMatrix("M", "base", XReal, XImag)
i = 2   %0-based array
j = 3

MsgBox("XReal(" & i + 1 & "," & j + 1 & ")" & _
    " = " & XReal(i, j))
```

5. To close and terminate MATLAB, click **Ok**.

See Also

`PutFullMatrix` | `GetVariable` | `GetWorkspaceData` | `Execute`

Introduced before R2006a

getnext

Get next value from ValueIterator

Syntax

```
X = getnext(ValIter)
```

Description

`X = getnext(ValIter)` returns the next available value in `ValIter`. Use the `hasnext` function to confirm availability of values in `ValIter` before calling `getnext`.

Examples

Get Values from ValueIterator in Reduce Function

Use the `hasnext` and `getnext` functions in a `while` loop within the reduce function to iteratively get values from the `ValueIterator` object. For example,

```
function MeanDistReduceFun(sumLenKey, sumLenIter, outKVStore)
    sumLen = [0 0];
    while hasnext(sumLenIter)
        sumLen = sumLen + getnext(sumLenIter);
    end
    add(outKVStore, 'Mean', sumLen(1)/sumLen(2));
end
```

Always call `hasnext` before `getnext` to confirm availability of a value. `mapreduce` returns an error if you call `getnext` with no remaining values in the `ValueIterator` object.

Input Arguments

`ValIter` — Intermediate value iterator

`ValueIterator` object

Intermediate value iterator, specified as a `ValueIterator` object. The `mapreduce` function automatically creates this object during execution. The second input to the reduce function specifies the variable name for the `ValueIterator` object, which is the variable name to use with the `hasnext` and `getnext` functions.

For more information, see Using ValueIterator Objects.

More About

- Using ValueIterator Objects
- "Build Effective Algorithms with MapReduce"

See Also

`hasnext` | `mapreduce`

Introduced in R2014b

getpixelposition

Get component position in pixels

Syntax

```
position = getpixelposition(handle)
position = getpixelposition(handle,recursive)
```

Description

`position = getpixelposition(handle)` gets the position, in pixel units, of the component with handle `handle`. MATLAB returns the `position` as a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

`position = getpixelposition(handle,recursive)` gets the position as above. If `recursive` is true, the returned position is relative to the parent figure of `handle`.

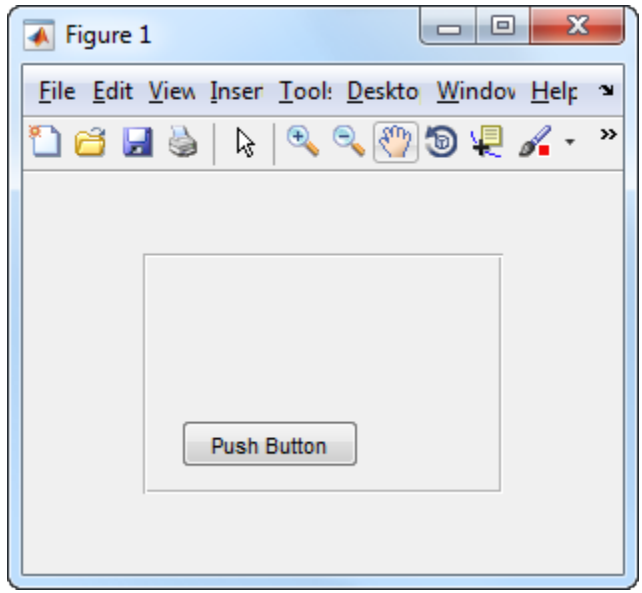
Use the `getpixelposition` function only to obtain coordinates for children of figures and container components (uipanels, or uibuttongroups). Results are not reliable for children of axes or other Handle Graphics objects.

Examples

This example creates a push button within a panel, and then retrieves its position, in pixels, relative to the *panel*.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton',...
     'Units','Normalized',...
     'String','Push Button',...
     'Position',[.1 .1 .5 .2]);
drawnow;
pos1 = getpixelposition(h1)
```

```
pos1 =

   18.6000   12.6000   88.0000   23.2000
```



The following statement retrieves the position of the push button, in pixels, relative to the *figure*.

```
pos1 = getpixelposition(h1,true)
```

```
pos1 =
      78.6000   52.6000   88.0000   23.2000
```

See Also

`setpixelposition` | `uicontrol` | `uipanel`

getpoints

Return points that define animated line

To use this function, you must first create an animated line with the `animatedline` function. For more information on line animations, see Using Animated Line Objects.

Syntax

```
[x,y] = getpoints(h)
[x,y,z] = getpoints(h)
```

Description

`[x,y] = getpoints(h)` returns the *x* and *y* coordinates for the points that define the animated line specified by `h`.

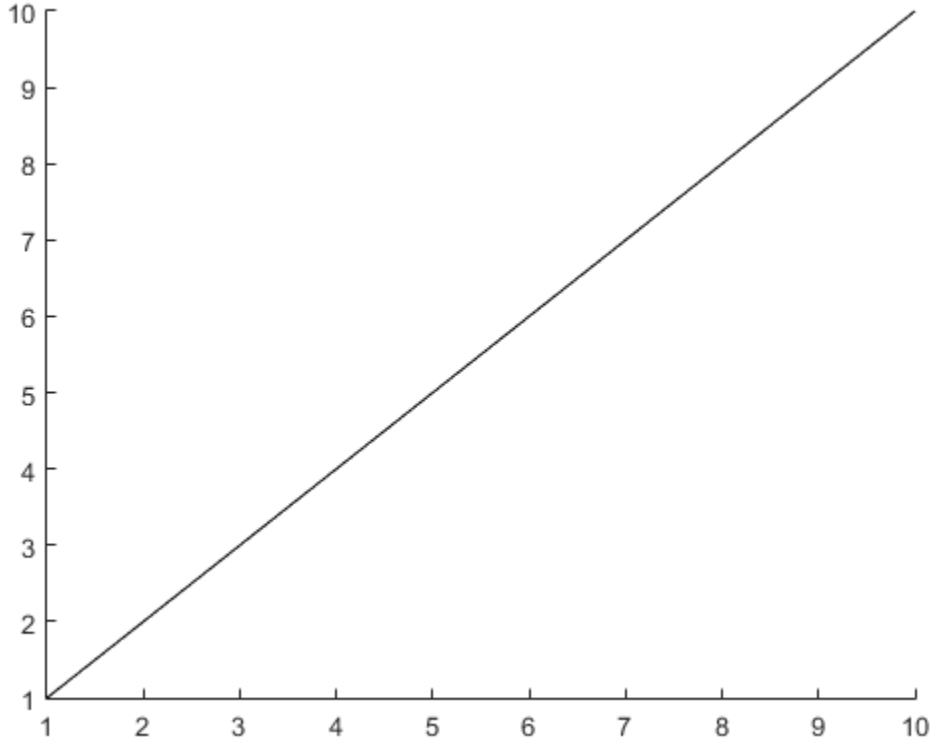
`[x,y,z] = getpoints(h)` returns the coordinates for the 3-D animated line specified by `h`. If the line does not have any *z* values, then `getpoints` returns `z` as a vector of zeros the same length as `x` and `y`.

Examples

Return Points from Animated Line

Create an animated line with 10 points. Then, return the points stored in the animated line.

```
h = animatedline(1:10,1:10);
```




```
[x,y] = getpoints(h)

x =

     1     2     3     4     5     6     7     8     9    10

y =

     1     2     3     4     5     6     7     8     9    10
```

Input Arguments

h — Animated line object

animated line object

Animated line object. Create an animated line object using the `animatedline` function.

Output Arguments

x — x values

vector

x values that define the animated line, returned as vector.

y — y values

vector

y values that define the animated line, returned as vector.

z — z values

vector

z values that define the 3-D animated line, returned as vector.

See Also

Functions

`addpoints` | `animatedline` | `clearpoints`

Using Objects

Using Animated Line Objects

getpref

Preference

Syntax

```
getpref('group','pref')
getpref('group','pref',default)
getpref('group',{'pref1','pref2',...'prefn'})
getpref('group',{'pref1',...'prefn'},{default1,...defaultn})
getpref('group')
getpref
```

Description

`getpref('group','pref')` returns the value for the preference specified by `group` and `pref`. It is an error to get a preference that does not exist.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g. `'ApplicationOnePrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`getpref('group','pref',default)` returns the current value if the preference specified by `group` and `pref` exists. Otherwise creates the preference with the specified default value and returns that value.

`getpref('group',{'pref1','pref2',...'prefn'})` returns a cell array containing the values for the preferences specified by `group` and the cell array of preference names. The return value is the same size as the input cell array. It is an error if any of the preferences do not exist.

`getpref('group',{'pref1',...'prefn'},{default1,...defaultn})` returns a cell array with the current values of the preferences specified by `group` and the cell array of preference names. Any preference that does not exist is created with the specified default value and returned.

`getpref('group')` returns the names and values of all preferences in the `group` as a structure.

`getpref` returns all groups and preferences as a structure.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

Get the Value of an Existing Preference

```
addpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')

ans =
     1.0
```

Create a Preference Using Specified Default Value

```
setpref('mytoolbox','version','1.0')
rmpref('mytoolbox','version')
getpref('mytoolbox','version','1.0');
getpref('mytoolbox','version')


ans =

1.0
```

The first call to `getpref` adds the `'version'` preference and sets its value to the default value, `1.0`. The second call to `getpref` verifies that the preference exists, and that its value is `1.0`.

See Also

`addpref` | `ispref` | `rmpref` | `setpref` | `uigetpref` | `uisetpref`

Introduced before R2006a

VideoWriter.getProfiles

Class: VideoWriter

List profiles and file formats supported by `VideoWriter`

Syntax

```
profiles = VideoWriter.getProfiles()
```

Description

`profiles = VideoWriter.getProfiles()` returns an array of `audiovideo.writer.ProfileInfo` objects that indicate the types of files `VideoWriter` can create.

Output Arguments

`profiles`

An array of `audiovideo.writer.ProfileInfo` objects, which have the following read-only properties:

`Name`	String profile name, such as `'Uncompressed AVI'`.
`Description`	String description of the profile.
`FileExtensions`	Cell array of strings containing file extensions supported by the file format.
`ColorChannels`	Number of color channels in each output video frame.
`CompressionRatio`	Number greater than 1 that specifies the target ratio between the number of bytes in the input image and the number of bytes in the compressed image. Only applies to objects associated with Motion JPEG 2000 files. Default: `10`.
`FrameRate`	Rate of playback for the video in frames per second. Default: `30`.

`LosslessCompression`	Boolean value (logical `true` or `false`) that specifies whether to use reversible mode, so that the decompressed data is identical to the input data. When `true`, `VideoWriter` ignores values for `CompressionRatio`. Only applies to objects associated with Motion JPEG 2000 files.
`MJ2BitDepth`	Number of least significant bits in the input image data, from 1 to 16. Only applies to objects associated with Motion JPEG 2000 files.
`Quality`	Number from 0 through 100. Higher values correspond to higher quality video and larger files. Only applies to objects associated with the `MPEG-4` or `Motion JPEG AVI` profile. Default: `75`.
`VideoBitsPerPixel`	Number of bits per pixel in each output video frame.
`VideoCompressionMethod`	String indicating the type of video compression, such as `'None'` or `'Motion JPEG'`.
`VideoFormat`	String indicating the MATLAB representation of the video format, such as `'RGB24'`.

Examples

Profile Information

View the list of available profiles and specific information about the `'Uncompressed AVI'` profile.

```
profiles = VideoWriter.getProfiles()

uncompAVI = find(ismember({profiles.Name},'Uncompressed AVI'));
profiles(uncompAVI)
profiles(uncompAVI).FileExtensions
```

See Also

`VideoWriter`

getsampleusingtime (tscollection)

Extract data samples into new `tscollection` object

Syntax

```
tsc2 = getsampleusingtime(tsc1,Time)
tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)
```

Description

`tsc2 = getsampleusingtime(tsc1,Time)` returns a new `tscollection` tsc2 with a single sample corresponding to Time in tsc1.

`tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)` returns a new `tscollection` tsc2 with samples between the times StartTime and EndTime in tsc1.

More About

Tips

When the time vector in ts1 is numeric, `StartTime` and `EndTime` must also be numeric. When the times in ts1 are date strings and the `StartTime` and `EndTime` values are numeric, then the `StartTime` and `EndTime` values are treated as `datenum` values.

See Also

`tscollection`

Introduced before R2006a

getTag

Class: Tiff

Value of specified tag

Syntax

```
tagValue = getTag(tagId)
```

Description

`tagValue = getTag(tagId)` retrieves the value of the TIFF tag specified by `tagId`. You can specify `tagId` as a character string (`'ImageWidth'`) or using the numeric tag identifier defined by the TIFF specification (256). To see a list of all the tags with their numeric identifiers, view the value of the Tiff object `TagID` property. Use the `TagID` property to specify the value of a tag. For example, `Tiff.TagID.ImageWidth` is equivalent to the tag's numeric identifier.

Examples

Get Value of Tag

Open a `Tiff` object and get the value of a tag. Specify the tag by name.

```
t = Tiff('example.tif','r');
tagval = t.getTag('ImageWidth')
```

```
tagval =

   600
```

Alternatively, specify the tag by numeric identifier.

```
tagval1 = t.getTag(256)
```

```
tagval1 =
```

```
   600
```

Another way to specify the numeric identifier is to use the `TagID` property.

```
tagval2 = t.getTag(Tiff.TagID.ImageWidth)
```

```
tagval2 =

   600
```

Close the `Tiff` object.

```
t.close();
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFGetField` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTiff - TIFF Library and Utilities.

See Also

`Tiff.setTag`

Tiff.getTagNames

Class: Tiff

List of recognized TIFF tags

Syntax

```
tagNames = Tiff.getTagNames()
```

Description

`tagNames = Tiff.getTagNames()` returns a cell array of TIFF tags recognized by the `Tiff` object.

Examples

Retrieve a list of TIFF tags recognized by the `Tiff` object.

```
Tiff.getTagNames

ans =

    'SubFileType'
    'ImageWidth'
    'ImageLength'
    'BitsPerSample'
    'Compression'
    'Photometric'
    'Thresholding'
    'FillOrder'
    'DocumentName'
    'ImageDescription'
     .
     .
     .
```

See Also

`Tiff.getTag`

gettimeseriesnames

Cell array of names of `timeseries` objects in `tscollection` object

Syntax

```
names = gettimeseriesnames(tsc)
```

Description

`names = gettimeseriesnames(tsc)` returns `names` of `timeseries` objects in a `tscollection` object `tsc`. `names` is a cell array of strings.

Examples

1. Create `timeseries` objects `a` and `b`.

   ```
   a = timeseries(rand(1000,1),'name','position');
   b = timeseries(rand(1000,1),'name','response');
   ```

2. Create a `tscollection` object that includes these two time series.

   ```
   tsc = tscollection({a,b});
   ```

3. Get the names of the `timeseries` objects in `tsc`.

   ```
   names = gettimeseriesnames(tsc)

   names =

       'position'    'response'
   ```

See Also

`timeseries` | `tscollection`

Introduced before R2006a

gettsafteratevent

New `timeseries` object with samples occurring at or after event

Syntax

```
ts1 = gettsafteratevent(ts,event)
ts1 = gettsafteratevent(ts,event,n)
```

Description

`ts1 = gettsafteratevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring at and after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of the time series ts that matches the event name specifies the time.

`ts1 = gettsafteratevent(ts,event,n)` returns a new `timeseries` object ts1 with samples at and after an event in ts, where n is the number of the event occurrence with a matching event name.

More About

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `gettsbetweenevents` | `tsdata.event` | `timeseries`

Introduced before R2006a

gettsafterevent

New `timeseries` object with samples occurring after event

Syntax

```
ts1 = gettsafterevent(ts,event)
ts1 = ttsafterevent(ts,event,n)
```

Description

`ts1 = gettsafterevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = ttsafterevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring after an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

More About

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterateevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries` | `tsdata.event`

Introduced before R2006a

gettsatevent

New `timeseries` object with samples occurring at event

Syntax

```
ts1 = gettsatevent(ts,event)
ts1 = gettsatevent(ts,event,n)
```

Description

`ts1 = gettsatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsatevent(ts,event,n)` returns a new time series `ts1` with samples occurring at an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

More About

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in the `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsafteratevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries`

Introduced before R2006a

gettsbeforeatevent

New `timeseries` object with samples occurring before or at event

Syntax

```
ts1 = gettsbeforeatevent(ts,event)
ts1 = gettsbeforeatevent(ts,event,n)
```

Description

`ts1 = gettsbeforeatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeatevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

More About

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `gettsbetweenevents` | `tsdata.event`

Introduced before R2006a

gettsbeforeevent

New `timeseries` object with samples occurring before event

Syntax

```
ts1 = gettsbeforeevent(ts,event)
ts1 = gettsbeforeevent(ts,event,n)
```

Description

`ts1 = gettsbeforeevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where n is the number of the event occurrence with a matching event name.

More About

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeatevent` | `gettsbetweenevents` | `tsdata.event`

Introduced before R2006a

gettsbetweenevents

New `timeseries` object with samples occurring between events

Syntax

```
ts1 = gettsbetweenevents(ts,event1,event2)
ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)
```

Description

`ts1 = gettsbetweenevents(ts,event1,event2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `event1` and `event2` can be either a `tsdata.event` object or a string. When `event1` and `event2` are `tsdata.event` objects, the time defined by the events is used. When `event1` and `event2` are strings, the first `tsdata.event` object in the `Events` property of `ts` that matches the event names specifies the time.

`ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `n1` and `n2` are the `n`th occurrences of the events with matching event names.

More About

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `tsdata.event`

Introduced before R2006a

GetVariable

Data from variable in Automation server workspace

Syntax

IDL Method Signature

```
HRESULT GetVariable([in] BSTR varname, [in] BSTR workspace,
    [out, retval] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetVariable(varname As String, workspace As String) As Object
```

MATLAB Client

```
D = GetVariable(h,'varname','workspace')
```

Description

`D = GetVariable(h,'varname','workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`.

Do *not* use `GetVariable` on sparse arrays, structures, or function handles.

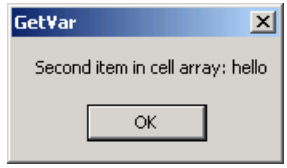
If your scripting language requires a result be returned explicitly, use the `GetVariable` function in place of `GetWorkspaceData`, `GetFullMatrix`, or `GetCharArray`.

Examples

This example uses a Visual Basic .NET client to read data from a MATLAB Automation server:

```
Dim Matlab As Object
```

```
Dim Result As String
Dim C2 As Variant
Matlab = CreateObject("matlab.application")
Result = Matlab.Execute("C1 = {25.72, 'hello', rand(4)};")
C2 = Matlab.GetVariable("C1", "base")
MsgBox("Second item in cell array: " & C2(0, 1))
```



See Also

GetWorkspaceData | GetFullMatrix | GetCharArray | Execute

Introduced before R2006a

Tiff.getVersion

Class: Tiff

LibTIFF library version

Syntax

```
versionString = Tiff.getVersion()
```

Description

`versionString = Tiff.getVersion()` returns the version number and other information about the LibTIFF library.

Examples

Display version of LibTIFF library:

```
Tiff.getVersion

ans =

LIBTIFF, Version 3.9.5
Copyright (c) 1988-1996 Sam Leffler
Copyright (c) 1991-1996 Silicon Graphics, Inc.
```

References

This method corresponds to the `TIFFGetVersion` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

GetWorkspaceData

Data from Automation server workspace

Syntax

IDL Method Signature

```
HRESULT GetWorkspaceData([in] BSTR varname, [in] BSTR workspace,
    [out] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetWorkspaceData(varname As String, workspace As String) As Object
```

MATLAB Client

```
D = GetWorkspaceData(h,'varname','workspace')
```

Description

`D = GetWorkspaceData(h,'varname','workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`.

Use `GetWorkspaceData` instead of `GetFullMatrix` and `GetCharArray` to get numeric and character array data, respectively. Do *not* use `GetWorkspaceData` on sparse arrays, structures, or function handles.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

This example uses a Visual Basic .NET client to read data from a MATLAB Automation server:

```
Dim Matlab As Object
Dim C2 As Variant
Dim Result As String
Matlab = CreateObject("matlab.application")
Result = MatLab.Execute("C1 = {25.72, 'hello', rand(4)};")
MsgBox("In MATLAB, type" & vbCrLf & "C1")
Matlab.GetWorkspaceData("C1", "base", C2)
MsgBox("second value of C1 = " & C2(0, 1))
```

See Also

PutWorkspaceData | GetFullMatrix | GetCharArray | GetVariable | Execute

Introduced before R2006a

ginput

Graphical input from mouse or cursor

Syntax

```
[x,y] = ginput(n)
[x,y] = ginput
[x,y,button] = ginput(...)
```

Description

`ginput` raises crosshairs in the current axes to for you to identify points in the figure, positioning the cursor with the mouse. The figure must have focus before `ginput` can receive input. If it has no axes, one is created upon the first click or keypress.

`[x,y] = ginput(n)` enables you to identify `n` points from the current axes and returns their *x*- and *y*-coordinates in the `x` and `y` column vectors. Press the **Return** key to terminate the input before entering `n` points. Specify `n` as a positive integer.

`[x,y] = ginput` gathers an unlimited number of points until you press the **Return** key.

`[x,y,button] = ginput(...)` returns the *x*-coordinates, the *y*-coordinates, and the button or key designation. `button` is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right), or ASCII numbers indicating which keys on the keyboard you pressed.

Clicking an axes makes that axes the current axes. Even if you set the current axes before calling `ginput`, whichever axes you click becomes the current axes and `ginput` returns points relative to that axes. If you select points from multiple axes, the results returned are relative to the coordinate system of the axes they come from.

Note: MATLAB returns errors such as the following if you start MATLAB with the `–noFigureWindows` or `–nodisplay` flag and then run `ginput`:

```
Error using ginput (line 31)
Terminal mode is no longer supported
```

Definitions

Coordinates returned by `ginput` are scaled to the `XLim` and `YLim` bounds of the axes you click (data units). Setting the axes or figure `Units` property has no effect on the output from `ginput`. You can click anywhere within the figure canvas to obtain coordinates. If you click outside the axes limits, `ginput` extrapolates coordinate values so they are still relative to the axes origin.

The figure `CurrentPoint` property, by contrast, is always returned in figure Units, irrespective of axes Units or limits.

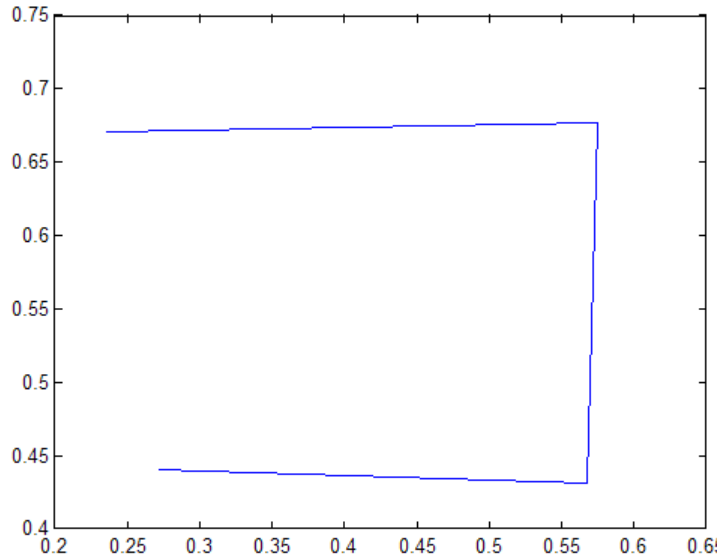
Examples

Pick 4 two-dimensional points from the figure window.

```
[x,y] = ginput(4)
```

Position the cursor with the mouse. Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 4 points, press the **Return** key.

```
x =
    0.2362
    0.5749
    0.5680
    0.2707

y =
    0.6711
    0.6769
    0.4313
    0.4401

plot(x,y)
```



In this example, `plot` rescaled the axes *x*-limits and *y*-limits from `[0 1]` and `[0 1]` to `[0.20 0.65]` and `[0.40 0.75]`. The rescaling occurred because the axes `XLimMode` and `YLimMode` are set to `'auto'` (the default). Consider setting `XLimMode` and `YLimMode` to `'manual'` if you want to maintain consistency when you gather results from `ginput` and plot them together.

See Also

`gtext`

Introduced before R2006a

global

Declare variables as global

Syntax

```
global var1 ... varN
```

Description

`global var1 ... varN` declares variables `var1 ... varN` as global in scope.

Ordinarily, each MATLAB function has its own local variables, which are separate from those of other functions and from those of the base workspace. However, if several functions all declare a particular variable name as `global`, then they all share a single copy of that variable. Any change of value to that variable, in any function, is visible to all the functions that declare it as global.

If the global variable does not exist the first time you issue the `global` statement, it is initialized to an empty `0x0` matrix.

If a variable with the same name as the global variable already exists in the current workspace, MATLAB issues a warning and changes the value of that variable and its scope to match the global variable.

Examples

Share Global Variable Between Functions

Create a function in your current working folder that sets the value of a global variable.

```
function setGlobalx(val)
global x
x = val;
```

Create a function in your current working folder that returns the value of a global variable. These two functions have separate function workspaces, but they both can access the global variable.

```
function r = getGlobalx
global x
r = x;
```

Set the value of the global variable, `x`, and obtain it from a different workspace.

```
setGlobalx(1138)
r = getGlobalx
```

```
r =

        1138
```

Share Global Variable Between Function and Command Line

Assign a value to the global variable using the function that you defined in the previous example.

```
clear all
setGlobalx(42)
```

Display the value of the global variable, `x`. Even though the variable is global, it is not accessible at the command line.

```
x
```

```
Undefined function or variable 'x'.
```

Declare `x` as a global variable at the command line, and display its value.

```
global x
x
```

```
x =

    42
```

Change the value of `x` and use the function that you defined in the previous example to return the global value from a different workspace.

```
x = 1701;
r = getGlobalx
```

```
r =
```

1701

More About

Tips

- To clear a global variable from all workspaces, use `clear global` *`variable`*.
- To clear a global variable from the current workspace but not other workspaces, use `clear` *`variable`*.
- "Share Data Between Workspaces"

See Also

`clear` | `persistent` | `who`

Introduced before R2006a

gmres

Generalized minimum residual method (with restarts)

Syntax

```
x = gmres(A,b)
gmres(A,b,restart)
gmres(A,b,restart,tol)
gmres(A,b,restart,tol,maxit)
gmres(A,b,restart,tol,maxit,M)
gmres(A,b,restart,tol,maxit,M1,M2)
gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag] = gmres(A,b,...)
[x,flag,relres] = gmres(A,b,...)
[x,flag,relres,iter] = gmres(A,b,...)
[x,flag,relres,iter,resvec] = gmres(A,b,...)
```

Description

`x = gmres(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x)` returns `A*x`. For this syntax, `gmres` does not restart; the maximum number of iterations is `min(n,10)`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `gmres` converges, a message to that effect is displayed. If `gmres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`gmres(A,b,restart)` restarts the method every `restart` inner iterations. The maximum number of outer iterations is `min(n/restart,10)`. The maximum number

of total iterations is `restart*min(n/restart,10)`. If `restart` is `n` or `[]`, then `gmres` does not restart and the maximum number of total iterations is `min(n,10)`.

`gmres(A,b,restart,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `gmres` uses the default, `1e-6`.

`gmres(A,b,restart,tol,maxit)` specifies the maximum number of outer iterations, i.e., the total number of iterations does not exceed `restart*maxit`. If `maxit` is `[]` then `gmres` uses the default, `min(n/restart,10)`. If `restart` is `n` or `[]`, then the maximum number of total iterations is `maxit` (instead of `restart*maxit`).

`gmres(A,b,restart,tol,maxit,M)` and `gmres(A,b,restart,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `gmres` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`gmres(A,b,restart,tol,maxit,M1,M2,x0)` specifies the first initial guess. If `x0` is `[]`, then `gmres` uses the default, an all-zero vector.

`[x,flag] = gmres(A,b,...)` also returns a convergence flag:

`flag = 0`	`gmres` converged to the desired tolerance `tol` within `maxit` outer iterations.
`flag = 1`	`gmres` iterated `maxit` times but did not converge.
`flag = 2`	Preconditioner `M` was ill-conditioned.
`flag = 3`	`gmres` stagnated. (Two consecutive iterates were the same.)

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = gmres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`. The third output, `relres`, is the relative residual of the preconditioned system.

`[x,flag,relres,iter] = gmres(A,b,...)` also returns both the outer and inner iteration numbers at which `x` was computed, where `0 <= iter(1) <= maxit` and `0 <= iter(2) <= restart`.

`[x,flag,relres,iter,resvec] = gmres(A,b,...)` also returns a vector of the residual norms at each inner iteration. These are the residual norms for the preconditioned system.

Examples

Using gmres with a Matrix Input

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = gmres(A,b,10,tol,maxit,M1);
```

displays the following message:

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

Using gmres with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in a function `run_gmres` that

- Calls `gmres` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_gmres` are available to `afun` and `mfun`.

The following shows the code for `run_gmres`:

```
function x1 = run_gmres
n = 21;
b = afun(ones(n,1));
tol = 1e-12;  maxit = 15;
x1 = gmres(@afun,b,10,tol,maxit,@mfun);
```

```
    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_gmres;
```

MATLAB software displays the message

```
gmres(10) converged at outer iteration 2 (inner iteration 10)
to a solution with relative residual 1.1e-013.
```

Using a Preconditioner without Restart

This example demonstrates the use of a preconditioner without restarting `gmres`.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

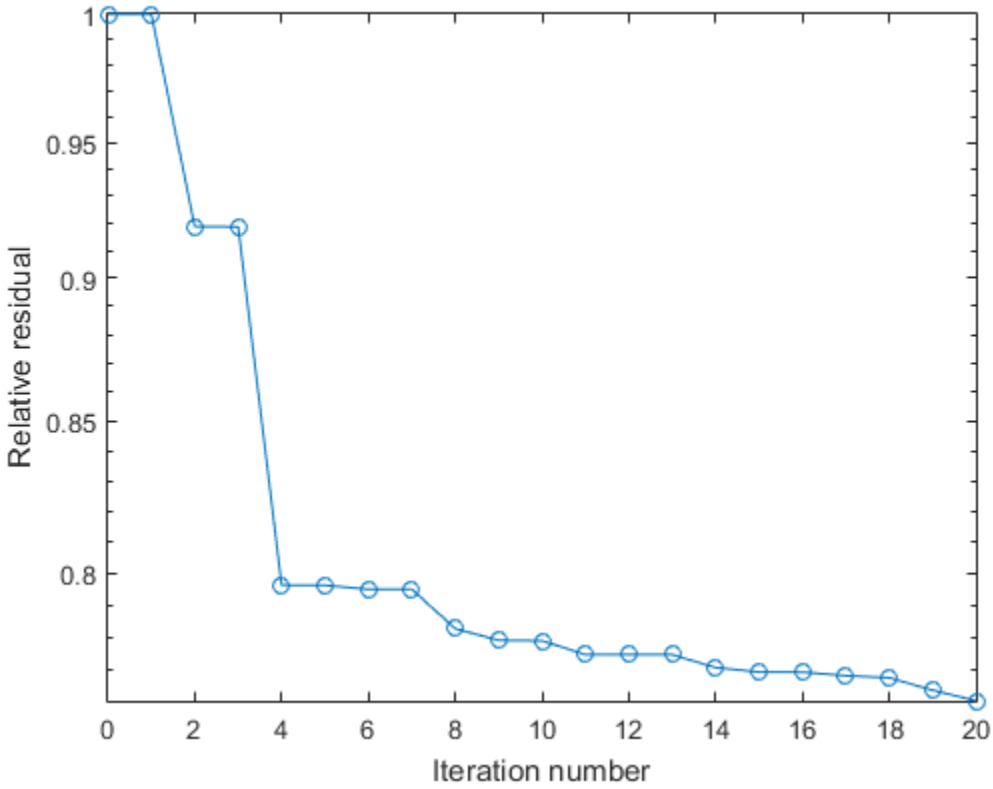
Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
[x0,fl0,rr0,it0,rv0] = gmres(A,b,[],tol,maxit);
```

`fl0` is 1 because `gmres` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The best approximate solution that `gmres` returns is the last one (as indicated by `it0(2) = 20`). MATLAB stores the residual history in `rv0`.

Plot the behavior of `gmres`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution converges slowly. A preconditioner may improve the outcome.

Use `ilu` to form the preconditioner, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
```

```
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

Note MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

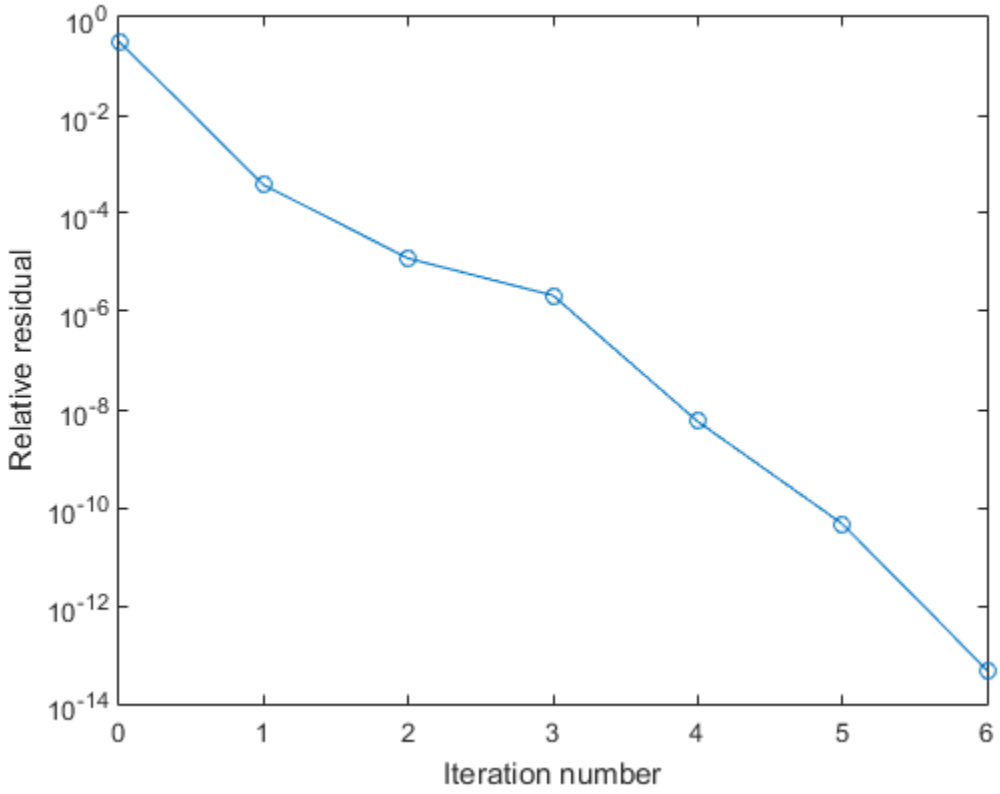
As indicated by the error message, try again with a reduced drop tolerance.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = gmres(A,b,[],tol,maxit,L,U);
```

`fl1` is 0 because `gmres` drives the relative residual to `9.5436e-14` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1(2)`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output, `rv1(1)`, is `norm(M\b)`, where `M = L*U`. The output, `rv1(7)`, is `norm(U\(L\(b-A*x1)))`.

Follow the progress of `gmres` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1(2),rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



Using a Preconditioner with Restart

This example demonstrates the use of a preconditioner with restarted `gmres`.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Construct an incomplete LU preconditioner as in the previous example.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
```

The benefit to using restarted `gmres` is to limit the amount of memory required to execute the method. Without restart, `gmres` requires `maxit` vectors of storage to keep the basis of the Krylov subspace. Also, `gmres` must orthogonalize against all of the previous vectors at each step. Restarting limits the amount of workspace used and the amount of work done per outer iteration. Note that even though preconditioned `gmres` converged in six iterations above, the algorithm allowed for as many as twenty basis vectors and therefore, allocated all of that space up front.

Execute `gmres(3)`, `gmres(4)`, and `gmres(5)`

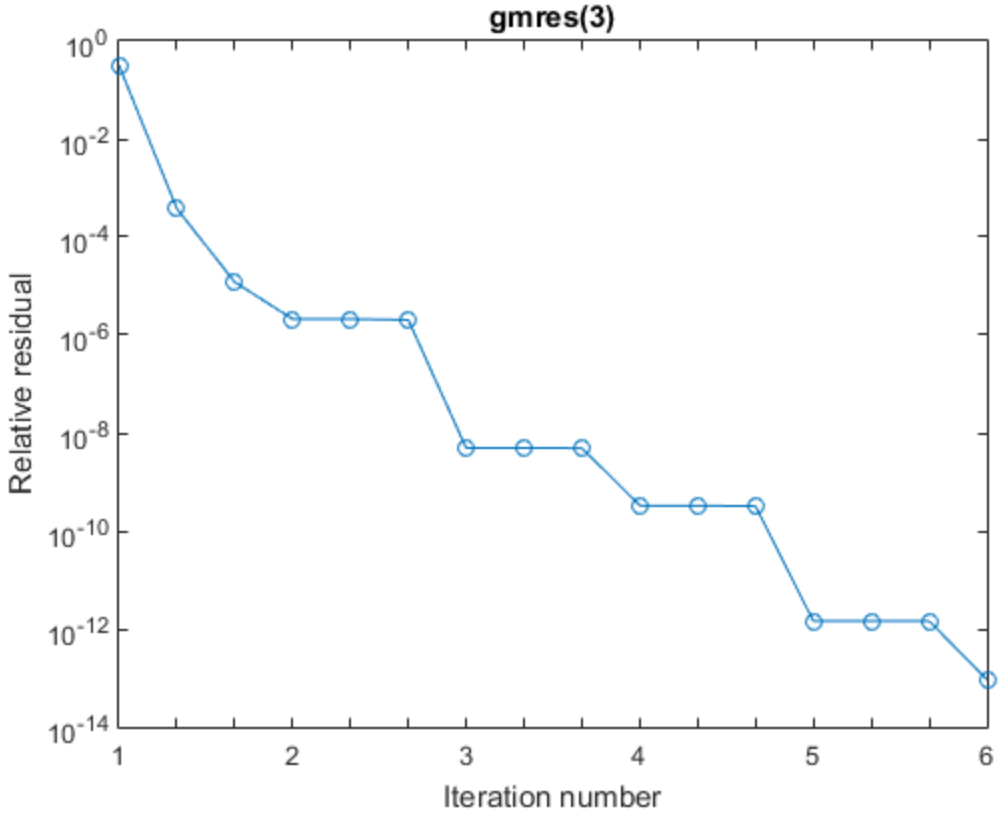
```
tol = 1e-12;
maxit = 20;
re3 = 3;
[x3,fl3,rr3,it3,rv3] = gmres(A,b,re3,tol,maxit,L,U);
re4 = 4;
[x4,fl4,rr4,it4,rv4] = gmres(A,b,re4,tol,maxit,L,U);
re5 = 5;
[x5,fl5,rr5,it5,rv5] = gmres(A,b,re5,tol,maxit,L,U);
```

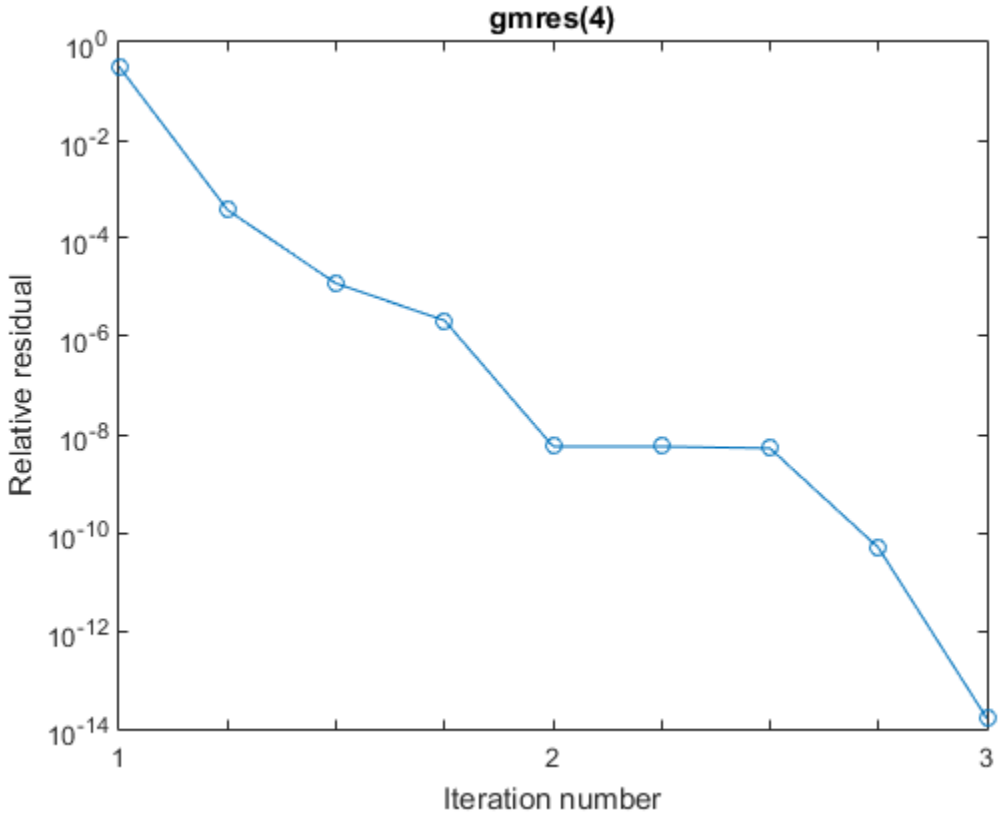
`fl3`, `fl4`, and `fl5` are all 0 because in each case restarted `gmres` drives the relative residual to less than the prescribed tolerance of `1e-12`.

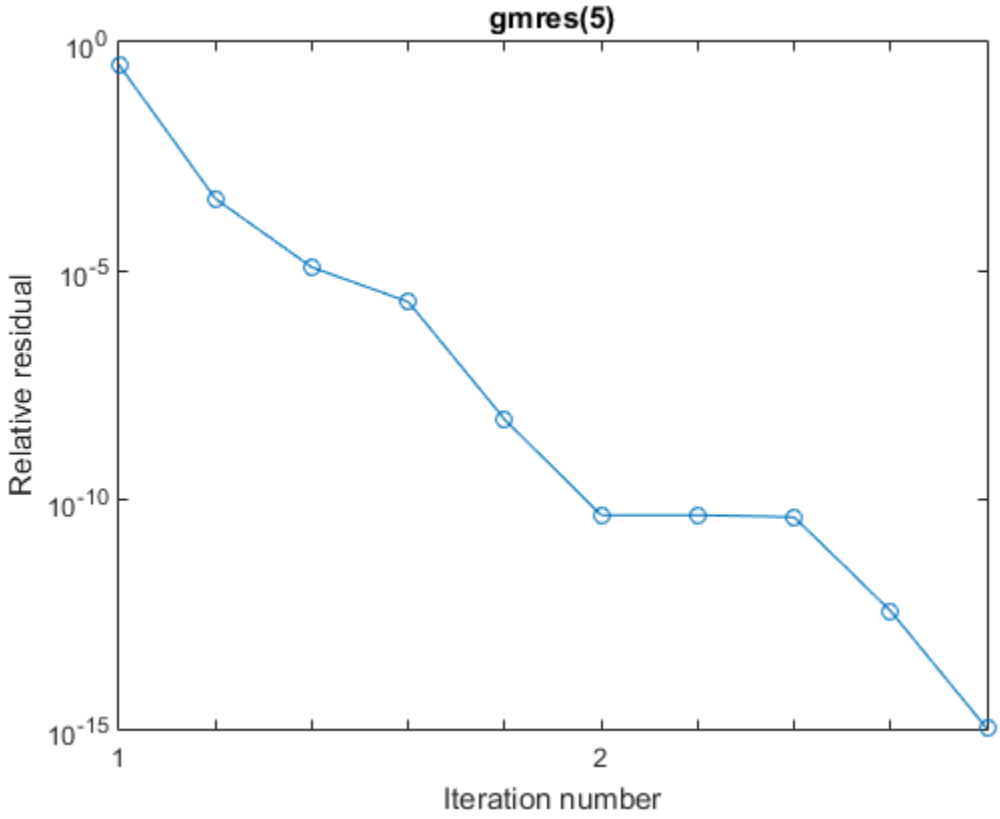
The following plots show the convergence histories of each restarted `gmres` method. `gmres(3)` converges at outer iteration 5, inner iteration 3 (`it3 = [5, 3]`) which would be the same as outer iteration 6, inner iteration 0, hence the marking of 6 on the final tick mark.

```
figure
semilogy(1:1/3:6,rv3/norm(b),'-o');
h1 = gca;
h1.XTick = [1:1/3:6];
h1.XTickLabel = ['1';' ';' ';'2';' ';' ';'3';' ';' ';'4';' ';' ';'5';' ';' ';'6';];
title('gmres(3)')
xlabel('Iteration number');
ylabel('Relative residual');

figure
semilogy(1:1/4:3,rv4/norm(b),'-o');
h2 = gca;
h2.XTick = [1:1/4:3];
h2.XTickLabel = ['1';' ';' ';' ';'2';' ';' ';' ';'3'];
```

```
title('gmres(4)')
xlabel('Iteration number');
ylabel('Relative residual');

figure
semilogy(1:1/5:2.8,rv5/norm(b),'-o');
h3 = gca;
h3.XTick = [1:1/5:2.8];
h3.XTickLabel = ['1';' ';' ';' ';' ';'2';' ';' ';' ';' '];
title('gmres(5)')
xlabel('Iteration number');
ylabel('Relative residual');
```




gmres(4)
Relative residual
Iteration number
10^0
10^-2
10^-4
10^-6
10^-8
10^-10
10^-12
10^-14
1
2
3



References

Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Saad, Youcef and Martin H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, July 1986, Vol. 7, No. 3, pp. 856-869.

See Also

`bicg` | `bicgstab` | `cgs` | `function_handle` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Introduced before R2006a

gobjects

Initialize array for graphics objects

Syntax

```
H = gobjects(n)
H = gobjects(s1,...,sn)
H = gobjects(v)

H = gobjects
H = gobjects(0)
```

Description

`H = gobjects(n)` returns an n-by-n graphics object array. Use the `gobjects` function instead of the `ones` or `zeros` functions to preallocate an array to store graphics objects.

`H = gobjects(s1,...,sn)` returns an `s1`-by-...-by-`sn` graphics object array, where the list of integers s1,...,sn defines the dimensions of the array. For example, `gobjects(2,3)` returns a 2-by-3 array.

`H = gobjects(v)` returns a graphics object array where the elements of the row vector, `v`, define the dimensions of the array. For example, `gobjects([2,3,4])` returns a 2-by-3-by-4 array.

`H = gobjects` returns a 1-by-1 graphics object array.

`H = gobjects(0)` returns an empty graphics object array.

Examples

Specify Array Dimensions

Preallocate a 4-by-1 array to store graphics handles.

```
H = gobjects(4,1)
```

```
H =

  4x1 GraphicsPlaceholder array:

  GraphicsPlaceholder
  GraphicsPlaceholder
  GraphicsPlaceholder
  GraphicsPlaceholder
```

Specify Array Dimensions with Size of Existing Array

Create an array to store graphics handles using the size of an existing array.

Define A as a 3-by-4 array.

```
A = [1,2,3,2; 4,5,6,6; 7,8,9,7];
```

Create an array of graphics handles using the size of A.

```
v = size(A);
H = gobjects(v);
```

The dimensions of the graphics handle array are the same as the dimensions of A.

```
isequal(size(H),size(A))
```

```
ans =

     1
```

Return Empty Handle Array

Use the `gobjects` function to return an empty array.

```
H = gobjects(0)
```

```
H =

  0x0 empty GraphicsPlaceholder array.
```

Input Arguments

n — defines n-by-n array

integer value

Size of the object array, specified as an integer value. Negative integers are treated as 0. The array has dimensions n-by-n.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

s1,...,sn — Size of each array dimension

two or more integer values

Size of each array dimension, specified as a list of two or more integer values. Negative integers are treated as 0.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

v — Size of each array dimension

row vector of integer values

Size of each array dimension, specified as a row vector of integer values. Negative integers are treated as 0.

Example: [2,4,6,7]

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

0 — Define an empty array

one or more dimensions equal to 0

Define an empty array by specifying one or more dimension equal to 0

Output Arguments

H — Initialized graphics object array

graphics object array of specified size

Initialized graphics object array of type `GraphicsPlaceholder`. Use this array to contain any type of graphics object.

More About

- "Graphics Arrays"

gplot

Plot nodes and links representing adjacency matrix

Syntax

```
gplot(A,Coordinates)
gplot(A,Coordinates,LineSpec)
```

Description

The `gplot` function graphs a set of coordinates using an adjacency matrix.

`gplot(A,Coordinates)` plots a graph of the nodes defined in `Coordinates` according to the n-by-n adjacency matrix `A`, where n is the number of nodes. `Coordinates` is an n-by-2 matrix, where n is the number of nodes and each coordinate pair represents one node.

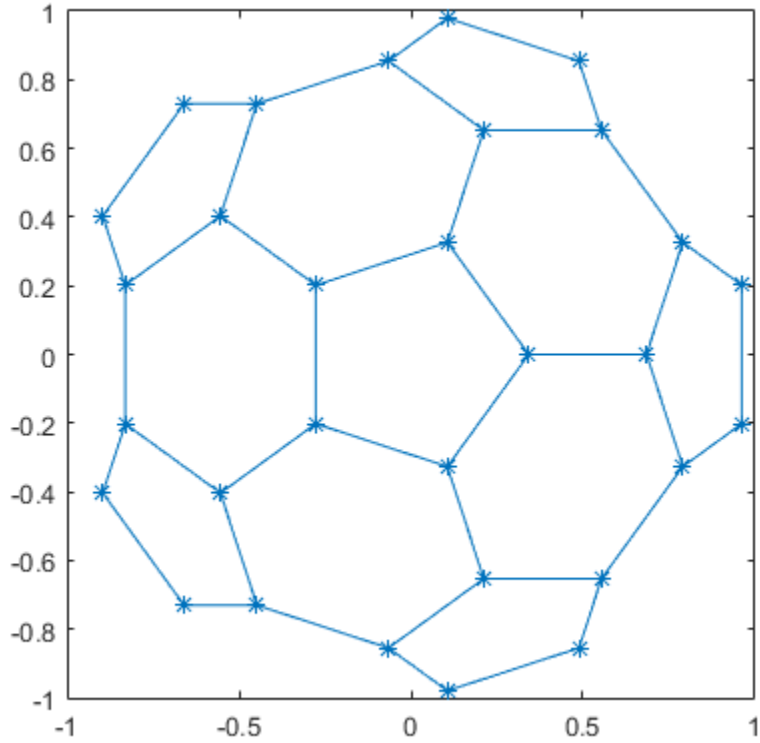
`gplot(A,Coordinates,LineSpec)` plots the nodes using the line type, marker symbol, and color specified by *`LineSpec`*.

Examples

Plot Graph of Nodes Using Asterisks

Plot half of a "Bucky ball" carbon molecule, placing asterisks at each node.

```
k = 1:30;
[B,XY] = bucky;
gplot(B(k,k),XY(k,:),'-*')
axis square
```



More About

Tips

For two-dimensional data, `Coordinates(i,:) = [x(i) y(i)]` denotes node `i`, and `Coordinates(j,:) = [x(j)y(j)]` denotes node `j`. If node `i` and node `j` are connected, `A(i,j)` or `A(j,i)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

See Also

`LineSpec` | `sparse` | `spy`

Introduced before R2006a

grabcode

Extract MATLAB code from file published to HTML

Syntax

```
grabcode('name.html')
grabcode('urlname')
codeString = grabcode('name.html')
```

Description

`grabcode('name.html')` copies MATLAB code from the file `name.html` and pastes it into an untitled document in the Editor. Use `grabcode` to get MATLAB code from published files when the source code is not readily available. The file `name.html` was created by publishing `name.m`, a MATLAB code file containing cells. The MATLAB code from `name.m` is included at the end of `name.html` as HTML comments.

`grabcode('urlname')` copies MATLAB code from the `urlname` location and pastes it into an untitled document in the Editor.

`codeString = grabcode('name.html')` gets MATLAB code from the file `name.html` and assigns it the variable `codeString`.

Examples

This example illustrates how to use `grabcode` to get MATLAB code from an existing HTML file:

```
% Copy sine_wave_f.html and the images it includes
% from the MATLAB examples directory to your
% current folder:

copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
 'examples','sine_wave_f.html'), 'my_sine_wave.html')

copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
```

```
 'examples','sine_wave_f_01.png'), 'sine_wave_f_01.png')

copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
 'examples','sine_wave_f_02.png'), 'sine_wave_f_02.png')

% If you want to view the file, double-click my_sine_wave.html
% in your current folder.

% Extract the MATLAB code from sine_wave_f.html:
code = grabcode('my_sine_wave.html')
```

MATLAB returns:

```
code =

%% Plot Sine Wave
% Calculate and plot a sine wave.

%% Calculate and Plot Sine Wave
% Calculate and plot |y = sin(x)|.

function sine_wave_f(x)

y = sin(x);
plot(x,y)

%% Modify Plot Properties

title('Sine Wave', 'FontWeight','bold')
xlabel('x')
ylabel('sin(x)')
set(gca, 'Color', 'w')
set(gcf, 'MenuBar', 'none')
```

See Also

`publish`

Introduced before R2006a

gradient

Numerical gradient

Syntax

```
FX = gradient(F)
[FX,FY] = gradient(F)
[FX,FY,FZ,...] = gradient(F)
[...] = gradient(F,h)
[...] = gradient(F,h1,h2,...)
```

Definitions

The *gradient* of a function of two variables, $F(x,y)$, is defined as

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j}$$

and can be thought of as a collection of vectors pointing in the direction of increasing values of F. In MATLAB software, numerical gradients (differences) can be computed for functions with any number of variables. For a function of N variables, $F(x,y,z, ...)$,

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} + \frac{\partial F}{\partial z}\hat{k} + \ldots$$

Description

`FX = gradient(F)`, where `F` is a vector, returns the one-dimensional numerical gradient of `F`. Here `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction.

`[FX,FY] = gradient(F)`, where `F` is a matrix, returns the x and y components of the two-dimensional numerical gradient. `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction. `FY` corresponds to $\partial F/\partial y$, the differences in the y (vertical) direction. The spacing between points in each direction is assumed to be one.

`[FX,FY,FZ,...] = gradient(F)`, where `F` has `N` dimensions, returns the `N` components of the gradient of `F`. There are two ways to control the spacing between values in `F`:

- A single spacing value, `h`, specifies the spacing between points in every direction.
- `N` spacing values (`h1,h2,...`) specifies the spacing for each dimension of `F`. Scalar spacing parameters specify a constant spacing for each dimension. Vector parameters specify the coordinates of the values along corresponding dimensions of `F`. In this case, the length of the vector must match the size of the corresponding dimension.

Note The first output `FX` is always the gradient along the 2nd dimension of `F`, going across columns. The second output `FY` is always the gradient along the 1st dimension of `F`, going across rows. For the third output `FZ` and the outputs that follow, the Nth output is the gradient along the Nth dimension of `F`.

`[...] = gradient(F,h)`, where `h` is a scalar, uses `h` as the spacing between points in each direction.

`[...] = gradient(F,h1,h2,...)` with `N` spacing parameters specifies the spacing for each dimension of `F`.

Examples

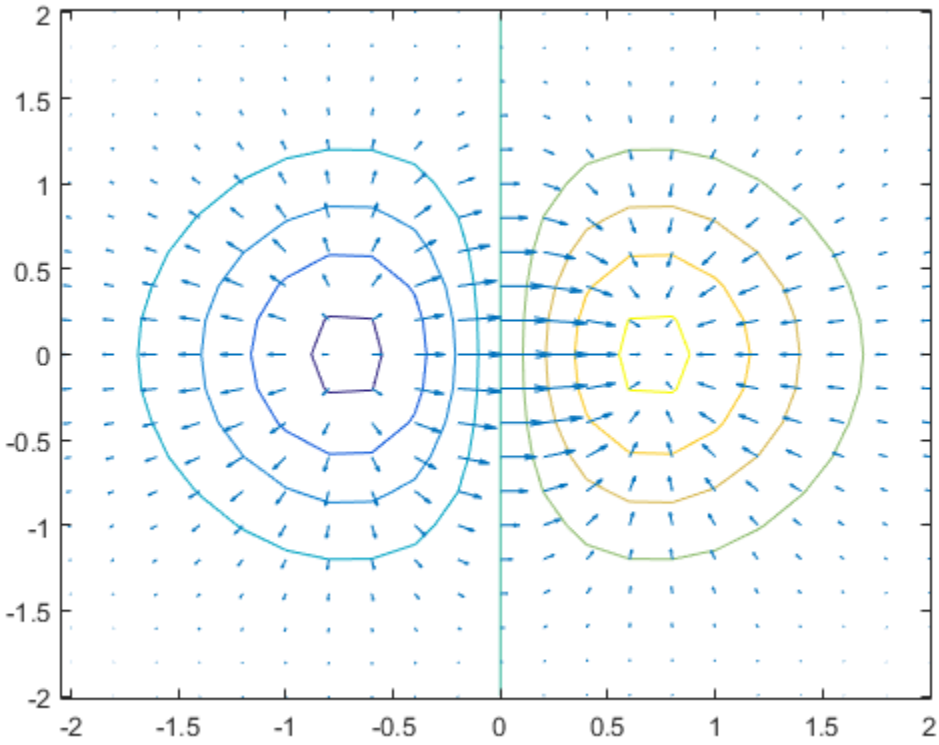
Contour Plot of Vector Field

Calculate the 2-D gradient of $xe^{-x^2-y^2}$ on a grid.

```
v = -2:0.2:2;
[x,y] = meshgrid(v);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,.2,.2);
```

Plot the contour lines and vectors in the same figure.

```
contour(v,v,z)
hold on
quiver(v,v,px,py)
hold off
```



Specify Dimensional Spacing of Points

Create a 3-D array.

```
F(:,:,1) = magic(3);
F(:,:,2) = pascal(3);
```

The command,

```
gradient(F)
```

takes `dx = dy = dz = 1`. However, the command,

```
[PX,PY,PZ] = gradient(F,0.2,0.1,0.2)
```

takes `dx = 0.2`, `dy = 0.1`, and `dz = 0.2`.

More About

Algorithms

`gradient` calculates the *central difference* for interior data points. For example, consider a matrix with unit-spaced data, `A`, that has horizontal gradient `G = gradient(A)`. The interior gradient values, `G(:,j)`, are:

```
G(:,j) = 0.5*(A(:,j+1) - A(:,j-1));
```

where `j` varies between `2` and `N-1`, where `N` is `size(A,2)`.

The gradient values along the edges of the matrix are calculated with *single-sided differences*, so that

```
G(:,1) = A(:,2) - A(:,1);
G(:,N) = A(:,N) - A(:,N-1);
```

If the point spacing is specified, then the differences are scaled appropriately. If two or more outputs are specified, `gradient` also calculates differences along other dimensions in a similar manner. Unlike the `diff` function, `gradient` returns an array with the same number of elements as the input.

See Also

`del2` | `diff`

Introduced before R2006a

matlab.graphics.Graphics class

Package: matlab.graphics

Common base class for graphics objects

Description

The `matlab.graphics.Graphics` class is the base class of all graphics objects. Because graphics objects are part of a heterogeneous hierarchy, you can create arrays of mixed classes (for example, an array can contain lines, surfaces, axes, and other graphics objects).

The class of an array of mixed objects is `matlab.graphics.Graphics` because this class is common to all graphics object.

Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

To learn about attributes of classes, see Class Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.graphics.GraphicsPlaceholder`

More About

- Class Attributes
- Property Attributes

matlab.graphics.GraphicsPlaceholder class

Package: matlab.graphics
Superclasses: matlab.graphics.Graphics

Default graphics object

Description

The `matlab.graphics.GraphicsPlaceholder` class defines the default graphics object. Instances of this class appear as:

- Elements of pre-allocated arrays created with `gobjects`.
- Unassigned array element placeholders
- Graphics object properties that hold object handles, but are set to empty values
- Empty values returned by functions that return object handles (for example, `findobj`).

Attributes

`Sealed`	`true`
`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

To learn about attributes of classes, see Class Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Test for Current Figure

```
if isempty(get(groot,'CurrentFigure')
   ... % There is no current figure
end
```

More About

- "Graphics Object Programming"
- Class Attributes
- Property Attributes

graymon

Set default figure properties for grayscale monitors

Syntax

```
graymon
```

Description

`graymon` sets defaults for graphics properties to produce more legible displays for grayscale monitors.

See Also

`axes` | `figure`

Introduced before R2006a

grid

Display or hide axes grid lines

Syntax

```
grid on
grid off
grid

grid minor

grid(ax, ___ )
```

Description

`grid on` displays the major grid lines for the current axes. Major grid lines extend from each tick mark.

`grid off` removes all grid lines from the current axes.

`grid` toggles the visibility of the major grid lines.

`grid minor` toggles the visibility of the minor grid lines. Minor grid lines lie between the tick marks.

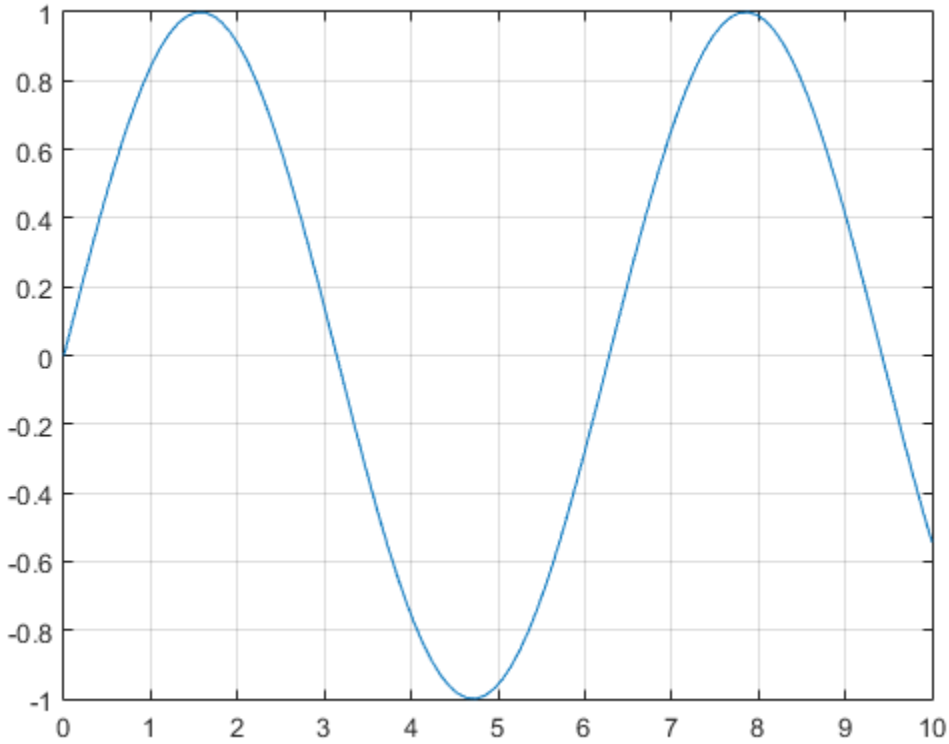
`grid(ax, ___ )` uses the axes defined by `ax` instead of the current axes (`gca`). Specify `ax` as a scalar axes object. You can specify an axes with any of the input arguments in the previous syntaxes. Use single quotes around input arguments that are character strings, for example, `grid(ax,'on')`, `grid(ax,'off')`, and `grid(ax,'minor')`.

Examples

Display Grid Lines

Display the grid lines for a sine plot.

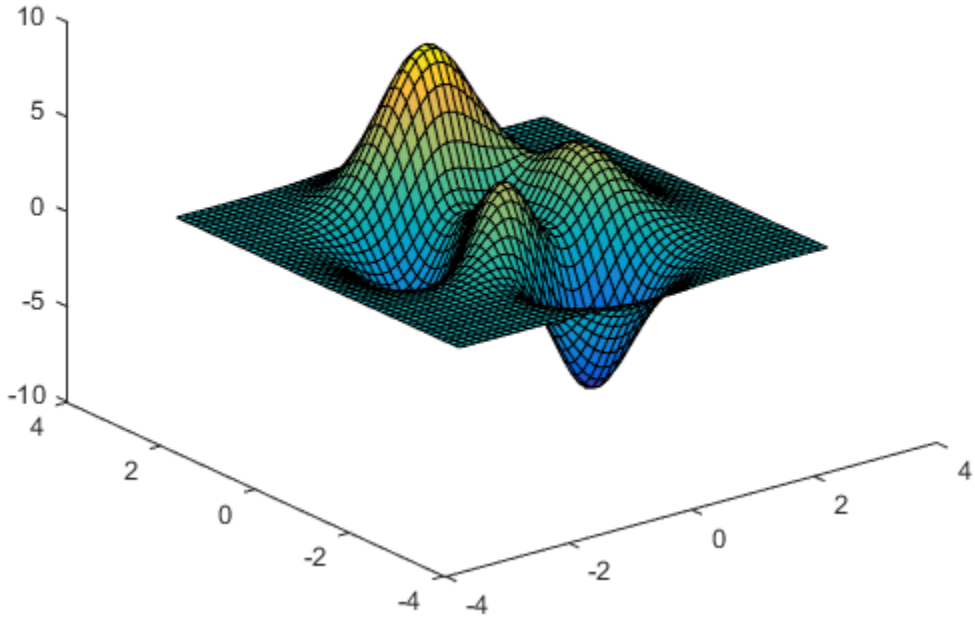
```
x = linspace(0,10);
y = sin(x);
plot(x,y)
grid on
```



Remove Grid Lines

Create a surface plot and remove the grid lines.

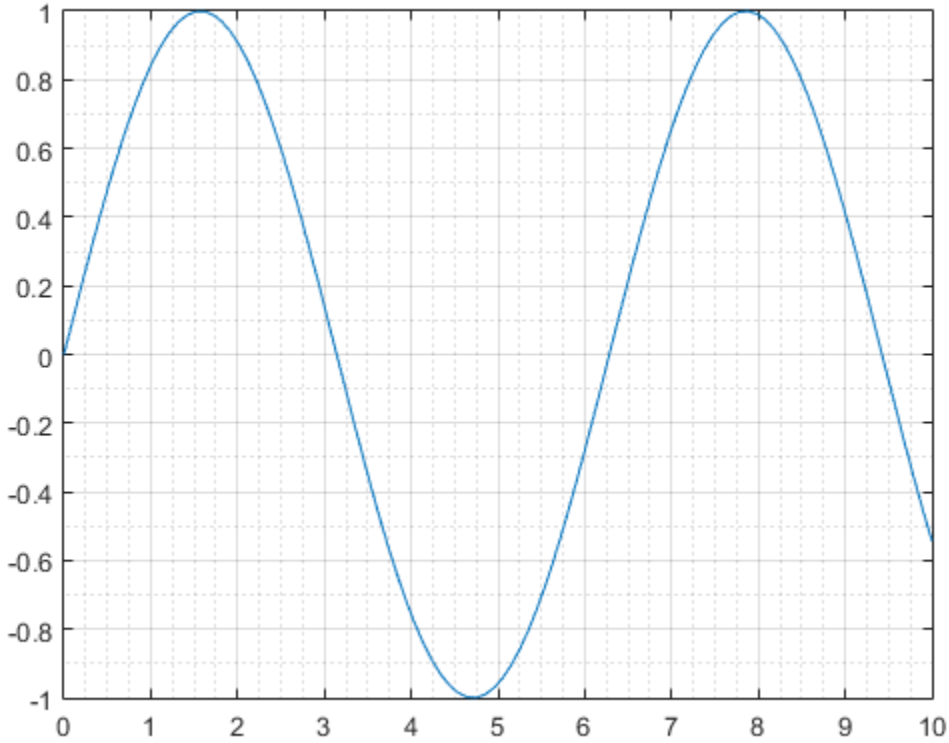
```
[X,Y,Z] = peaks;
surf(X,Y,Z)
grid off
```



Display Major and Minor Grid Lines

Display the major and minor grid lines for a sine plot.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
grid on
grid minor
```



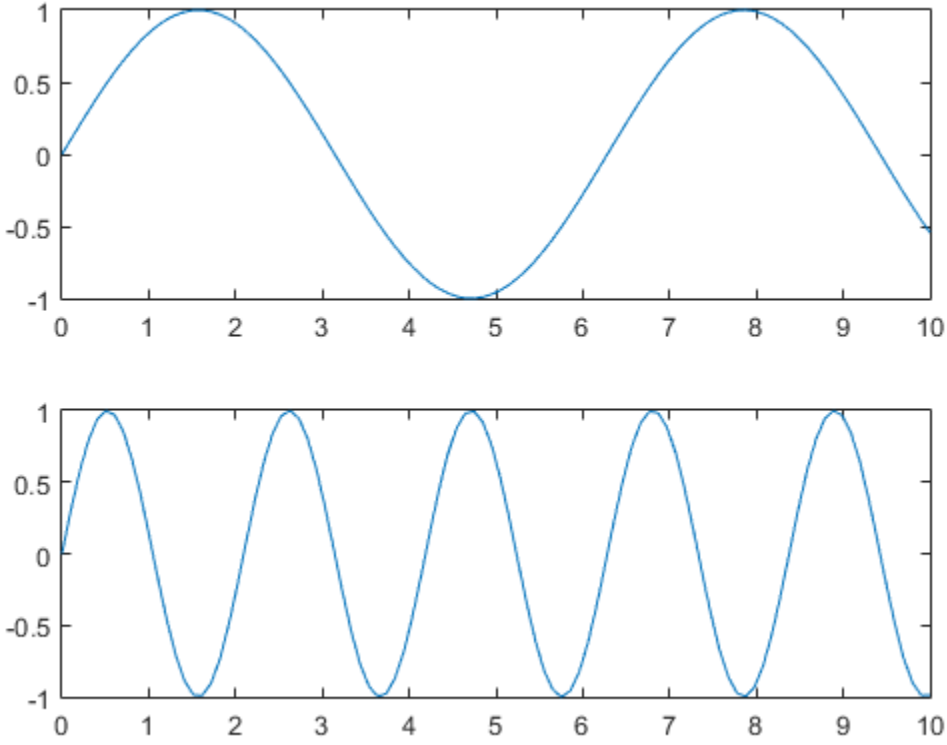
By default, major grid lines use a solid line style and align with the tick marks. Minor grid lines use a dotted line style and lie between the tick marks.

Display Grid Lines on Specific Axes

Create a figure with two subplots and plot a sine wave in each one.

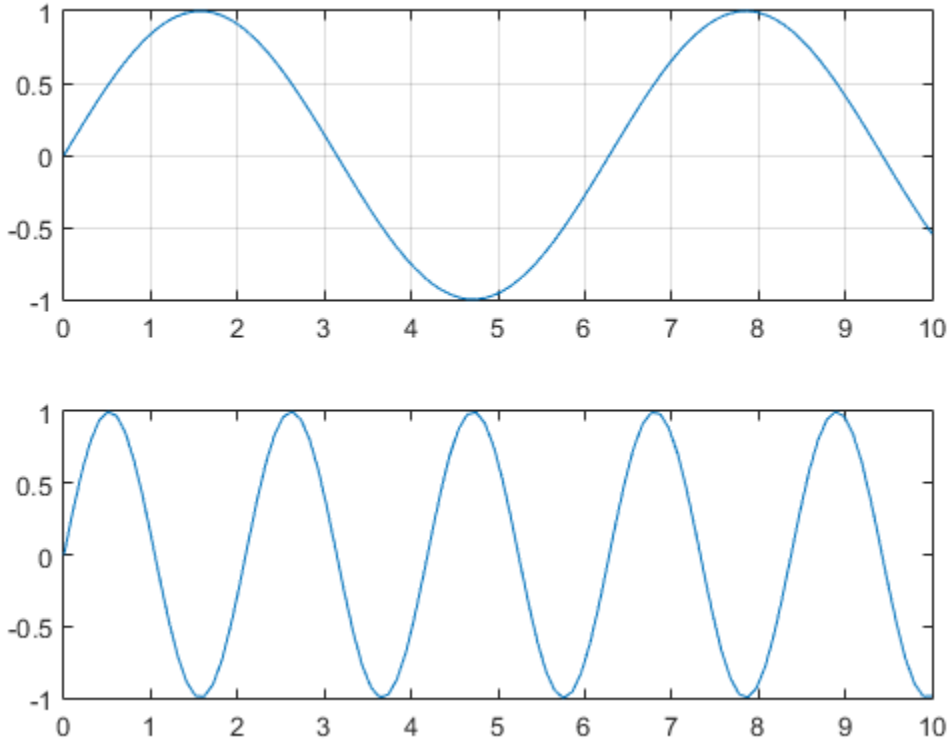
```
x = linspace(0,10);
y = sin(x);
ax1 = subplot(2,1,1);
plot(x,y)

y2 = sin(3*x);
ax2 = subplot(2,1,2);
```

```
plot(x,y2)
```



Display the grid lines on the upper subplot.

```
grid(ax1,'on')
```



More About

Tips

- Some axes properties affect the appearance of the grid lines. By changing property values, you can modify the appearance. This table lists a subset of axes properties related to the grid.

Axes Property	Description
`XTick`, `YTick`, `ZTick`	Location of tick marks and major grid lines for each axis direction

Axes Property	Description
`XGrid`, `YGrid`, `ZGrid`	Display of major grid lines for each axis direction
`XMinorGrid`, `YMinorGrid`, `ZMinorGrid`	Display of minor grid lines for each axis direction
`LineWidth`	Line width of grid lines, axes box outline, and tick marks
`GridLineStyle`	Major grid line style
`MinorGridLineStyle`	Minor grid line style
`GridColor`	Major grid line color
`MinorGridColor`	Minor grid line style
`GridAlpha`	Major grid line transparency
`MinorGridAlpha`	Minor grid line transparency
`Layer`	Location of grid lines with respect to the plotted data

For a full list, see Axes Properties.

- To turn on or off grid lines for each axis direction separately, you must set the axes properties individually. The `grid` function controls all major and minor grid lines together, setting the corresponding axes properties to either `'on'` or `'off'`.

See Also

`axis` | `box` | `legend` | `title` | `xlabel` | `ylabel`

Introduced before R2006a

griddata

Interpolate scattered data

Note: Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `griddata`.

In a future release, the following syntaxes will be removed:

```
[Xq,Yq,Vq] = griddata(x,y,v,xq,yq)
[Xq,Yq,Vq] = griddata(x,y,v,xq,yq, method)
```

In addition, `griddata` will not accept any input vectors of mixed orientation in a future release. To specify a grid of query points, construct a full grid with `ndgrid` or `meshgrid` before calling `griddata`.

Syntax

```
vq = griddata(x,y,v,xq,yq)
vq = griddata(x,y,z,v,xq,yq,zq)
vq = griddata( ___ ,method)
```

Description

`vq = griddata(x,y,v,xq,yq)` fits a surface of the form $v = f(x,y)$ to the scattered data in the vectors `(x,y,v)`. The `griddata` function interpolates the surface at the query points specified by `(xq,yq)` and returns the interpolated values, `vq`. The surface always passes through the data points defined by `x` and `y`.

`vq = griddata(x,y,z,v,xq,yq,zq)` fits a hypersurface of the form $v = f(x,y,z)$.

`vq = griddata( ___ ,method)` uses a specified interpolation method to compute vq using any of the input arguments in the previous syntaxes.

Input Arguments

x

Vector specifying the *x*- coordinates of the sample points.

y

Vector specifying the *y*- coordinates of the sample points.

z

Vector specifying the *z*- coordinates of the sample points.

v

Vector of sample values that correspond to the sample coordinates `x`, `y` (and `z` for 3-D interpolation).

xq

Vector or array that specifies *x*- coordinates of the query points to be evaluated. `xq` must be the same size as `yq` (and `zq` for 3-D interpolation).

- Specify an array if you want to pass a grid of query points. Use `ndgrid` or `meshgrid` to construct the array.
- Specify a vector if you want to pass a collection of scattered points.

yq

Vector or array that specifies *y*- coordinates of the query points to be evaluated. `yq` must be the same size as `xq` (and `zq` for 3-D interpolation).

- Specify an array if you want to pass a grid of query points. Use `ndgrid` or `meshgrid` to construct the array.
- Specify a vector if you want to pass a collection of scattered points.

zq

Vector or array that specifies *z*- coordinates of the query points to be evaluated. `zq` must be the same size as `xq` and `yq`.

- Specify an array if you want to pass a grid of query points. Use `ndgrid` or `meshgrid` to construct the array.

- Specify a vector if you want to pass a collection of scattered points.

`method`

Keyword that specifies the interpolation method. Use one of the following:

method string	Description	Continuity
`'nearest'`	Triangulation-based nearest neighbor interpolation supporting 2-D and 3-D interpolation.	Discontinuous
`'linear'`	Triangulation-based linear interpolation (default) supporting 2-D and 3-D interpolation	C^0
`'natural'`	Triangulation-based natural neighbor interpolation supporting 2-D and 3-D interpolation. This method is an efficient tradeoff between linear and cubic.	C^1 except at sample points
`'cubic'`	Triangulation-based cubic interpolation supporting 2-D interpolation only	C^2
`'v4'`	Biharmonic spline interpolation (MATLAB 4 `griddata` method) supporting 2-D interpolation only. Unlike the other methods, this interpolation is not based on a triangulation.	C^2

Output Arguments

`vq`

The interpolated values at the query points.

- For 2-D interpolation, where `xq` and `yq` specify an `m`-by-`n` grid of query points, `vq` is an `m`-by-`n` array.
- For 3-D interpolation, where `xq`, `yq`, and `zq` specify an `m`-by-`n`-by-`p` grid of query points, `vq` is an `m`-by-`n`-by-`p` array.
- If `xq`, `yq`, (and `zq` for 3-D interpolation) are vectors that specify scattered points, `vq` is a vector of the same length.

Examples

Interpolate Scattered Data Over a Uniform Grid

Sample a function at 200 random points between `-2.5` and `2.5`.

```
xy = -2.5 + 5*gallery('uniformdata',[200 2],0);
x = xy(:,1);
y = xy(:,2);
v = x.*exp(-x.^2-y.^2);
```

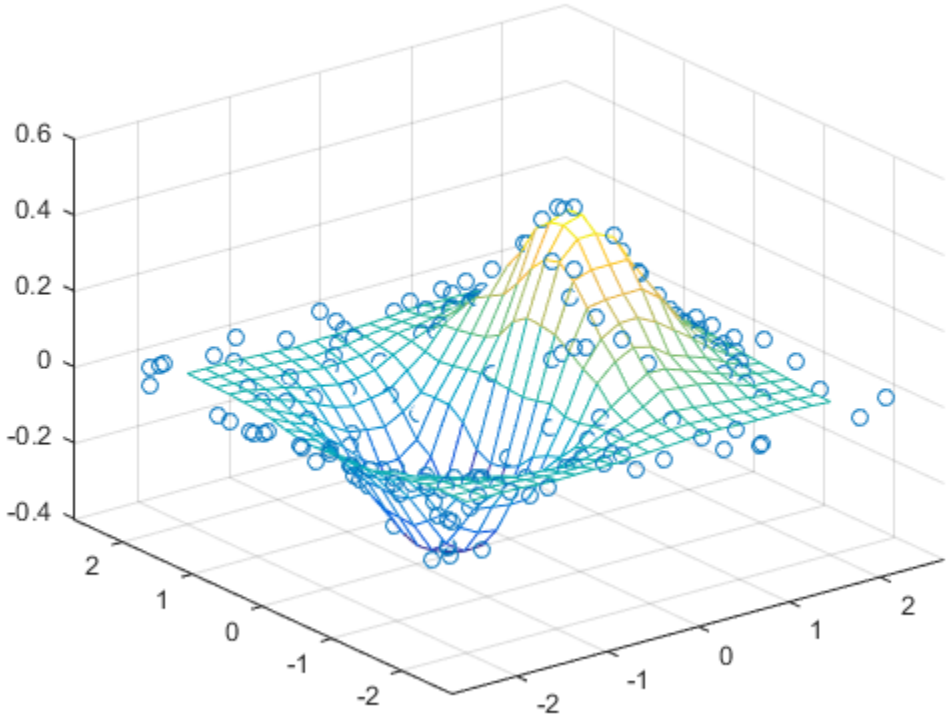
`x`, `y`, and `v` are vectors containing scattered (nonuniform) sample points and data.

Define a regular grid and interpolate the scattered data over the grid.

```
[xq,yq] = meshgrid(-2:.2:2, -2:.2:2);
vq = griddata(x,y,v,xq,yq);
```

Plot the gridded data as a mesh and the scattered data as dots.

```
figure
mesh(xq,yq,vq);
hold on
plot3(x,y,v,'o');

h = gca;
h.XLim = [-2.7 2.7];
h.YLim = [-2.7 2.7];
```



Interpolate 3-D Data Set Over a Grid in the x-y Plane

Sample a function at 5000 random points between `-1` and `1`. Initialize the random number generator to make the output of `randn` repeatable.

```
rng default
x = 2*rand(5000,1) - 1;
y = 2*rand(5000,1) - 1;
z = 2*rand(5000,1) - 1;
v = x.^2 + y.^2 + z.^2;
```

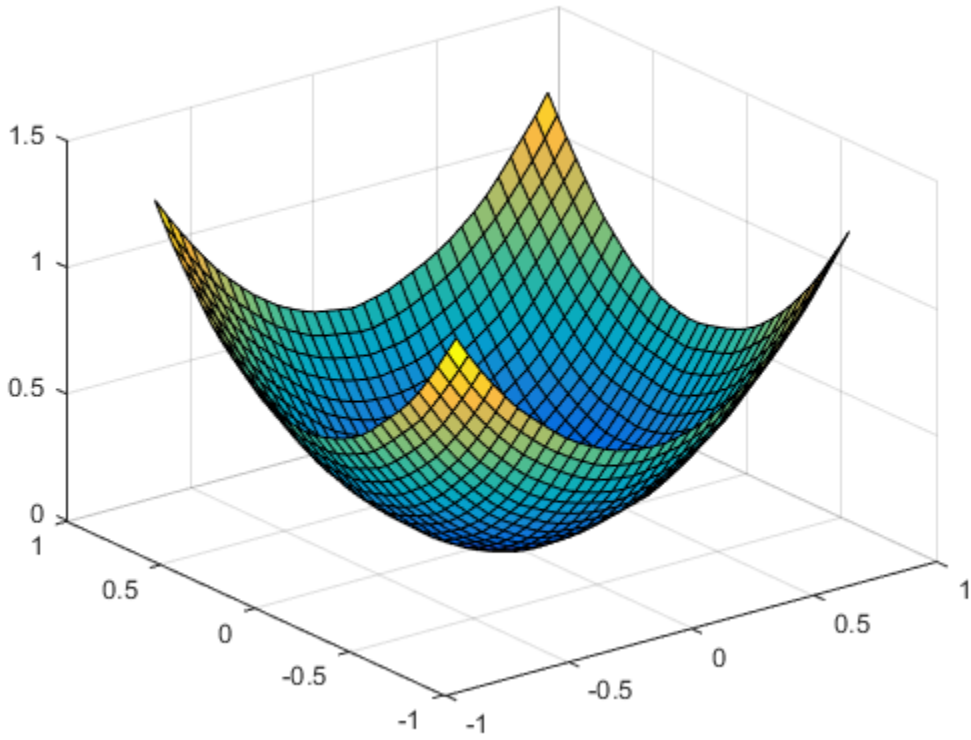
`x`, `y`, and `z` are now vectors containing nonuniformly sampled data.

Define a regular grid with points in the range [-0.8, 0.8].

```
d = -0.8:0.05:0.8;
[xq,yq,zq] = meshgrid(d,d,0);
```

Interpolate the scattered data over a rectangular region at `z=0`. Then, plot the results.

```
vq = griddata(x,y,z,v,xq,yq,zq);
surf(xq,yq,vq);

h = gca;
h.XTick = [-1 -0.5 0 0.5 1];
h.YTick = [-1 -0.5 0 0.5 1];
```



See Also

`scatteredInterpolant` | `delaunay` | `griddatan` | `interpn` | `meshgrid` | `ndgrid`

Introduced before R2006a

griddatan

Data gridding and hypersurface fitting (dimension ≥ 2)

Syntax

```
yi = griddatan(x,y,xi)
yi = griddatan(x,y,xi,method)
yi = griddatan(x,y,xi,method,options)
```

Description

`yi = griddatan(x,y,xi)` fits a hyper-surface of the form $y = f(x)$ to the data in the (usually) nonuniformly-spaced vectors (`x`, `y`). `griddatan` interpolates this hyper-surface at the points specified by `xi` to produce `yi`. `xi` can be nonuniform.

`X` is of dimension `m`-by-`n`, representing `m` points in n-dimensional space. `y` is of dimension `m`-by-`1`, representing `m` values of the hyper-surface *f*(`X`). `xi` is a vector of size `p`-by-`n`, representing `p` points in the n-dimensional space whose surface value is to be fitted. `yi` is a vector of length `p` approximating the values *f*(`xi`). The hypersurface always goes through the data points (`X`,`y`). `xi` is usually a uniform grid (as produced by `meshgrid`).

`yi = griddatan(x,y,xi,method)` defines the type of surface fit to the data, where `'method'` is one of:

method string	Description	Continuity
`'linear'`	Triangulation-based linear interpolation (default).	C^0
`'nearest'`	Nearest neighbor interpolation.	Discontinuous

All the methods are based on a Delaunay triangulation of the data.

If `method` is `[]`, the default `'linear'` method is used.

`yi = griddatan(x,y,xi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default.

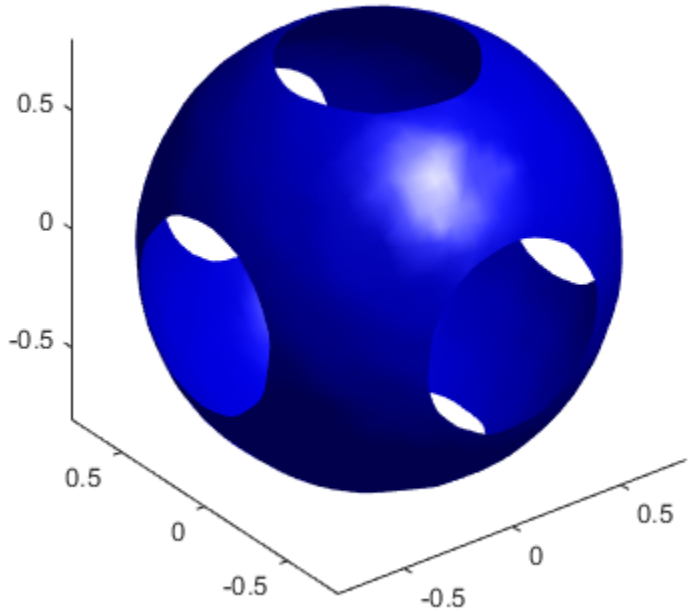
Examples

Fit a Hypersurface

```
X = 2*gallery('uniformdata',[5000 3],0)-1;;
Y = sum(X.^2,2);
d = -0.8:0.05:0.8;
[y0,x0,z0] = ndgrid(d,d,d);
XI = [x0(:) y0(:) z0(:)];
YI = griddatan(X,Y,XI);
```

Since it is difficult to visualize 4-D data sets, use isosurface at 0.8:

```
YI = reshape(YI, size(x0));
figure
p = patch(isosurface(x0,y0,z0,YI,0.8));
isonormals(x0,y0,z0,YI,p)

p.FaceColor = 'blue';
p.EdgeColor = 'none';

view(3)
axis equal
camlight
lighting phong
```



See Also

`delaunayn` | `griddata` | `meshgrid`

Introduced before R2006a

griddedInterpolant class

Gridded data interpolation

Note: The behavior of `griddedInterpolant` has changed. All interpolation methods now support extrapolation by default. Set `F.ExtrapolationMethod` to `'none'` to preserve the pre-R2013a behavior when `F.Method` is `'linear'`, `'cubic'` or `'nearest'`. Before R2013a, evaluation returned `NaN` values at query points outside the domain when `F.Method` was set to `'linear'`, `'cubic'` or `'nearest'`.

Description

Use `griddedInterpolant` to perform interpolation on a 1-D, 2-D, 3-D, or N-D "Gridded Data" on page 1-3067 set. For example, you can pass a set of `(x,y)` points and values, `v`, to `griddedInterpolant`, and it returns a surface of the form $v = F(x, y)$. This surface always passes through the sample values at the point locations. You can evaluate this surface at any query point, `(xq,yq)`, to produce an interpolated value, `vq`.

Use `griddedInterpolant` to create the "Interpolant" on page 1-3066, `F`. Then you can evaluate `F` at specific points using any of the following syntaxes:

- `Vq = F(Xq)` evaluates `F` at a set of query points in matrix `Xq`. The points in `Xq` are scattered, and each row of `Xq` contains the coordinates of a query point.
- `Vq = F(xq1,xq2,...,xqn)` specifies the query locations, `xq1,xq2,...,xqn`, as column vectors of length `m` representing `m` points scattered in `n`-dimensional space.
- `Vq = F(Xq1,Xq2,...,Xqn)` specifies the query locations as `n` `n`-dimensional arrays, `Xq1,Xq2,...,Xqn`, of equal size which define a "Full Grid" on page 1-3067 of points.
- `Vq = F({xgq1,xgq2,...,xgqn})` specifies the query locations as "Grid Vectors" on page 1-3067 . Use this syntax to conserve memory when you want to query a large grid of points.

Construction

`F = griddedInterpolant(x,v)` creates a 1-D interpolant from a vector of sample points, `x`, and corresponding values, `v`.

`F = griddedInterpolant(X1,X2,...,Xn,V)` creates a 2-D, 3-D, or N-D interpolant using a "Full Grid" on page 1-3067 of sample points passed as a set of `n`-dimensional arrays, `X1,X2,...,Xn`. The V array contains the sample values associated with the point locations in `X1,X2,...,Xn`. Each of the arrays, `X1,X2,...,Xn` must be the same size as `V`.

`F = griddedInterpolant(V)` uses the default grid to create the interpolant. When you use this syntax, `griddedInterpolant` defines the grid as set of points whose spacing is `1` and range is [`1`, `size(V,i)`] in the `i`th dimension. Use this syntax when you want to conserve memory and are not concerned about the absolute distances between points.

`F = griddedInterpolant({xg1,xg2,...,xgn},V)` specifies `n` "Grid Vectors" on page 1-3067 to describe an `n`-dimensional grid of sample points. Use this syntax when you want to use a specific grid and also conserve memory.

`F = griddedInterpolant(___,Method)` specifies a string that describes an interpolation method: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, or `'spline'`. You can specify Method as the last input argument in any of the first four syntaxes.

`F = griddedInterpolant(___,Method,ExtrapolationMethod)` specifies both the interpolation and extrapolation methods as strings. `griddedInterpolant` uses `ExtrapolationMethod` to estimate the value when your query points fall outside the domain of your sample points. Specify Method and ExtrapolationMethod together as the last two input arguments in any of the first four syntaxes.

Input Arguments

x

Sample points vector, specified as a vector of input coordinates the same size as `v`.

v

Sample values vector, specified as a vector of input values the same size as `x`.

`X1,X2,...,Xn`

Sample points in "Full Grid" on page 1-3067 form, specified as a set of `n`-dimensional arrays. You can create the arrays, `X1,X2,...,Xn`, using the `ndgrid` function. These arrays are all the same size, and each one is the same size as `V`.

`{xg1,xg2,...,xgn}`

Sample points in grid vector form, specified as a cell array of grid vectors. These vectors must specify a grid that is the same size as `V`. In other words, `size(V) = [length(xg1) length(xg2),...,length(xgn)]`. Use this form as an alternative to the full grid to save memory when your grid is very large.

`V`

Sample values, specified as an array. The elements of `V` are the values that correspond to the sample points. The size of `V` must be the size of the full grid of sample points.

- If you specify the sample points as a full grid consisting of N-D arrays, then `V` must be the same size as any one of: `X1,X2,...,Xn`.
- If you specify the sample points as grid vectors, then `size(V) = [length(xg1) length(xg2) ... length(xgn)]`.

`Method`

Interpolation method, specified as a string from the table below.

Method	Description	Continuity	Comments
`'linear'`(defa	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension.	C^0	• Requires at least 2 grid points in each dimension. • Requires more memory than `'nearest'`.
`'nearest'`	Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires 2 grid points in each dimension. • Fastest computation with modest memory requirements
`'next'`	Next neighbor interpolation (for 1-D only). The interpolated	Discontinuous	• Requires at least 2 points.

Method	Description	Continuity	Comments
	value at a query point is the value at the next sample grid point.		• Same memory requirements and computation time as `'nearest'`.
`'previous'`	Previous neighbor interpolation (for 1-D only). The interpolated value at a query point is the value at the previous sample grid point.	Discontinuous	• Requires at least 2 points. • Same memory requirements and computation time as `'nearest'`.
`'pchip'`	Shape-preserving piecewise cubic interpolation (for 1-D only). The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.	C^1	• Requires at least 4 points. • Requires more memory and computation time than `'linear'`.
`'cubic'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.	C^1	• Grid must have uniform spacing, though the spacing in each dimension does not have to be the same. • Requires at least 4 points in each dimension. • Requires more memory and computation time than `'linear'`.
`'spline'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic spline using not-a-knot end conditions.	C^2	• Requires 4 points in each dimension. • Requires more memory and computation time than `'cubic'`.

ExtrapolationMethod

Extrapolation method, specified as any of the `Method` choices: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, or `'spline'`. In addition, you can specify `'none'` if you want queries outside the domain of your grid return `NaN` values.

If you omit `ExtrapolationMethod`, the default value is the string you specify for `Method`. If you omit both the `Method` and `ExtrapolationMethod` arguments, both default to `'linear'`.

Properties

GridVectors

Cell array containing grid vectors, `{xg1,xg2,...,xgn}`. These vectors specify the grid points (locations) for the values in `F.Values`.

Values

Array of values associated with the grid points in `F.GridVectors`.

Method

A string specifying the name of a method used to interpolate the data. `Method` is one of the strings: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, or `'spline'`. The default value is `'linear'`.

ExtrapolationMethod

A string specifying the name of a method used to extrapolate the data. `ExtrapolationMethod` is one of the strings: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, `'spline'`, or `'none'`. A value of `'none'` indicates that extrapolation is disabled. The default value is the value of `F.Method`.

Definitions

Interpolant

Interpolating function that you can evaluate at query locations.

Gridded Data

A set of points that are axis-aligned and ordered.

Scattered Data

A set of points that have no structure among their relative locations.

Full Grid

A grid represented as a set of arrays. For example, you can create a full grid using `ndgrid`.

Grid Vectors

A set of vectors that serve as a compact representation of a grid in `ndgrid` format. For example, `[X,Y] = ndgrid(xg,yg)` returns a full grid in the matrices `X` and `Y`. You can represent the same grid using the grid vectors, `xg` and `yg`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Indexing

Index-based editing of the properties of F are not supported. Instead, wholly replace the `GridVectors` or `Values` arrays as necessary. See “Interpolation with the griddedInterpolant Class” in the MATLAB Mathematics documentation for more information.

Examples

2-D Interpolation Over Finer Grid

Interpolate coarsely sampled data using a full grid with spacing of `0.5`.

Define the sample points as a full grid with range [1, 10] in both dimensions.

```
[X,Y] = ndgrid(1:10,1:10);
```

Sample $f(x,y) = x^2 + y^2$ at the grid points.

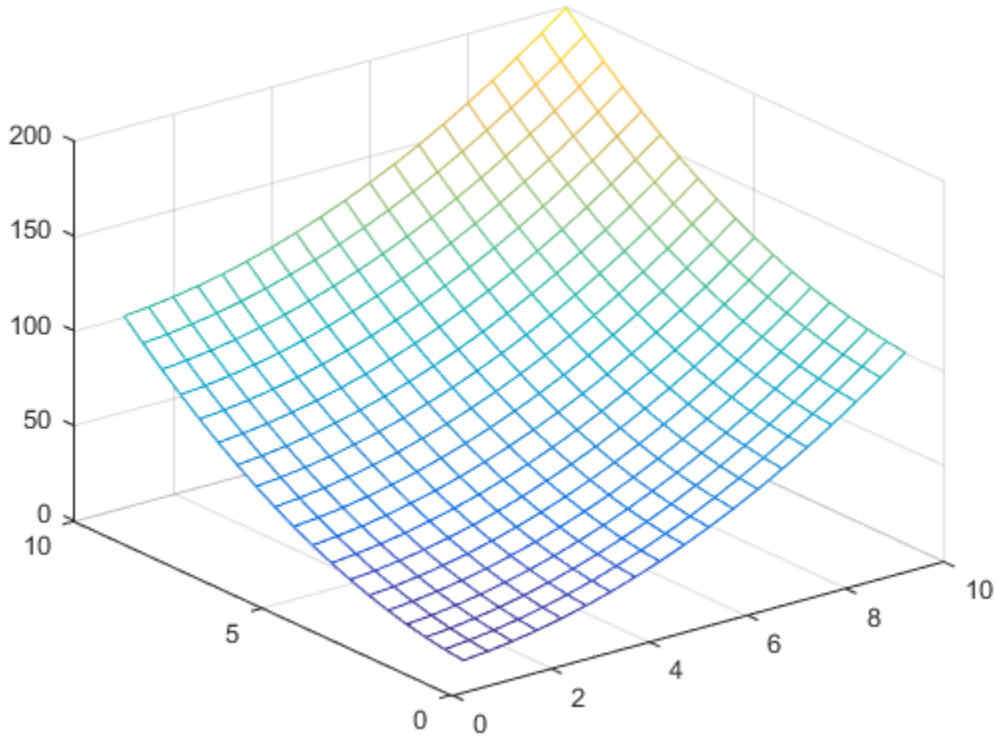
```
V = X.^2 + Y.^2;
```

Create the interpolant, specifying cubic interpolation.

```
F = griddedInterpolant(X,Y,V,'cubic');
```

Define a full grid of query points with `0.5` spacing and evaluate the interpolant at those points. Then plot the result.

```
[Xq,Yq] = ndgrid(1:0.5:10,1:0.5:10);
Vq = F(Xq,Yq);
mesh(Xq,Yq,Vq);
```



1-D Extrapolation

Compare results of querying the interpolant outside the domain of `F` using the `'pchip'` and `'nearest'` extrapolation methods.

Create the interpolant and specify `'pchip'` as the interpolation method.

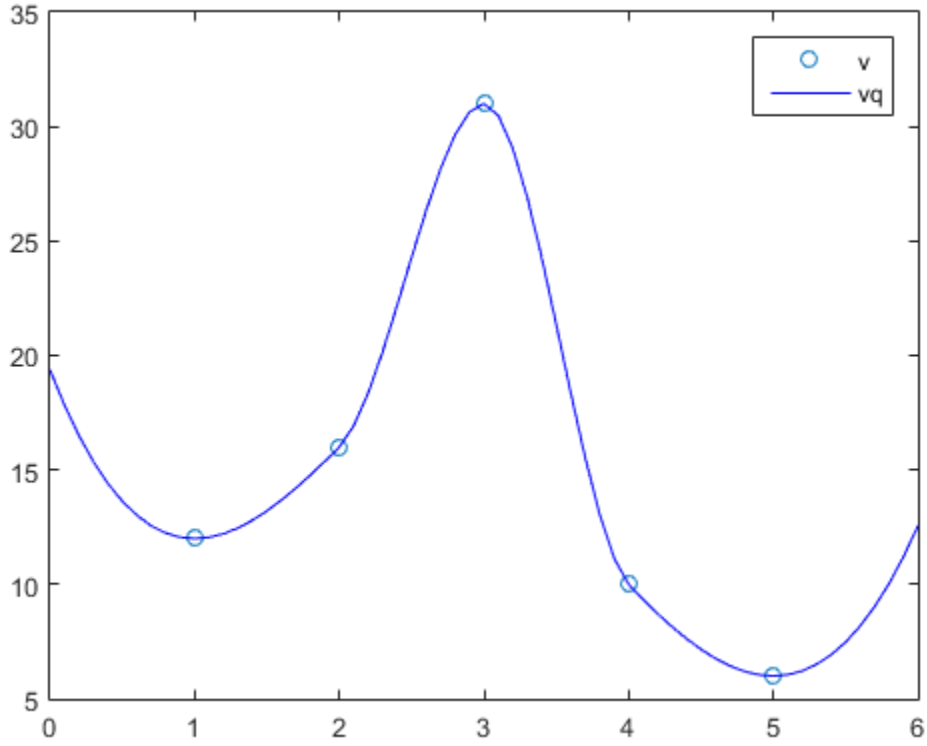
```
x = [1 2 3 4 5];
v = [12 16 31 10 6];
F = griddedInterpolant(x,v,'pchip')
```

```
F =
```

```
  griddedInterpolant with properties:

            GridVectors: {[1 2 3 4 5]}
                 Values: [12 16 31 10 6]
                 Method: 'pchip'
   ExtrapolationMethod: 'pchip'
```

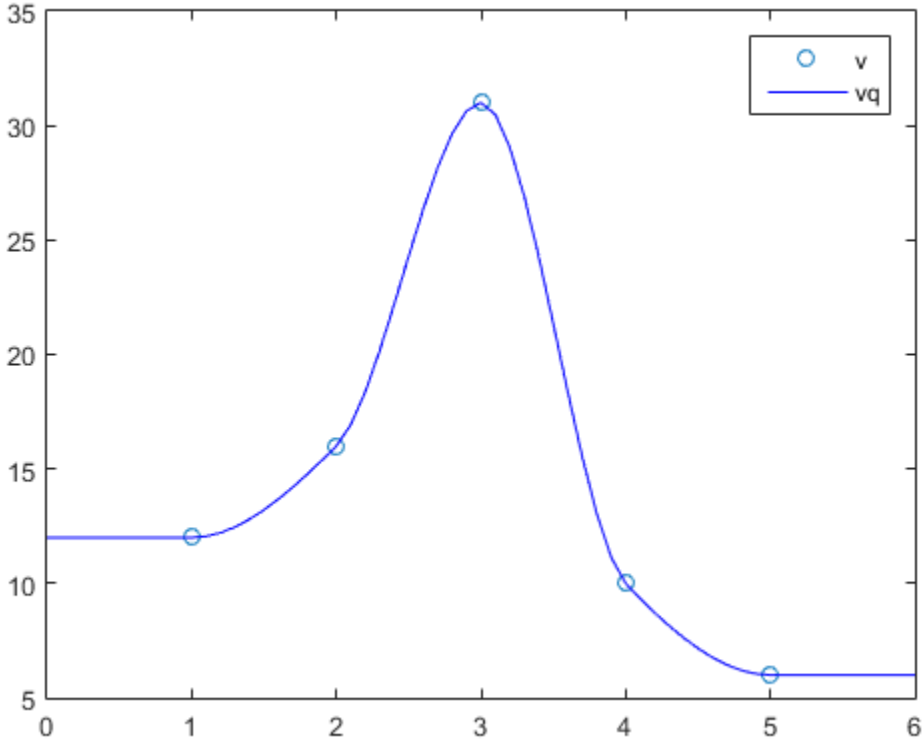
Query the interpolant, and include points outside the domain of `F`.

```
xq = 0:0.1:6;
vq = F(xq);
figure
plot(x,v,'o',xq,vq,'-b');
legend ('v','vq')
```



Query the interpolant at the same points again, using the nearest neighbor extrapolation method.

```
F.ExtrapolationMethod = 'nearest';
figure
vq = F(xq);
plot(x,v,'o',xq,vq,'-b');
legend ('v','vq')
```



See Also

`scatteredInterpolant` | `interp1` | `interp2` | `interp3` | `interpn` | `ndgrid` | `meshgrid`

How To

- Class Attributes
- Property Attributes
- “Interpolating Gridded Data”

groot

Graphics root object

Syntax

```
groot
r = groot
```

Description

`groot` refers to the graphics root object. Use `groot` to access root properties. For a list of properties, see Root Properties.

`r = groot` stores the graphics root object handle. To set root properties using dot notation, you must store the handle first.

Examples

View Root Property Values

View a list of graphics root properties and their current values.

```
get(groot)
```

```
         CallbackObject: []
               Children: []
          CurrentFigure: []
     FixedWidthFontName: 'Courier New'
       HandleVisibility: 'on'
       MonitorPositions: [2x4 double]
                 Parent: []
        PointerLocation: [2957 348]
            ScreenDepth: 32
    ScreenPixelsPerInch: 96
             ScreenSize: [1 1 1920 1200]
      ShowHiddenHandles: 'off'
```

```
               Tag: ''
              Type: 'root'
             Units: 'pixels'
          UserData: []
```

Set Root Property Values

Set graphics root property values by storing the object handle and using dot notation.

```
r = groot;
r.ShowHiddenHandles = 'on';
```

More About

Tips

- Use the graphics root object to set default values on the root level for other types of objects. For example, set the default colormap for all future figures to the `summer` colormap.

  ```
  set(groot,'DefaultFigureColormap',summer)
  ```
  To restore a property to its original MATLAB default, use the `'remove'` keyword.

  ```
  set(groot,'DefaultFigureColormap','remove')
  ```
  For more information on setting default values, see “Default Property Values”.

See Also

Functions
`gca` | `gcf`

Properties
Root Properties

gsvd

Generalized singular value decomposition

Syntax

```
[U,V,X,C,S] = gsvd(A,B)
sigma = gsvd(A,B)
```

Description

`[U,V,X,C,S] = gsvd(A,B)` returns unitary matrices `U` and `V`, a (usually) square matrix `X`, and nonnegative diagonal matrices `C` and `S` so that

```
A = U*C*X'
B = V*S*X'
C'*C + S'*S = I
```

`A` and `B` must have the same number of columns, but may have different numbers of rows. If `A` is `m`-by-`p` and `B` is `n`-by-`p`, then `U` is `m`-by-`m`, `V` is `n`-by-`n` and `X` is `p`-by-`q` where `q = min(m+n,p)`.

`sigma = gsvd(A,B)` returns the vector of generalized singular values, `sqrt(diag(C'*C)./diag(S'*S))`.

The nonzero elements of `S` are always on its main diagonal. If `m >= p` the nonzero elements of `C` are also on its main diagonal. But if `m < p`, the nonzero diagonal of `C` is `diag(C,p-m)`. This allows the diagonal elements to be ordered so that the generalized singular values are nondecreasing.

`gsvd(A,B,0)`, with three input arguments and either `m` or `n >= p`, produces the "economy-sized"decomposition where the resulting `U` and `V` have at most `p` columns, and `C` and `S` have at most `p` rows. The generalized singular values are `diag(C)./diag(S)`.

When `B` is square and nonsingular, the generalized singular values, `gsvd(A,B)`, are equal to the ordinary singular values, `svd(A/B)`, but they are sorted in the opposite order. Their reciprocals are `gsvd(B,A)`.

In this formulation of the gsvd, no assumptions are made about the individual ranks of A or B. The matrix X has full rank if and only if the matrix [A;B] has full rank. In fact, svd(X) and cond(X) are equal to svd([A;B]) and cond([A;B]). Other formulations, eg. G. Golub and C. Van Loan [1], require that null(A) and null(B) do not overlap and replace X by inv(X) or inv(X').

Note, however, that when null(A) and null(B) do overlap, the nonzero elements of C and S are not uniquely determined.

Examples

Example 1

The matrices have at least as many rows as columns.

```
A = reshape(1:15,5,3)
B = magic(3)
A =
         1     6    11
         2     7    12
         3     8    13
         4     9    14
         5    10    15
B =
         8     1     6
         3     5     7
         4     9     2
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 5-by-5 orthogonal U, a 3-by-3 orthogonal V, a 3-by-3 nonsingular X,

```
X =
        2.8284   -9.3761   -6.9346
       -5.6569   -8.3071  -18.3301
        2.8284   -7.2381  -29.7256
```

and

```
C =
        0.0000         0         0
```

```
             0    0.3155         0
             0         0    0.9807
             0         0         0
             0         0         0
S =
        1.0000         0         0
             0    0.9489         0
             0         0    0.1957
```

Since A is rank deficient, the first diagonal element of C is zero.

The economy sized decomposition,

```
[U,V,X,C,S] = gsvd(A,B,0)
```

produces a 5-by-3 matrix U and a 3-by-3 matrix C.

```
U =
        0.5700   -0.6457   -0.4279
       -0.7455   -0.3296   -0.4375
       -0.1702   -0.0135   -0.4470
        0.2966    0.3026   -0.4566
        0.0490    0.6187   -0.4661

C =
        0.0000         0         0
             0    0.3155         0
             0         0    0.9807
```

The other three matrices, V, X, and S are the same as those obtained with the full decomposition.

The generalized singular values are the ratios of the diagonal elements of C and S.

```
sigma = gsvd(A,B)
sigma =
        0.0000
        0.3325
        5.0123
```

These values are a reordering of the ordinary singular values

```
svd(A/B)
ans =
        5.0123
        0.3325
```

```
         0.0000
```

Example 2

The matrices have at least as many columns as rows.

```
A = reshape(1:15,3,5)
B = magic(5)
A =

     1     4     7    10    13
     2     5     8    11    14
     3     6     9    12    15

B =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 3-by-3 orthogonal `U`, a 5-by-5 orthogonal `V`, a 5-by-5 nonsingular `X` and

```
C =

         0          0    0.0000          0          0
         0          0          0    0.0439          0
         0          0          0          0    0.7432

S =

    1.0000          0          0          0          0
         0    1.0000          0          0          0
         0          0    1.0000          0          0
         0          0          0    0.9990          0
         0          0          0          0    0.6690
```

In this situation, the nonzero diagonal of `C` is `diag(C,2)`. The generalized singular values include three zeros.

```
sigma = gsvd(A,B)
sigma =

         0
         0
    0.0000
    0.0439
    1.1109
```

Reversing the roles of `A` and `B` reciprocates these values, producing two infinities.

```
gsvd(B,A)
ans =

   1.0e+16 *

    0.0000
    0.0000
    8.8252
       Inf
       Inf
```

Diagnostics

The only warning or error message produced by `gsvd` itself occurs when the two input arguments do not have the same number of columns.

More About

Algorithms

The generalized singular value decomposition uses the C-S decomposition described in [1], as well as the built-in `svd` and `qr` functions. The C-S decomposition is implemented in a local function in the `gsvd` program file.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

`qr` | `svd`

Introduced before R2006a

gt, >

Determine greater than

Syntax

```
A > B
gt(A,B)
```

Description

`A > B` returns a logical array with elements set to logical 1 (`true`) where `A` is greater than `B`; otherwise, it returns logical `0` (`false`).

The test compares only the real part of numeric arrays. `gt` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`gt(A,B)` is an alternate way to execute `A > B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Determine if vector elements are greater than a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are greater than `10`.

```
A > 10
```

```
ans =

     0     1     1     0     0     1     0     1
```

The result is a vector with values of logical 1 (`true`) where the elements of A satisfy the expression.

Use the vector of logical values as an index to view the values in A that are greater than 10.

```
A(A > 10)
```

```
ans =

    12    18    11    15
```

The result is a subset of the elements in A.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values greater than 9 with the value 10.

```
A(A > 9) = 10
```

```
A =

    10     2     3    10
     5    10    10     8
     9     7     6    10
     4    10    10     1
```

The result is a new matrix whose largest element is 10.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
```

```
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)

A =

     large       medium      small
     medium      small       large
```

The array has three categories: `'small'`, `'medium'`, and `'large'`.

Find all values greater than the category `'medium'`.

```
A > 'medium'

ans =

     1     0     0
     0     0     1
```

A value of logical `1` (`true`) indicates a value greater than the category `'medium'`.

Compare the rows of `A`.

```
A(1,:) > A(2,:)

ans =

     1     1     0
```

The function returns logical `1` (`true`) where the first row has a category value greater than the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are greater than `2`.

```
A(A > 2)

ans =

   5.0000 - 1.0000i
```

`gt` compares only the real part of the elements in `A`.

Use `abs` to find which elements are outside a radius of `2` from the origin.

```
A(abs(A) > 2)
```

```
ans =

   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i   5.0000 - 1.0000i
```

The result has more elements since `abs` accounts for the imaginary part of the numbers.

Compare Dates

Create a vector of dates.

```
A = datetime([2014,05,01;2014,05,31])
```

```
A =

   01-May-2014
   31-May-2014
```

Find the dates that occur after May 10, 2014.

```
A(A > '2014-05-10')
```

```
ans =

   31-May-2014
```

Input Arguments

A — Left array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Left array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical

arrays, they must have the same sets of categories, including their order. See “Compare Categorical Array Elements” for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

`B` — Right array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Right array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See “Compare Categorical Array Elements” for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

More About

- “Ordinal Categorical Arrays”

See Also

`eq` | `ge` | `le` | `lt` | `ne`

Introduced before R2006a

gtext

Mouse placement of text in 2-D view

Syntax

```
gtext('string')
gtext({'string1','string2','string3',...})
gtext({'string1';'string2';'string3';...})
gtext(...,'PropertyName',PropertyValue,...)
h = gtext(...)
```

Description

`gtext` displays a text string in the current figure window after you select a location with the mouse.

`gtext('string')` waits for you to press a mouse button or keyboard key while the pointer is within a figure window. Pressing a mouse button or any key places `'string'` on the plot at the selected location.

`gtext({'string1','string2','string3',...})` places all strings with one click, each on a separate line.

`gtext({'string1';'string2';'string3';...})` places one string per click, in the sequence specified.

`gtext(...,'PropertyName',PropertyValue,...)` sets the values of the specified text properties. For a list of properties, see Text Properties.

`h = gtext(...)` returns the handle to a text graphics object that is placed on the plot at the location you select.

Examples

Place a label on the current plot:

```
gtext('Note this divergence!')
```

More About

Tips

As you move the pointer into a figure window, the pointer becomes crosshairs to indicate that `gtext` is waiting for you to select a location. `gtext` uses the functions `ginput` and `text`.

See Also

`ginput` | `text`

Introduced before R2006a

guidata

Store or retrieve UI data

Syntax

```
guidata(object_handle,data)
data = guidata(object_handle)
```

Description

`guidata(object_handle,data)` stores the variable `data` with the object specified by `object_handle`. If `object_handle` is not a figure, then the object's parent figure is used. `data` can be any MATLAB variable, but is typically a structure, which enables you to add new fields as required.

`guidata` can manage only one variable at any time. Subsequent calls to `guidata(object_handle,data)` overwrite the previously stored `data`.

GUIDE Uses guidata GUIDE uses `guidata` to store and maintain the `handles` structure. In a GUIDE code file, do not overwrite the `handles` structure or your program will no longer work. If you need to store other data, you can do so by adding new fields to the `handles` structure.

`data = guidata(object_handle)` returns previously stored data, or an empty matrix if nothing is stored.

To change the data managed by `guidata`:

1. Get a copy of the data with the command `data = guidata(object_handle)`.
2. Make the desired changes to `data`.
3. Save the changed version of `data` with the command `guidata(object_handle,data)`.

`guidata` provides application developers with a convenient interface to a figure's application data:

- You do not need to create and maintain a hard-coded property name for the application data throughout your source code.
- You can access the data from within a local function callback routine using the component's handle (which is returned by `gcbo`), without needing to find the figure's handle.

If you are not using GUIDE, `guidata` is particularly useful in conjunction with `guihandles`, which creates a structure containing the all of the components in a UI.

Examples

This example calls `guidata` to save a structure containing a figure's application data from within the initialization section of the application code file. The first section shows how to do this within a programmatic UI. The second section shows how the code differs when you use GUIDE to create a template code file. GUIDE provides a `handles` structure as an argument to all local function callbacks, so you do not need to call `guidata` to obtain it. You do, however, need to call `guidata` to save changes you make to the structure.

Using guidata in a Programmatic UI

Calling the `guihandles` function creates the structure into which your code places additional data. It contains all handles used by the figure at the time it is called, generating field names based on each object's `Tag` property.

```
% Create figure
figure_handle = figure('Toolbar','none');
% create structure of handles
myhandles = guihandles(figure_handle);
% Add some additional data as a new field called numberOfErrors
myhandles.numberOfErrors = 0;
% Save the structure
guidata(figure_handle,myhandles)
```

You can recall the data from within a local callback function, modify it, and then replace the structure in the figure:

```
function My_Callback()
% ...
% Get the structure using guidata in the local function
myhandles = guidata(gcbo);
% Modify the value of your counter
myhandles.numberOfErrors = myhandles.numberOfErrors + 1;
```

```
% Save the change you made to the structure
guidata(gcbo,myhandles)
```

Using guidata in GUIDE

If you use GUIDE, you do not need to call `guihandles` to create a structure. You can add your own data to the handles `structure`. For example, this code adds the field, `numberOfErrors`, to the structure:

```
% --- Executes just before simple_gui_tab is made visible.
function my_GUIDE_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to simple_gui_tab (see VARARGIN)
% ...

% add some additional data as a new field called numberOfErrors
handles.numberOfErrors = 0;
% Save the change you made to the structure
guidata(hObject,handles)
```

Notice that you use the input argument `hObject` in place of `gcbo` to refer to the object whose callback is executing.

Suppose you needed to access the `numberOfErrors` field in a push button callback. Your callback code now looks something like this:

```
% --- Executes on button press in pushbutton1.
function my_GUIDE_GUI_pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% ...

% No need to call guidata to obtain a structure;
% it is provided by GUIDE via the handles argument
handles.numberOfErrors = handles.numberOfErrors + 1;
% save the changes to the structure
guidata(hObject,handles)
```

See Also

`guide` | `guihandles` | `getappdata` | `setappdata`

Introduced before R2006a

guide

Open GUIDE

Syntax

```
guide
guide('filename.fig')
guide('fullpath')
guide(HandleList)
```

Description

`guide` launches GUIDE, a UI design environment.

`guide` opens the GUIDE Quick Start dialog where you can choose to open a previously created or create a new one using one of the provided templates.

`guide('filename.fig')` opens the FIG-file named `filename.fig` for editing if it is on the MATLAB path.

`guide('fullpath')` opens the FIG-file at `fullpath` even if it is not on the MATLAB path.

`guide(HandleList)` opens the content of each of the figures in `HandleList` in a separate copy of the GUIDE design environment.

More About

- "Ways to Build MATLAB UIs"
- "Create a Simple UI Using GUIDE"

See Also

`inspect`

Introduced before R2006a

guihandles

Create structure of handles

Syntax

```
handles = guihandles(object_handle)
handles = guihandles
```

Description

`handles = guihandles(object_handle)` returns a structure containing the handles of the objects in a figure, using the value of their `Tag`properties as the field names, with the following caveats:

- Objects are excluded if their Tag properties are empty, or are not legal variable names.
- If several objects have the same Tag, that field in the structure contains a vector of handles.
- Objects with hidden handles are included in the structure.

`handles = guihandles` returns a structure of handles for the current figure.

See Also

`guidata` | `guide` | `getappdata` | `setappdata`

Introduced before R2006a

gunzip

Uncompress GNU zip files

Syntax

```
gunzip(files)
gunzip(files,outputdir)
gunzip(url, ...)
filenames = gunzip(...)
```

Description

`gunzip(files)` uncompresses GNU zip files from the list of files specified in `files`. Directories recursively `gunzip` all of their content. The output files have the same name, excluding the extension `.gz`, and are written to the same directory as the input files.

`files` is a string or cell array of strings containing a list of files or directories. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~username/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`gunzip(files,outputdir)` writes the gunzipped file into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`gunzip(url, ...)` extracts the GNU zip contents from an Internet universal resource locator (URL). The URL must include the protocol type (for example, `'http://'`). MATLAB downloads the URL to the temp directory, and then deletes it.

`filenames = gunzip(...)` gunzips the files and returns the relative path names of the gunzipped files in the string cell array `filenames`.

Examples

To `gunzip` all `.gz` files in the current directory, type:

```
gunzip('*.gz');
```

To `gunzip` Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`, type:

```
url ='http://www.mathworks.com/moler/ncm.tar.gz';
gunzip(url,'ncm')
untar('ncm/ncm.tar','ncm')
```

See Also

`gzip` | `tar` | `untar` | `unzip` | `zip`

Introduced before R2006a

gzip

Compress files into GNU zip files

Syntax

```
gzip(files)
gzip(files,outputdir)
filenames = gzip(...)
```

Description

`gzip(files)` creates GNU zip files from the list of files specified in `files`. Directories recursively `gzip` all their contents. Each output gzipped file is written to the same directory as the input file and with the file extension `.gz`.

`files` is a string or cell array of strings containing a list of files or directories to `gzip`. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~username/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`gzip(files,outputdir)` writes the gzipped files into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`filenames = gzip(...)` gzips the files and returns the relative path names of all gzipped files in the string cell array `filenames`.

Examples

To `gzip` all `.m` and `.mat` files in the current directory and store the results in the directory `archive`, type:

```
gzip({'*.m','*.mat'},'archive');
```

See Also

gunzip | tar | untar | unzip | zip

Introduced before R2006a

h5create

Create HDF5 data set

Syntax

```
h5create(filename,datasetname,size,Name,Value)
```

Description

`h5create(filename,datasetname,size,Name,Value)` creates an HDF5 data set in the file specified by `filename`.

Input Arguments

`filename`

Text string specifying the name of an HDF5 file. If `filename` does not already exist, `h5create` creates it, with additional options specified by one or more Name,Value pair arguments.

Default:

`datasetname`

Text string specifying the name of the data set you want to create. If `datasetname` is a full path name, `h5create` creates all intermediate groups, if they don't already exist.

Default:

`size`

Array specifying the extents of the dataset. To specify an unlimited extent, set the corresponding element of `size` to `Inf`.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Datatype'`

Any of the following MATLAB datatypes.

`double`	`uint64`	`uint32`	`uint16`	`uint8`
`single`	`int64`	`int32`	`int16`	`int8`

Default: `double`

`'ChunkSize'`

Defines chunking layout.

Default: Not chunked

`'Deflate'`

Defines gzip compression level (0-9).

Default: 0

`'FillValue'`

Defines the fill value for numeric data sets.

Default:

`'Fletcher32'`

Turns on the Fletcher32 checksum filter.

Default: `false`

`'Shuffle'`

Turns on the Shuffle filter.

Default: `false`

Examples

Create a fixed-size 100-by-200 data set.

```
h5create('myfile.h5','/myDataset1',[100 200]);
h5disp('myfile.h5');
```

Create a single-precision 1000-by-2000 data set with a chunk size of 50-by-80. Apply the highest level of compression.

```
h5create('myfile.h5','/myDataset2',[1000 2000], 'Datatype','single', ...
         'ChunkSize',[50 80],'Deflate',9);
h5disp('myfile.h5');
```

Create a two-dimensional data set that is unlimited along the second extent.

```
h5create('myfile.h5','/myDataset3',[200 Inf],'ChunkSize',[20 20]);
h5disp('myfile.h5');
```

See Also

`h5read` | `h5write` | `h5info` | `h5disp`

Introduced in R2011a

h5disp

Display contents of HDF5 file

Syntax

```
h5disp(filename)
h5disp(filename,location)
h5disp(filename,location,mode)
```

Description

`h5disp(filename)` displays the structure (metadata) of the entire HDF5 file, `filename`.

`h5disp(filename,location)` displays the metadata for the specified location.

`h5disp(filename,location,mode)` displays the file metadata according to the value of `mode`.

Input Arguments

`filename`

Text string specifying the name of an HDF5 file.

Default:

`location`

Text string specifying the full path to a location in an HDF5 file. To display the metadata for the entire file, specify `'/'` as the value of `location`. If `location` is a group, `h5disp` displays all objects below the group.

Default:

`mode`

Either of the following text strings.

Value	Description
`min`	Minimal, display only group and data set names.
`simple`	Display data set metadata and attribute values, if the attribute is an integer, floating point, or a scalar string.

Default: `simple`

Examples

Display the entire contents of an HDF5 file.

```
h5disp('example.h5')
```

Display metadata for one data set in an HDF5 file.

```
 h5disp('example.h5','/g4/world');
```

See Also

`h5info`

Introduced in R2011a

h5info

Return information about HDF5 file

Syntax

```
info = h5info(filename)
info = h5info(filename,location)
```

Description

`info = h5info(filename)` returns information about the entire HDF5 file, specified by `filename`.

`info = h5info(filename,location)` returns information about the group, data set, or named datatype specified by `location` in the HDF5 file, `filename`.

Input Arguments

`filename`

Text string specifying the name of an HDF5 file.

Default:

`location`

Text string specifying the location of a group, data set, or named datatype in an HDF5 file.

Default:

Output Arguments

`info`

A structure containing information about the HDF5 file. The set of fields in the structure depends on the `location` parameter. The first field is always `'Filename'`. Other fields that might be present in the `info` structure are as follows.

Location Type	Field	Description
Files and Groups		
	`Name`	Text string specifying name of the group. If you specify only a file name, this value is `'/'`.
	`Groups`	Array of structures describing subgroups.
	`Datasets`	Array of structures describing data sets.
	`Datatypes`	Array of structures describing named datatypes.
	`Links`	Array of structures describing soft, external, user-defined, and certain hard links.
	`Attributes`	Array of structures describing group attributes.
Data sets		
	`Name`	Text string specifying the name of the data set.
	`Datatype`	Structure describing the datatype.
	`Dataspace`	Structure describing the size of the dataset.
	`ChunkSize`	Extents of the data set's chunk size, if defined.
	`FillValue`	Data set's fill value, if defined.
	`Filter`	Array of structures describing any defined filters such as compression.

Location Type	Field	Description
	`Attributes`	Array of structures describing data set attributes.
Named Datatypes		
	`Name`	Text string specifying the name of the datatype object.
	`Class`	HDF5 class of the named datatype.
	`Type`	Text string or struct further describing the datatype.
	`Size`	Size of the named datatype in bytes.

Examples

Return all information.

```
info = h5info('example.h5');
```

Return information about a group and all data sets contained within the group.

```
info = h5info('example.h5','/g4');
```

Return information about a specific dataset.

```
info = h5info('example.h5','/g4/time');
```

See Also

`h5disp`

Introduced in R2011a

h5read

Read data from HDF5 data set

Syntax

```
data = h5read(filename,datasetname)
data = h5read(filename,datasetname,start,count)
data = h5read(filename,datasetname,start,count,stride)
```

Description

`data = h5read(filename,datasetname)` retrieves all of the data from the HDF5 data set `datasetname` in the file `filename`.

`data = h5read(filename,datasetname,start,count)` reads a subset of data from the data set `datasetname`. `start` is the one-based index of the first element to be read. `count` defines how many elements to read along each dimension. If a particular element of `count` is `Inf`, `h5read` reads data until the end of the corresponding dimension.

`data = h5read(filename,datasetname,start,count,stride)` reads a subset of data, where `stride` specifies the interelement spacing along each data set extent.

Input Arguments

`filename`

Text string specifying the name of an HDF5 file.

Default:

`datasetname`

Text string specifying the name of a data set in an HDF5 file.

Default:

`start`

Numeric index value specifying the place to start reading data in the dataset in an HDF5 file. Indices are 1-based.

Default:

`count`

Numeric value specifying the amount of data to read.

Default:

`stride`

Numeric value specifying the intervalue spacing during the read operation. For example, a spacing of 2 indicates reading every other value.

Default:

Output Arguments

`data`

Data read from the data set.

Examples

Read an entire data set.

```
h5disp('example.h5','/g4/lat');
data = h5read('example.h5','/g4/lat');
```

Read the first 5-by-3 subset of a data set.

```
h5disp('example.h5','/g4/world');
data = h5read('example.h5','/g4/world',[1 1],[5 3]);
```

Read a data set of references to other data sets.

```
h5disp('example.h5','/g3/reference');
```

```
data = h5read('example.h5','/g3/reference');
```

See Also

h5write | h5readatt | h5disp | h5writeatt

Introduced in R2011a

h5readatt

Read attribute from HDF5 file

Syntax

```
attval = h5readatt(filename,location,attr)
```

Description

`attval = h5readatt(filename,location,attr)` retrieves the value for the named attribute `attr` from the given `location` in the HDF5 file `filename`.

Input Arguments

`filename`

Text string specifying the name of an HDF5 file.

Default:

`location`

Text string specifying the full path of the attribute in an HDF5 file. `location` can refer to either a group or a data set.

Default:

`attr`

Text string specifying the name of an attribute in an HDF5 file.

Default:

Output Arguments

`attval`

Value of the attribute.

Examples

Read a group attribute.

```
attval = h5readatt('example.h5','/','attr2');
```

Read a data set attribute.

```
attval = h5readatt('example.h5','/g4/lon','units');
```

See Also

h5writeatt | h5info

Introduced in R2011a

h5write

Write to HDF5 data set

Syntax

```
h5write(filename,datasetname,data)
h5write(filename,datasetname,data,start,count)
h5write(filename,datasetname,data,start,count,stride)
```

Description

`h5write(filename,datasetname,data)` writes `data` to an entire data set, `datasetname`, in the HDF5 file, `filename`.

`h5write(filename,datasetname,data,start,count)` writes a subset of the `data` to a data set, `datasetname`, in the HDF5 file, `filename`. `start` is a one-based index value that specifies the first element to be written. `count` specifies the number of elements to write along each dimension. `h5write` extends an extendable data set along any unlimited dimensions, if necessary.

`h5write(filename,datasetname,data,start,count,stride)` writes a hyperslab of data, where `stride` specifies the inter-element spacing along each dimension.

Input Arguments

`filename`

Text string specifying the name of an HDF5 file.

Default:

`datasetname`

Text string specifying the name of a data set in the HDF5 file.

Default:

`data`

Data to be written to the HDF5 file. You can specify only floating-point and integer data sets.

Default:

`start`

Numeric index value specifying where in the data set to start writing to the file.

Default:

`count`

Numeric value specifying how much data to write to the file.

Default:

`stride`

Numeric value specifying the interelement spacing of data to write to the file.

Default: Vector of ones.

Examples

Write to an entire data set.

```
h5create('myfile.h5','/DS1',[10 20]);
mydata = rand(10,20);
h5write('myfile.h5', '/DS1', mydata);
```

Write a hyperslab of data to the last 5-by-7 block of a data set.

```
 h5create('myfile.h5','/DS2',[10 20]);
        mydata = rand(5,7);
        h5write('myfile.h5','/DS2',mydata,[6 14],[5 7]);
```

Append data to an unlimited data set.

```
h5create('myfile.h5','/DS3',[20 Inf],'ChunkSize',[5 5]);
for j = 1:10
```

```
      data = j*ones(20,1);
      start = [1 j];
      count = [20 1];
      h5write('myfile.h5','/DS3',data,start,count);
end
h5disp('myfile.h5');
```

Limitations

- `h5write` supports only floating point and integer data sets. To write to string data sets, you must use the H5D package.

More About

Hyperslab

A hyperslab is a collection of points in a data space. The points can be contiguous or form a regular pattern of points or blocks in a data space.

See Also

`h5read` | `h5create` | `h5writeatt` | `h5disp` | `H5D.create` | `H5D.write`

Introduced in R2011a

h5writeatt

Write HDF5 attribute

Syntax

```
h5writeatt(filename,location,attname,attvalue)
```

Description

`h5writeatt(filename,location,attname,attvalue)` writes the attribute named `attname` with the value `attvalue` to the HDF5 file `filename`. The parent object `location` can be either a group or variable. `location` is the complete path name of the group or variable to which you want to associate the attribute.

Input Arguments

`filename`

Text string specifying the name of an HDF5 file.

Default:

`location`

Text string specifying the full path identifying a group or variable in an HDF5 file.

Default:

`attname`

Text string specifying the name of an attribute in an HDF5 file. If the attribute does not exist, `h5writeatt` creates the attribute with the name specified.

If the specified attribute already exists but does not have a datatype or dataspace consistent with `attvalue`, `h5writeatt` deletes the attribute and recreates it. String attributes are created with a scalar dataspace.

Default:

`attvalue`

Value to be written to the attribute in an HDF5 file.

Default:

Examples

Create a root group attribute whose value is the current time.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
h5writeatt('myfile.h5','/','creation_date',datestr(now));
```

Create a double-precision data set attribute.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
attData = [0 1 2 3];
h5writeatt('myfile.h5','/g4/world','attr',attData);
h5disp('myfile.h5','/g4/world');
```

See Also

`h5readatt` | `h5disp` | `h5write`

Introduced in R2011a

H5.close

Close HDF5 library

Syntax

```
H5.close()
```

Description

`H5.close()` closes the HDF5 library.

See Also

`H5.open`

H5.garbage_collect

Free unused memory in HDF5 library

Syntax

```
H5.garbage_collect()
```

Description

`H5.garbage_collect()` frees unused memory in the HDF5 library.

H5.get_libversion

Version of HDF5 library

Syntax

```
[majnum,minnum,relnum] = H5.get_libversion()
```

Description

`[majnum,minnum,relnum] = H5.get_libversion()` returns the version of the HDF5 library in use.

H5.open

Open HDF5 library

Syntax

```
H5.open()
```

Description

`H5.open()` opens the HDF5 library.

See Also

`H5.close`

H5.set_free_list_limits

Set size limits on free lists

Syntax

```
H5.set_free_list_limits(reg_global_lim,reg_list_lim,arr_global_lim,arr_list_li
```

Description

`H5.set_free_list_limits(reg_global_lim,reg_list_lim,arr_global_lim,arr_list_li`
sets size limits on all types of free lists.

H5A.close

Close specified attribute

Syntax

```
H5A.close(attr_id)
```

Description

`H5A.close(attr_id)` terminates access to the attribute specified by `attr_id`, releasing the identifier.

See Also

`H5A.open`

H5A.create

Create attribute

Syntax

```
attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id)
attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id,aapl_id)
```

Description

`attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id)` creates the attribute name that is attached to the object specified by `loc_id`. `loc_id` is a group, dataset, or named datatype identifier. The datatype and dataspace identifiers of the attribute, `type_id` and `space_id`, respectively, are created with the H5T and H5S interfaces. The attribute property list, `acpl_id`, is currently unused and should be set to `'H5P_DEFAULT'`. This interface corresponds to the 1.6.x version of `H5Acreate`.

`attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id,aapl_id)` creates the attribute with the additional attribute access property list identifier `aapl_id`. `aapl_id` should currently be set to `'H5P_DEFAULT'`. This interface corresponds to the 1.8.x version of `H5Acreate`.

Examples

```
acpl_id = H5P.create('H5P_ATTRIBUTE_CREATE');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
space_id = H5S.create('H5S_SCALAR');
fid = H5F.create('myfile.h5');
attr_id = H5A.create(fid,'my_attr',type_id,space_id,acpl_id);
H5A.close(attr_id);
H5F.close(fid);
```

See Also

`H5A.close` | `H5P.create`

H5A.delete

Delete attribute

Syntax

```
H5A.delete(loc_id,name)
```

Description

`H5A.delete(loc_id,name)` removes the attribute specified by `name` from the dataset, group, or named datatype specified by `loc_id`.

Examples

Delete a root group attribute.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
gid = H5G.open(fid,'/');
H5A.delete(gid,'attr1');
H5G.close(gid);
H5F.close(fid);
```

H5A.get_info

Information about attribute

Syntax

```
info = H5A.get_info(attr_id)
```

Description

`info = H5A.get_info(attr_id)` returns information about an attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
info = H5A.get_info(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open`

H5A.get_name

Attribute name

Syntax

```
attr_name = H5A.get_name(attr_id)
```

Description

`attr_name = H5A.get_name(attr_id)` returns the name of the attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_INC';
attr_id = H5A.open_by_idx(gid,'dset1.1.1',idx_type,order,0);
name = H5A.get_name(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open_by_idx`

H5A.get_space

Copy of attribute data space

Syntax

```
dspace_id = H5A.get_space(attr_id)
```

Description

`dspace_id = H5A.get_space(attr_id)` returns a copy of the data space for the attribute specified by `attr_id`.

Examples

Retrieve the dimensions of an attribute data space.

```
fid = H5F.open('example.h5');
attr_id = H5A.open(fid,'attr2');
space = H5A.get_space(attr_id);
[~,dims] = H5S.get_simple_extent_dims(space);
H5A.close(attr_id);
H5F.close(fid);
```

See Also

`H5A.open` | `H5S.close`

H5A.get_type

Copy of attribute data type

Syntax

```
type_id = H5A.get_type(attr_id)
```

Description

`type_id = H5A.get_type(attr_id)` returns a copy of the data type for the attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
type_id = H5A.get_type(attr_id);
H5T.close(type_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open` | `H5T.close`

H5A.iterate

Execute function for attributes attached to object

Syntax

```
[status,idx_stop,cdata_out] =
H5A.iterate(obj_id,idx_type,order,idx_start,iter_func,cdata_in)
H5A.iterate(loc_id,attr_idx,iterator_func)
```

Description

`[status,idx_stop,cdata_out] = H5A.iterate(obj_id,idx_type,order,idx_start,iter_func,cdata_in)` executes the specified function `iter_func` for each attribute connected to an object. `obj_id` identifies the object to which attributes are attached. `idx_type` is the type of index and valid values include the following.

`'H5_INDEX_NAME'`	An alpha-numeric index by attribute name
`'H5_INDEX_CRT_ORDER'`	An index by creation order

`order` specifies the index traversal order. Valid values include the following.

`'H5_ITER_INC'`	Iteration from beginning to end
`'H5_ITER_DEC'`	Iteration from end to beginning
`'H5_ITER_NATIVE'`	Iteration in the fastest available order

`idx_start` specifies the starting point of the iteration. `idx_stop` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function, `iter_func`, must have the following signature:

```
[status,cdata_out] = iter_func(obj_id,attr_name,info,cdata_in)
```

`cdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `cdata_in` parameter. The `cdata_out` of an iteration step

forms the `cdata_in` for the next iteration step. Then, the final `cdata_out` at the end of the iteration is returned to the caller as `cdata_out`. This form of `H5A.iterate` corresponds to the `H5Aiterate2` function in the HDF5 C API.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

`H5A.iterate(loc_id,attr_idx,iterator_func)` executes the specified function for each attribute of the group, dataset, or named datatype specified by `loc_id`. The `attr_idx` argument specifies where the iteration begins. `iterator_func` must be a function handle.

The iterator function must have the following signature:

```
status = iterator_func(loc_id,attr_name)
```

`loc_id` still specifies the group, dataset, or named data type passed into `H5A.iterate`, and `attr_name` specifies the current attribute. This form of `H5A.iterate` corresponds to `H5Aiterate1` function in the HDF5 C API.

H5A.open

Open attribute

Syntax

```
attr_id = H5A.open(obj_id,attr_name)
attr_id = H5A.open(obj_id,attr_name,aapl_id)
```

Description

`attr_id = H5A.open(obj_id,attr_name)` opens an attribute for an object specified by a parent object identifier and attribute name.

`attr_id = H5A.open(obj_id,attr_name,aapl_id)` opens an attribute with an attribute access property list identifier, `aapl_id`. The only currently valid value for `aapl_id` is `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.close` | `H5A.open_by_idx` | `H5A.open_by_name`

H5A.open_by_idx

Open attribute specified by index

Syntax

```
attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n)
attr_id =
H5A.open_by_idx(loc_id,obj_name,idx_type,order,n,aapl_id,lapl_id)
```

Description

`attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n)` opens an existing attribute at index `n` attached to an object specified by its location, `loc_id`, and name, `obj_name`.

`idx_type` is the type of index and valid values include the following.

`'H5_INDEX_NAME'`	An alpha-numeric index by attribute name
`'H5_INDEX_CRT_ORDER'`	An index by creation order

`order` specifies the index traversal order. Valid values include the following.

`'H5_ITER_INC'`	Iteration from beginning to end
`'H5_ITER_DEC'`	Iteration from end to beginning
`'H5_ITER_NATIVE'`	Iteration in the fastest available order

`attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n,aapl_id,lapl_id)` opens an attribute with attribute access property list, `aapl_id`, and link access property list, `lapl_id`. The `aapl_id` argument must currently be specified as `'H5P_DEFAULT'`. Also, `lapl_id` can be specified by `'H5P_DEFAULT'`.

Examples

Loop through a set of dataset attributes in reverse alphabetical order.

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
info = H5O.get_info(dset_id);
for idx = 0:info.num_attrs-1
  attr_id =H5A.open_by_idx(gid,'dset1.1.1','H5_INDEX_NAME','H5_ITER_DEC',idx);
  fprintf('attribute name:  %s\n',H5A.get_name(attr_id));
  H5A.close(attr_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5A.close | H5A.open | H5A.open_by_name

H5A.open_by_name

Open attribute specified by name

Syntax

```
attr_id = H5A.open_by_name(loc_id,obj_name,attr_name)
attr_id =
H5A.open_by_name(loc_id,obj_name,attr_name,aapl_id,lapl_id)
```

Description

`attr_id = H5A.open_by_name(loc_id,obj_name,attr_name)` opens an existing attribute `attr_name` attached to an object specified by its location `loc_id` and name `obj_name`.

`attr_id = H5A.open_by_name(loc_id,obj_name,attr_name,aapl_id,lapl_id)` opens an existing attribute with the attribute access property list `aapl_id` and link access property list `lacpl_id`. `aapl_id` must be specified as `'H5P_DEFAULT'`. `lapl_id` may also be specified by `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
attr_id = H5A.open_by_name(gid,'dset1.1.1','attr1');
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.close` | `H5A.open` | `H5A.open_by_idx`

H5A.read

Read attribute

Syntax

```
attr = H5A.read(attr_id)
attr = H5A.read(attr_id, mem_type_id)
```

Description

`attr = H5A.read(attr_id)` reads the attribute specified by `attr_id`. MATLAB will determine the appropriate memory datatype.

`attr = H5A.read(attr_id, mem_type_id)` reads the attribute specified by `attr_id`. `mem_type_id` specifies the attribute's memory datatype and should usually be given as `'H5ML_DEFAULT'`, which specifies that MATLAB will determine the appropriate memory datatype.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. If the HDF5 library reports the attribute size as 3-by-4-by-5, then the corresponding MATLAB array size is 5-by-4-by-3. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
data = H5A.read(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open` | `H5A.write`

H5A.write

Write attribute

Syntax

```
H5A.write(attr_id,type_id,buf)
```

Description

`H5A.write(attr_id,type_id,buf)` writes the data in buf into the attribute specified by `attr_id`. `type_id` specifies the attribute's memory datatype. The memory datatype should be `'H5ML_DEFAULT'`, which specifies that MATLAB should determine the appropriate memory datatype.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. If the MATLAB array size is 5-by-4-by-3, then the HDF5 library should be reporting the attribute size as 3-by-4-by-5. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Write a scalar double precision attribute.

```
acpl = H5P.create('H5P_ATTRIBUTE_CREATE');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
space_id = H5S.create('H5S_SCALAR');
fid = H5F.create('myfile.h5');
attr_id = H5A.create(fid,'my_attr',type_id,space_id,acpl);
H5A.write(attr_id,'H5ML_DEFAULT',10.0)
H5A.close(attr_id);
H5F.close(fid);
H5T.close(type_id);
```

See Also

`H5A.read`

H5D.close

Close dataset

Syntax

```
H5D.close(dataset_id)
```

Description

`H5D.close(dataset_id)` ends access to a dataset specified by `dataset_id` and releases resources used by it.

See Also

`H5D.create` | `H5D.open`

H5D.create

Create new dataset

Syntax

```
dataset_id = H5D.create(loc_id,name,type_id,space_id,plist_id)
dataset_id =
H5D.create(loc_id,name,type_id,space_id,lcpl_id,dcpl_id,dapl_id)
```

Description

`dataset_id = H5D.create(loc_id,name,type_id,space_id,plist_id)` creates the data set specified by `name` in the file or in the group specified by `loc_id`. `type_id` and `space_id` identify the datatype and dataspace, respectively. `plist_id` identifies the dataset creation property list. This interface corresponds to the `H5Dcreate1` function in the HDF5 library C 1.6 API.

`dataset_id = H5D.create(loc_id,name,type_id,space_id,lcpl_id,dcpl_id,dapl_id)` creates the data set with three distinct property lists:

`lcpl_id`	link creation property list
`dcpl_id`	dataset creation property list
`dapl_id`	dataset access property list

This interface corresponds to the `H5Dcreate` function in the HDF5 library C 1.8 API.

Examples

Create a 10x5 double precision dataset with default property list settings.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [10 5];
h5_dims = fliplr(dims);
```

```
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5S.close(space_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
h5disp('myfile.h5');
```

Create a 6x3 fixed length string dataset. Each string will have a length of 4 characters.

```
fid = H5F.create('myfile_strings.h5');
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,4);
dims = [6 3];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5S.close(space_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
h5disp('myfile_strings.h5');
```

See Also

H5D.close | H5S.close | H5S.create_simple | H5T.copy

H5D.get_access_plist

Copy of dataset access property list

Syntax

```
plist_id = H5D.get_access_plist(dataset_id)
```

Description

`plist_id = H5D.get_access_plist(dataset_id)` returns a copy of the dataset access property list used to open the specified dataset.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dapl = H5D.get_access_plist(dset_id);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_create_plist` | `H5P.close`

H5D.get_create_plist

Copy of dataset creation property list

Syntax

```
plist_id = H5D.get_create_plist(dataset_id)
```

Description

`plist_id = H5D.get_create_plist(dataset_id)` returns the identifier to a copy of the dataset creation property list for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dcpl = H5D.get_create_plist(dset_id);
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_access_plist` | `H5P.close`

H5D.get_offset

Location of dataset in file

Syntax

```
offset = H5D.get_offset(dataset_id)
```

Description

`offset = H5D.get_offset(dataset_id)` returns the location in the file of the dataset specified by `dataset_id`. The location is expressed as an offset, in bytes, from the beginning of the file.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
offset = H5D.get_offset(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.get_space

Copy of dataset data space

Syntax

```
dspace_id = H5D.get_space(dataset_id)
```

Description

`dspace_id = H5D.get_space(dataset_id)` returns an identifier for a copy of the data space for a dataset.

Examples

Retrieve the dimensions of an attribute data space.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
space = H5D.get_space(dset_id);
[~,dims] = H5S.get_simple_extent_dims(space);
H5S.close(space);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.open` | `H5S.close`

H5D.get_space_status

Determine if space is allocated

Syntax

```
status = H5D.get_space_status(dataset_id)
```

Description

`status = H5D.get_space_status(dataset_id)` determines whether space has been allocated for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
status = H5D.get_space_status(dset_id);
switch(status)
    case H5ML.get_constant_value('H5D_SPACE_STATUS_NOT_ALLOCATED')
        fprintf('Not allocated.\n');
    case H5ML.get_constant_value('H5D_SPACE_STATUS_ALLOCATED')
        fprintf('Allocated.\n');
    case H5ML.get_constant_value('H5D_SPACE_STATUS_PART_ALLOCATED')
        fprintf('Part allocated.\n');
end
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_space`

H5D.get_storage_size

Determine required storage size

Syntax

```
dataset_size = H5D.get_storage_size(dataset_id)
```

Description

`dataset_size = H5D.get_storage_size(dataset_id)` returns the amount of storage that is required for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dataset_size = H5D.get_storage_size(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.get_type

Copy of datatype

Syntax

```
type_id = H5D.get_type(dataset_id)
```

Description

`type_id = H5D.get_type(dataset_id)` returns an identifier for a copy of the data type for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
type_id = H5D.get_type(dset_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5T.close`

H5D.open

Open specified dataset

Syntax

```
dataset_id = H5D.open(loc_id,name)
dataset_id = H5D.open(loc_id,name,dapl_id)
```

Description

`dataset_id = H5D.open(loc_id,name)` opens the dataset specified by `name` in the file or group specified by `loc_id`.

`dataset_id = H5D.open(loc_id,name,dapl_id)` opens the dataset specified by `name` in the file or group specified by `loc_id`. The dataset access property list, `dapl_id`, provides information regarding access to the dataset.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
dset_id = H5D.open(gid,'dset2.2');
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.close`

H5D.read

Read data from HDF5 dataset

Syntax

```
data = H5D.read(dataset_id)
data =
H5D.read(dataset_id,mem_type_id,mem_space_id,file_space_id,dxpl)
```

Description

`data = H5D.read(dataset_id)` reads the entire dataset specified by `dataset_id`.

`data =
H5D.read(dataset_id,mem_type_id,mem_space_id,file_space_id,dxpl)` reads the dataset specified by `dataset_id`. The `mem_type_id` input specifies the memory data type and should usually be `'H5ML_DEFAULT'` to allow MATLAB to determine the appropriate value. `mem_space_id` describes how the data is to be arranged in memory and should usually be `'H5S_ALL'`. The `file_space_id` input describes how the data is to be selected from the file. It also can be given as `'H5S_ALL'`, but this results in the entire dataset being read into memory. `dxpl` is the dataset transfer property list identifier and should usually be `'H5P_DEFAULT'`.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Read an entire dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
data = H5D.read(dset_id);
```

```
H5D.close(dset_id);
H5F.close(fid);
```

Read the 2x3 hyperslab starting in the 4th row and 5th column of the example dataset.

```
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dims = fliplr([2 3]);
mem_space_id = H5S.create_simple(2,dims,[]);
file_space_id = H5D.get_space(dset_id);
offset = fliplr([3 4]);
block = fliplr([2 3]);
H5S.select_hyperslab(file_space_id,'H5S_SELECT_SET',offset,[],[],block);
data = H5D.read(dset_id,'H5ML_DEFAULT',mem_space_id,file_space_id,plist);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

H5D.open | H5D.write | H5S.create_simple

H5D.set_extent

Change size of dataset dimensions

Syntax

```
H5D.set_extent(dset_id,h5_extents)
```

Description

`H5D.set_extent(dset_id,h5_extents)` changes the dimensions of the dataset `dset_id` to the sizes specified in `h5_extents`.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_extents` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Extend an unlimited one-dimensional dataset from a length of 10 to a length of 20.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
dset_id = H5D.open(fid,'/g4/time');
H5D.set_extent(dset_id,20);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.vlen_get_buf_size

Determine variable length storage requirements

Syntax

```
size = H5D.vlen_get_buf_size(dataset_id,type_id,space_id)
```

Description

`size = H5D.vlen_get_buf_size(dataset_id,type_id,space_id)` determines the number of bytes required to store the VL data from the dataset, using the `space_id` for the selection in the dataset on disk and the `type_id` for the memory representation of the VL data in memory.

H5D.write

Write data to HDF5 dataset

Syntax

```
H5D.write(dataset_id,mem_type_id,mem_space_id,file_space_id,plist_id,buf)
```

Description

`H5D.write(dataset_id,mem_type_id,mem_space_id,file_space_id,plist_id,buf)` writes the dataset specified by `dataset_id` from the application memory buffer `buf` into the file. `plist_id` specifies the data transfer properties. `mem_type_id` identifies the memory datatype of the dataset. `mem_space_id` and `file_space_id` define the part of the dataset to write. The memory datatype should usually be `'H5ML_DEFAULT'`, which specifies that MATLAB should determine the appropriate memory datatype.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Write to the entire 36-by-19 `/g4/world` example dataset.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g4/world');
dims = [36 19];
data = rand(dims);
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5D.close(dset_id);
H5F.close(fid);
```

Write to the entire two-element `/g3/VLstring` dataset.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
h5disp('myfile.h5','/g3/VLstring');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g3/VLstring');
data = {'dogs'; 'dogs and cats'};
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5D.close(dset_id);
H5F.close(fid);
data_out = h5read('myfile.h5','/g3/VLstring');
```

Write a 10-by-5 block of data to the location starting at row index 15 and column index 5 of the same dataset. Recall that indexing is zero-based.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g4/world');
start = [15 5];
h5_start = fliplr(start);
block = [10 5];
h5_block = fliplr(block);
mem_space_id = H5S.create_simple(2,h5_block,[]);
file_space_id = H5D.get_space(dset_id);
H5S.select_hyperslab(file_space_id,'H5S_SELECT_SET',h5_start,[],[],h5_block);
data = rand(block);
H5D.write(dset_id,'H5ML_DEFAULT',mem_space_id,file_space_id,plist,data);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.read`

H5DS.attach_scale

Attach dimension scale to specific dataset dimension

Syntax

```
H5DS.attach_scale(dataset_id,dimscale_id,idx)
```

Description

`H5DS.attach_scale(dataset_id,dimscale_id,idx)` attaches a dimension scale `dimscale_id` to dimension `idx` of the dataset `dataset_id`.

Note: The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

Add the `'lon'` and `'lat'` dimension scales to the `'world'` dataset.

```
plist = 'H5P_DEFAULT';
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
world_dset_id = H5D.open(fid,'/g4/world',plist);
lat_dset_id = H5D.open(fid,'/g4/lat',plist);
lon_dset_id = H5D.open(fid,'/g4/lon',plist);
H5DS.attach_scale(world_dset_id,lat_dset_id,0);
H5DS.attach_scale(world_dset_id,lon_dset_id,1);
H5D.close(lat_dset_id);
H5D.close(lon_dset_id);
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.detach_scale`

H5DS.detach_scale

Detach dimension scale from specific dataset dimension

Syntax

```
H5DS.detach_scale(dataset_id,dimscale_id,idx)
```

Description

`H5DS.detach_scale(dataset_id,dimscale_id,idx)` detaches dimension scale `dimscale_id` from dimension `idx` of the dataset `dataset_id`.

Note: The ordering of the dimension scale indices are the same as the HDF5 library C API.

See Also

`H5DS.attach_scale`

H5DS.get_label

Retrieve label from specific dataset dimension

Syntax

```
label = H5DS.get_label(dataset_id,idx)
```

Description

`label = H5DS.get_label(dataset_id,idx)` retrieves the label for dimension `idx` of the dataset `dataset_id`.

Note: The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

```
fid = H5F.open('example.h5');
world_dset_id = H5D.open(fid,'/g4/world');
label = H5DS.get_label(world_dset_id,0);
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.set_label`

H5DS.get_num_scales

Number of scales attached to dataset dimension

Syntax

```
num_scales = H5DS.get_num_scales(dataset_id,idx)
```

Description

`num_scales = H5DS.get_num_scales(dataset_id,idx)` determines the number of dimension scales that are attached to dimension `idx` of the dataset `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
world_dset_id = H5D.open(fid,'/g4/world');
num_scales = H5DS.get_num_scales(world_dset_id,0);
H5D.close(world_dset_id);
H5F.close(fid);
```

H5DS.get_scale_name

Name of dimension scale

Syntax

```
name = H5DS.get_scale_name(dimscale_id)
```

Description

`name = H5DS.get_scale_name(dimscale_id)` retrieves the name of the dimension scale `dimscale_id`.

Examples

```
fid = H5F.open('example.h5');
lat_dset_id = H5D.open(fid,'/g4/lat');
scale_name = H5DS.get_scale_name(lat_dset_id);
H5D.close(lat_dset_id);
H5F.close(fid);
```

See Also

`H5DS.set_scale`

H5DS.is_scale

Determine if dataset is a dimension scale

Syntax

```
bool = H5DS.is_scale(dataset_id)
```

Description

`bool = H5DS.is_scale(dataset_id)` determines whether the dataset `dataset_id` is a dimension scale.

Examples

```
fid = H5F.open('example.h5');
lat_dset_id = H5D.open(fid,'/g4/lat');
if H5DS.is_scale(lat_dset_id)
    fprintf('/g4/lat is a dimension scale.\n');
else
    fprintf('/g4/lat is not a dimension scale.\n');
end
H5D.close(lat_dset_id);
H5F.close(fid);
```

H5DS.iterate_scales

Iterate on scales attached to dataset dimension

Syntax

```
[status,idx_out,opdata_out] =
H5DS.iterate_scales(dset_id,dim,idx_in,iter_func,opdata_in)
```

Description

`[status,idx_out,opdata_out] = H5DS.iterate_scales(dset_id,dim,idx_in,iter_func,opdata_in)` iterates over the scales attached to dimension `dim` of the dataset `dset_id` to perform a common operation whose function handle is `iter_func`.

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed. If `idx_in` is `[]`, then the iterator starts at the first member.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(dset_id,dim,dimscale_id,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`dimscale_id` specifies the current dimension scale dataset identifier and `dim` is the associated dimension.

status value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed

positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5DS.set_label

Set label for dataset dimension

Syntax

```
H5DS.set_label(dataset_id,idx,label)
```

Description

`H5DS.set_label(dataset_id,idx,label)` sets a label for dimension `idx` of the dataset `dataset_id`.

Note: The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

```
plist = 'H5P_DEFAULT';
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
world_dset_id = H5D.open(fid,'/g4/world',plist);
H5DS.set_label(world_dset_id,0,'latitude');
H5DS.set_label(world_dset_id,1,'longitude');
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.get_label`

H5DS.set_scale

Convert dataset to dimension scale

Syntax

```
H5DS.set_scale(dataset_id,dim_name)
```

Description

`H5DS.set_scale(dataset_id,dim_name)` converts the dataset, `dataset_id`, to a dimension scale with name `dim_name`.

Examples

Create a dimension scale with name `'xdim'`. The dataset has the name, `'x'`.

```
fid = H5F.create('myfile.h5');
space_id = H5S.create_simple(1,10,10);
dtype = 'H5T_NATIVE_INT';
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'x',dtype,space_id,dcpl);
H5DS.set_scale(dset_id,'xdim');
H5S.close(space_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5DS.get_scale_name`

H5E.clear

Clear error stack

Syntax

```
H5E.clear()
```

Description

`H5E.clear()` clears the error stack for the current thread.

H5E.get_major

Description of major error number

Syntax

```
err_string = H5E.get_major(major_number)
```

Description

`err_string = H5E.get_major(major_number)` returns a character string describing an error specified by the major error number, `major_number`.

The HDF5 group has deprecated the use of this function.

See Also

`H5E.get_minor`

H5E.get_minor

Description of minor error number

Syntax

```
err_string = H5E.get_minor(minor_number)
```

Description

`err_string = H5E.get_minor(minor_number)` returns a character string describing an error specified by the minor error number, `minor_number`.

The HDF5 group has deprecated the use of this function.

See Also

`H5E.get_major`

H5E.walk

Walk error stack

Syntax

```
H5E.walk(direction,func)
```

Description

`H5E.walk(direction,func)` walks the error stack for the current thread and calls the specified function for each error along the way. `func` is a function handle. `direction` specifies how the error stack is traversed and can be given by one of the following strings or the numeric equivalent.

```
'H5E_WALK_UPWARD'
'H5E_WALK_DOWNWARD'
```

The specified function must have the following signature:

```
status = func(n,error_struct)
```

where `n` is the indexed position of the error in the stack and `error_struct` is a structure with the following fields:

`maj_num`	Major error number
`min_num`	Minor error number
`func_name`	Function in which the error occurred
`file_name`	File in which the error occurred
`line`	Line in file where error occurs
`desc`	Optional supplied description

This function corresponds to the `H5Ewalk1` function in the HDF5 library C API.

See Also

`H5ML.get_constant_value`

H5F.close

Close HDF5 file

Syntax

```
H5F.close(file_id)
```

Description

`H5F.close(file_id)` terminates access to HDF5 file identified by `file_id`, flushing all data to storage.

See Also

`H5F.open`

H5F.create

Create HDF5 file

Syntax

```
file_id = H5F.create(filename)
file_id = H5F.create(name,flags,fcpl_id,fapl_id)
```

Description

`file_id = H5F.create(filename)` creates the file specified by `filename` with default library properties if the file does not already exist.

`file_id = H5F.create(name,flags,fcpl_id,fapl_id)` creates the file specified by `name`. `flags` specifies whether to truncate the file, if it already exists, or to fail if the file already exists. `flags` can be specified by one of the following strings or the numeric equivalent:

`'H5F_ACC_TRUNC'`	overwrite any existing file by the same name
`'H5F_ACC_EXCL'`	do not overwrite an existing file

`fcpl_id` is the file creation property list identifier. `fapl_id` is the file access property list identifier. A value of `'H5P_DEFAULT'` for either property list indicates that the library should use default values for the appropriate property list.

Examples

Create an HDF5 file called `'myfile.h5'`.

```
fid = H5F.create('myfile.h5');
H5F.close(fid);
```

Create an HDF5 file called `'myfile.h5'`, overwriting any existing file by the same name. Default file access and file creation properties shall apply.

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

H5F.close | H5ML.get_constant_value | H5P.create

H5F.flush

Flush buffers to disk

Syntax

```
H5F.flush(object_id,scope)
```

Description

`H5F.flush(object_id,scope)` causes all buffers associated with a file to be immediately flushed to disk without removing the data from the cache. `object_id` can be any object associated with the file, including the file itself, a dataset, a group, an attribute, or a named data type. `scope` specifies whether the scope of the flushing action is global or local. `scope` may be one of the following strings:

```
'H5F_SCOPE_GLOBAL'
'H5F_SCOPE_LOCAL'
```

H5F.get_access_plist

File access property list

Syntax

```
fapl_id = H5F.get_access_plist(file_id)
```

Description

`fapl_id = H5F.get_access_plist(file_id)` returns the file access property list identifier of the file specified by `file_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5F.get_create_plist`

H5F.get_create_plist

File creation property list

Syntax

```
fcpl_id = H5F.get_create_plist(file_id)
```

Description

`fcpl_id = H5F.get_create_plist(file_id)` returns a file creation property list identifier identifying the creation properties used to create the file specified by `file_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5F.get_access_plist`

H5F.get_filesize

Size of HDF5 file

Syntax

```
size = H5F.get_filesize(file_id)
```

Description

`size = H5F.get_filesize(file_id)` returns the size of the HDF5 file specified by `file_id`.

H5F.get_freespace

Amount of free space in file

Syntax

```
free_space = H5F.get_freespace(file_id)
```

Description

`free_space = H5F.get_freespace(file_id)` returns the amount of space that is unused by any objects in the file specified by `file_id`.

H5F.get_info

Global information about file

Syntax

```
file_info = H5F.get_info(obj_id)
```

Description

`file_info = H5F.get_info(obj_id)` returns global information for the file associated with the object identifier `obj_id`. For details about the fields of the `file_info` structure, please refer to the HDF5 documentation.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'g2');
info = H5F.get_info(gid);
H5G.close(gid);
H5F.close(fid);
```

H5F.get_mdc_config

Metadata cache configuration

Syntax

```
config_struct = H5F.get_mdc_config(file_id)
```

Description

`config_struct = H5F.get_mdc_config(file_id)` returns the current metadata cache configuration for the target file.

Examples

```
fid = H5F.open('example.h5');
config = H5F.get_mdc_config(fid);
H5F.close(fid);
```

See Also

`H5F.set_mdc_config`

H5F.get_mdc_hit_rate

Metadata cache hit-rate

Syntax

```
hitRate = H5F.get_mdc_hit_rate(file_id)
```

Description

`hitRate = H5F.get_mdc_hit_rate(file_id)` queries the metadata cache of the target file to obtain its hit-rate since the last time hit-rate statistics were reset. If the cache has not been accessed since the last time the hit-rate statistics were reset, the hit-rate is defined to be 0.0. The hit-rate is calculated as

```
(cache hits / (cache hits + cache misses))
```

Examples

```
fid = H5F.open('example.h5');
hit_rate = H5F.get_mdc_hit_rate(fid);
H5F.close(fid);
```

See Also

`H5F.get_mdc_config`

H5F.get_mdc_size

Metadata cache size data

Syntax

```
[max_sz,min_clean_sz,cursz,num_cur_entries] =
H5F.get_mdc_size(fileId)
```

Description

`[max_sz,min_clean_sz,cursz,num_cur_entries] = H5F.get_mdc_size(fileId)` queries the metadata cache of the target file to obtain current metadata cache size information.

Examples

```
fid = H5F.open('example.h5');
[maxsz,minsz,cursz,nent] = H5F.get_mdc_size(fid);
H5F.close(fid);
```

See Also

`H5F.get_mdc_config`

H5F.get_name

Name of HDF5 file

Syntax

```
name = H5F.get_name(obj_id)
```

Description

`name = H5F.get_name(obj_id)` returns the name of the file to which the object `obj_id` belongs. The object can be a group, dataset, attribute, or named data type.

Examples

```
fid = H5F.open('example.h5');
name = H5F.get_name(fid);
H5F.close(fid);
```

See Also

`H5A.get_name` | `H5I.get_name`

H5F.get_obj_count

Number of open objects in HDF5 file

Syntax

```
obj_count = H5F.get_obj_count(file_id,types)
```

Description

`obj_count = H5F.get_obj_count(file_id,types)` returns the number of open object identifiers for the file specified by `file_id` for the specified type. `types` can be one of the following strings.

```
'H5F_OBJ_FILE'
'H5F_OBJ_DATASET'
'H5F_OBJ_GROUP'
'H5F_OBJ_DATATYPE'
'H5F_OBJ_ATTR'
'H5F_OBJ_ALL'
'H5F_OBJ_LOCAL'
```

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
obj_count = H5F.get_obj_count(fid,'H5F_OBJ_GROUP');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5F.get_obj_ids`

H5F.get_obj_ids

List of open HDF5 file objects

Syntax

```
[num_obj_ids,obj_id_list] = H5F.get_obj_ids(file_id,types,max_objs)
```

Description

`[num_obj_ids,obj_id_list] = H5F.get_obj_ids(file_id,types,max_objs)` returns a list of all open identifiers for HDF5 objects of the type specified by `types` in the file specified by `file_id`. The `max_objs` input specifies the maximum number of object identifiers to return. `num_obj_ids` is the total number of objects in the list. `types` can be one of the following strings.

```
'H5F_OBJ_FILE'
'H5F_OBJ_DATASET'
'H5F_OBJ_GROUP'
'H5F_OBJ_DATATYPE'
'H5F_OBJ_ATTR'
'H5F_OBJ_ALL'
'H5F_OBJ_LOCAL'
```

Examples

```
fid = H5F.open('example.h5');
gid1 = H5G.open(fid,'/g1');
gid2 = H5G.open(fid,'/g2');
gid3 = H5G.open(fid,'/g3');
gid4 = H5G.open(fid,'/g4');
[num_obj_ids,objs] = H5F.get_obj_ids(fid,'H5F_OBJ_GROUP',3);
H5G.close(gid1);
H5G.close(gid2);
```

```
H5G.close(gid3);
H5G.close(gid4);
H5F.close(fid);
```

See Also

H5F.get_obj_count

H5F.is_hdf5

Determine if file is HDF5

Syntax

```
value = H5F.is_hdf5(name)
```

Description

`value = H5F.is_hdf5(name)` returns a positive number if the file specified by `name` is in the HDF5 format, and zero if it is not. A negative return value indicates failure.

Examples

```
value = H5F.is_hdf5('example.tif');
if value > 0
    fprintf('example.tif is an HDF5 file\n');
else
    fprintf('example.tif is not an HDF5 file\n');
end
```

H5F.mount

Mount HDF5 file onto specified location

Syntax

```
H5F.mount(loc_id,name,child_id,plist_id)
```

Description

`H5F.mount(loc_id,name,child_id,plist_id)` mounts the file specified by `child_id` onto the group specified by `loc_id` and `name`, using the mount properties specified by `plist_id`.

Examples

Mount one file with a dataset onto a group in a second file and access the dataset via the second file.

```
plist = 'H5P_DEFAULT';
fid2 = H5F.create('file2.h5','H5F_ACC_TRUNC',plist,plist);
gid2 = H5G.create(fid2,'g2',plist,plist,plist);
fid1 = H5F.create('file1.h5','H5F_ACC_TRUNC','H5P_DEFAULT',...
  'H5P_DEFAULT');
space_id = H5S.create('H5S_SCALAR');
dset_id = H5D.create(fid1,'DS1','H5T_NATIVE_DOUBLE',space_id,plist);
H5S.close(space_id);
H5D.close(dset_id);
H5F.mount(fid2,'g2',fid1,plist);
dset_id1 = H5D.open(fid1,'/g2/DS1',plist);
H5D.close(dset_id1);
H5F.unmount(fid1,'g2');
H5G.close(gid2);
H5F.close(fid1);
H5F.close(fid2);
```

See Also

`H5F.unmount`

H5F.open

Open HDF5 file

Syntax

```
file_id = H5F.open(filename)
file_id = H5F.open(name,flags,fapl_id)
```

Description

`file_id = H5F.open(filename)` opens the file specified by `filename` for read-only access and returns the file identifier, `file_id`.

`file_id = H5F.open(name,flags,fapl_id)` opens the file specified by `name`, returning the file identifier, `file_id`. `flags` specifies file access flags and can be specified by one of the following strings or their numeric equivalents:

`'H5F_ACC_RDWR'`	read-write mode
`'H5F_ACC_RDONLY'`	read-only mode

The file access property list, `fapl_id`, may be specified as `'H5P_DEFAULT'`, in which case the default I/O settings are used.

Examples

Open a file in read-only mode with default file access properties.

```
fid = H5F.open('example.h5');
H5F.close(fid);
```

Open a file in read-write mode.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
```

```
H5F.close(fid);
```

See Also

H5F.close | H5ML.get_constant_value

H5F.reopen

Reopen HDF5 file

Syntax

```
new_file_id = H5F.reopen(file_id)
```

Description

`new_file_id = H5F.reopen(file_id)` returns a new file identifier for the already open HDF5 file specified by `file_id`.

See Also

`H5F.open`

H5F.set_mdc_config

Configure HDF5 file metadata cache

Syntax

```
H5F.set_mdc_config(fileId,config)
```

Description

`H5F.set_mdc_config(fileId,config)` attempts to configure the file's metadata cache according to the supplied configuration structure. Before using this function, you should retrieve the current configuration using `H5F.get_mdc_config`.

See Also

`H5F.get_mdc_config`

H5F.unmount

Unmount file or group from mount point

Syntax

```
H5F.unmount(loc_id,name)
```

Description

`H5F.unmount(loc_id,name)` dissassociates the file or group specified by `loc_id` from the mount point specified by `name`. `loc_id` can be a file or group identifier.

See Also

`H5F.mount`

H5G.close

Close group

Syntax

```
H5G.close(group_id)
```

Description

`H5G.close(group_id)` releases resources used by the group specified by `group_id`. `group_id` was returned by either `H5G.create` or `H5G.open`.

See Also

`H5G.create` | `H5G.open`

H5G.create

Create group

Syntax

```
group_id = H5G.create(loc_id,name,size_hint)
group_id = H5G.create(loc_id,name,lcpl_id,gcpl_id,gapl_id)
```

Description

`group_id = H5G.create(loc_id,name,size_hint)` creates a new group with the name specified by `name` at the location specified by `loc_id`. `loc_id` can be a file or group identifier. `size_hint` specifies the number of bytes to reserve for the names that will appear in the group. This interface corresponds to the 1.6 version of `H5Gcreate`.

`group_id = H5G.create(loc_id,name,lcpl_id,gcpl_id,gapl_id)` creates a new group with link creation, group creation, and group access property lists `lcpl_id`, `gcpl_id`, and `gapl_id`. This interface corresponds to the 1.8 version of `H5Gcreate`.

Examples

Create an HDF5 file `'myfile.h5'` with a group `'my_group'` with default property list settings.

```
fid = H5F.create('myfile.h5');
plist = 'H5P_DEFAULT';
gid = H5G.create(fid,'my_group',plist,plist,plist);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.close` | `H5G.open`

H5G.get_info

Information about group

Syntax

```
info = H5G.get_info(group_id)
```

Description

`info = H5G.get_info(group_id)` retrieves information about the group specified by `group_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
info = H5G.get_info(gid);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.create` | `H5G.open`

H5G.open

Open specified group

Syntax

```
group_id = H5G.open(loc_id,name)
group_id = H5G.open(loc_id,name,gapl_id)
```

Description

`group_id = H5G.open(loc_id,name)` opens the group specified by `name` at the location specified by `loc_id`. `loc_id` is a file or group identifier. This interface corresponds to the 1.6 version of `H5Gopen`.

`group_id = H5G.open(loc_id,name,gapl_id)` opens the group with an additional group access property list, `gapl_id`. This interface corresponds to the 1.8 version of `H5Gopen`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.close` | `H5P.create`

H5I.dec_ref

Decrement reference count

Syntax

```
ref_count = H5I.dec_ref(obj_id)
```

Description

`ref_count = H5I.dec_ref(obj_id)` decrements the reference count of the object identified by `obj_id` and returns the new count.

See Also

`H5I.get_ref` | `H5I.inc_ref`

H5I.get_file_id

File identifier for specified object

Syntax

```
file_id = H5I.get_file_id(obj_id)
```

Description

`file_id = H5I.get_file_id(obj_id)` returns the identifier of the file associated with the object referenced by `obj_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g4');
fid2 = H5I.get_file_id(gid);
name = H5F.get_name(fid2);
fprintf('The filename is %s.\n',name);
H5G.close(gid);
H5F.close(fid);
H5F.close(fid2);
```

H5I.get_name

Name of object

Syntax

```
name = H5I.get_name(obj_id)
```

Description

`name = H5I.get_name(obj_id)` returns the name of the object specified by `obj_id`. If no name is attached to the object, the empty string is returned.

Examples

Display the names of all the objects in the `/g3` group in the example file by alphabetical order.

```
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_INC';
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g3');
info = H5G.get_info(gid);
for j = 1:info.nlinks
    obj_id = H5O.open_by_idx(fid,'g3',idx_type,order,j-1,'H5P_DEFAULT');
    name = H5I.get_name(obj_id);
    fprintf('Object %d: ''%s''.\n',j-1,name);
    H5O.close(obj_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.get_name` | `H5F.get_name`

H5I.get_ref

Reference count of object

Syntax

```
refcount = H5I.get_ref(obj_id)
```

Description

`refcount = H5I.get_ref(obj_id)` returns the reference count of the object specified by `obj_id`.

See Also

`H5I.dec_ref` | `H5I.inc_ref`

H5I.get_type

Type of object

Syntax

```
obj_type = H5I.get_type(obj_id)
```

Description

`obj_type = H5I.get_type(obj_id)` returns the type of the object specified by `obj_id`. `obj_type` corresponds to one of the following enumerated values.

```
H5I_FILE
H5I_GROUP
H5I_DATATYPE
H5I_DATASPACE
H5I_DATASET
H5I_ATTR
H5I_BADID
```

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g3');
dset_id = H5D.open(fid,'/g4/world');
[~,objs] = H5F.get_obj_ids(fid,'H5F_OBJ_ALL',3);
for j = 1:numel(objs)
    name = H5I.get_name(objs(j));
    fprintf('object ''%s'':  ==> ',name);
    type = H5I.get_type(objs(j));
    switch(type)
        case H5ML.get_constant_value('H5I_FILE')
            fprintf('FILE identifier.\n');
```

```
        case H5ML.get_constant_value('H5I_GROUP')
            fprintf('GROUP identifier.\n');
        case H5ML.get_constant_value('H5I_DATASET')
            fprintf('DATASET identifier.\n');
        otherwise
            fprintf('unknown identifier type.\n');
    end
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5ML.get_constant_value

H5I.inc_ref

Increment reference count of specified object

Syntax

```
ref_count = H5I.inc_ref(obj_id)
```

Description

`ref_count = H5I.inc_ref(obj_id)` increments the reference count of the object specified by `obj_id` and returns the new count.

See Also

`H5I.dec_ref` | `H5I.get_ref`

H5I.is_valid

Determine if specified identifier is valid

Syntax

```
tf = H5I.is_valid(obj_id)
```

Description

`tf = H5I.is_valid(obj_id)` determines whether the identifier `obj_id` is valid.

Examples

```
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.close(fapl);
if H5I.is_valid(fapl);
    fprintf('File access property list is valid.\n');
else
    fprintf('File access property list is not valid.\n');
end
```

H5L.copy

Copy link from source location to destination location

Syntax

```
H5L.copy(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)
```

Description

`H5L.copy(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)` copies the link specified by `src_name` from the file or group specified by `src_loc_id` to the destination `dest_loc_id`. The new copy of the link is created with the name `dest_name`.

`dest_loc_id` must refer to either the current file or a group in the current file. If `dest_loc_id` is the file identifier, the copy is placed in the file's root group.

The new link is created with the creation and access property lists specified by `lcpl_id` and `lapl_id`.

Examples

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
g1 = H5G.create(fid,'g1',plist_id);
g2 = H5G.create(fid,'g2',plist_id);
g11 = H5G.create(g1,'g3',plist_id);
H5L.copy(g1,'g3',g2,'g4',plist_id,plist_id);
```

H5L.create_external

Create soft link to external object

Syntax

```
H5L.create_external(filename,objname,link_loc_id,link_name,lcpl_id,lapl_id)
```

Description

`H5L.create_external(filename,objname,link_loc_id,link_name,lcpl_id,lapl_id)` creates a soft link to an object in a different file. `filename` identifies the target file containing the target object. `obj_name` specifies the path to the target object within that file. `obj_name` must start at the target file's root group but is not interpreted until lookup time.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
plist_id = 'H5P_DEFAULT';
fid1 = H5F.create('myfile1.h5');
g1 = H5G.create(fid1,'g1',plist_id,plist_id,plist_id);
H5G.close(g1);
H5F.close(fid1);
fid2 = H5F.create('myfile2.h5');
H5L.create_external('myfile1.h5','g1',fid2,'g2',plist_id,plist_id);
```

H5L.create_hard

Create hard link

Syntax

```
H5L.create_hard(obj_loc_id,obj_name,link_loc_id,link_name,lcpl_id,lapl_id)
```

Description

`H5L.create_hard(obj_loc_id,obj_name,link_loc_id,link_name,lcpl_id,lapl_id)` creates a new hard link to a pre-existing object in an HDF5 file. The new link may be one of many that point to that object. `obj_loc_id` and `obj_name` specify the location and name, respectively, of the target object, i.e., the object to which the new hard link points.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
fid = H5F.create('myfile.h5');
gid1 = H5G.create(fid,'/g1',0);
gid2 = H5G.create(gid1,'g2',0);
gid3 = H5G.create(gid2,'g3',0);
lcpl = 'H5P_DEFAULT';
lapl = 'H5P_DEFAULT';
H5L.create_hard(gid2,'g3',gid1,'g4',lcpl,lapl);
H5G.close(gid3);
H5G.close(gid2);
H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_soft`

H5L.create_soft

Create soft link

Syntax

```
H5L.create_soft(target_path,link_loc_id,link_name,lcpl_id,lapl_id)
```

Description

`H5L.create_soft(target_path,link_loc_id,link_name,lcpl_id,lapl_id)` creates a new soft link to an object in an HDF5 file. The new link may be one of many that point to that object. `target_path` specifies the path to the target object, i.e., the object that the new soft link points to. `target_path` can be anything and is interpreted at lookup time. This `target_path` may be absolute in the file or relative to `link_loc_id`.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5');
gid1 = H5G.create(fid,'/g1',0);
gid3 = H5G.create(gid1,'g3',0);
gid2 = H5G.create(fid,'/g2',0);
lcpl = 'H5P_DEFAULT';
lapl = 'H5P_DEFAULT';
H5L.create_soft('/g1/g3',gid2,'g4',lcpl,lapl);
H5G.close(gid3);
H5G.close(gid2);
H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_hard`

H5L.delete

Remove link

Syntax

```
H5L.delete(loc_id,name,lapl_id)
```

Description

`H5L.delete(loc_id,name,lapl_id)` removes the link specified by name from the location `loc_id`. `lapl_id` is a link access property list identifier.

Examples

Remove the only link to the `'/g3'` group in `example.h5`.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
H5L.delete(fid,'g3','H5P_DEFAULT');
H5F.close(fid);
```

See Also

`H5L.move`

H5L.exists

Determine if link exists

Syntax

```
bool = H5L.exists(loc_id,name,lapl_id)
```

Description

`bool = H5L.exists(loc_id,name,lapl_id)` checks if a link specified by the pairing of an object id and name exists within a group. `lapl_id` is a link access property list identifier.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.2/g1.2.1');
if H5L.exists(gid,'slink','H5P_DEFAULT')
    fprintf('link exists\n');
else
    fprintf('link does not exist\n');
end
```

H5L.get_info

Information about link

Syntax

```
linkStruct = H5L.get_info(location_id,link_name,lapl_id)
```

Description

`linkStruct = H5L.get_info(location_id,link_name,lapl_id)` returns information about a link.

A file or group identifier, `location_id`, specifies the location of the link. `link_name`, interpreted relative to `link_id`, specifies the link being queried.

Examples

```
fid = H5F.open('example.h5');
info = H5L.get_info(fid,'g3','H5P_DEFAULT');
H5F.close(fid);
```

H5L.get_name_by_idx

Information about link specified by index

Syntax

```
name =
H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,lapl_id)
```

Description

`name = H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,lapl_id)` retrieves information about a link at index `n`, present in group `group_name`, at location `loc_id`. The `lapl_id` input specifies the link access property list for querying the group.

`idx_type` is the type of index and valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the index traversal order. Valid values include the following.

`'H5_ITER_INC'`	Iteration from beginning to end
`'H5_ITER_DEC'`	Iteration from end to beginning
`'H5_ITER_NATIVE'`	Iteration in the fastest available order

Examples

```
fid = H5F.open('example.h5');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
lapl_id = 'H5P_DEFAULT';
name = H5L.get_name_by_idx(fid,'g3',idx_type,order,0,lapl_id);
H5F.close(fid);
```

H5L.get_val

Value of symbolic link

Syntax

```
linkval = H5L.get_val(link_loc_id,link_name,lapl_id)
```

Description

`linkval = H5L.get_val(link_loc_id,link_name,lapl_id)` returns the value of a symbolic link.

`link_loc_id` is a file or group identifier. `link_name` identifies a symbolic link and is defined relative to `link_loc_id`. Symbolic links include soft and external links and some user-defined links.

In the case of soft links, `linkval` is a cell array containing the path to which the link points.

In the case of external links, `linkval` is a cell array consisting of the name of the target file and the object name.

This function corresponds to the `H5L.get_val` and `H5Lunpack_elink_val` functions in the HDF5 1.8 C API.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.2/g1.2.1');
linkval = H5L.get_val(gid,'slink','H5P_DEFAULT');
H5G.close(gid);
H5F.close(fid);
```

H5L.iterate

Iterate over links

Syntax

```
[status,idx_out,opdata_out] =
H5L.iterate(group_id,index_type,order,idx_in,iter_func,opdata_in)
```

Description

`[status,idx_out,opdata_out] = H5L.iterate(group_id,index_type,order,idx_in,iter_func,opdata_in)` iterates through the links in a group, specified by `group_id`, to perform a common function whose function handle is `iter_func`. `H5L.iterate` does not recursively follow links into subgroups of the specified group.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they will first be sorted by that index then the iteration will begin. If the links have been so indexed, the sorting step will be unnecessary, so the iteration may begin more quickly. Valid values include the following:

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following:

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.iterate_by_name

Iterate through links in group specified by name

Syntax

```
[status,idx_out,opdata_out] =
H5L.iterate_by_name(loc_id,group_name,index_type,order,idx_in,iter_func,opdata
```

Description

`[status,idx_out,opdata_out] = H5L.iterate_by_name(loc_id,group_name,index_type,order,idx_in,iter_func,opdata` iterates through the links in a group to perform a common function whose function handle is `iter_func`. The starting point of the iteration is pairing of a specified by the location id and a relative group name. `H5L.iterate_by_name` does not recursively follow links into subgroups of the specified group. A link access property list, `lapl_id`, may affect the outcome depending upon the type of link being traversed.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they will first be sorted by that index then the iteration will begin. If the links have been so indexed, the sorting step will be unnecessary, so the iteration may begin more quickly. Valid values include the following:

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following:

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final opdata_out at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.move

Rename link

Syntax

```
H5L.move(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)
```

Description

`H5L.move(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)` renames a link within an HDF5 file. The original link, `src_name`, is removed from the group graph and the new link, `dest_name`, is inserted; this change is accomplished as an atomic operation.

`src_loc_id` and `src_name` identify the existing link. `src_loc_id` is either a file or group identifier; `src_name` is the path to the link and is interpreted relative to `src_loc_id`.

`dest_loc_id` and `dest_name` identify the new link. `dest_loc_id` is either a file or group identifier; `dest_name` is the path to the link and is interpreted relative to `dest_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and link access property lists, respectively, associated with the new link, `dest_name`.

Examples

Rename the `'/g2'` group to `'/g2/g3'`.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
g2id = H5G.open(fid,'g2');
H5L.move(fid,'g3',g2id,'g3','H5P_DEFAULT','H5P_DEFAULT');
H5G.close(g2id);
```

```
H5F.close(fid);
```

See Also

H5L.delete

H5L.visit

Recursively iterate through links in group specified by group identifier

Syntax

```
[status opdata_out] =
H5L.visit(group_id,index_type,order,iter_func,opdata_in)
```

Description

`[status opdata_out] = H5L.visit(group_id,index_type,order,iter_func,opdata_in)` recursively iterates through all links in and below a group, specified by `group_id`, to perform a common function whose function handle is `iter_func`.

`index_type` and order establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.)

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order

`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.visit_by_name

Recursively iterate through links in group specified by location and group name

Syntax

```
[status,opdata_out] =
H5L.visit_by_name(loc_id,group_name,index_type,order,iter_func,opdata_in,lapl_
```

Description

`[status,opdata_out] = H5L.visit_by_name(loc_id,group_name,index_type,order,iter_func,opdata_in,lapl_` recursively iterates though all links in and below a group to perform a common function whose function handle is `iter_func`. The starting point of the iteration is specified by the pairing of a location id and a relative group name. A link access property list, `lapl_id`, may affect the outcome depending upon the type of link being traversed.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.)

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

`'H5_ITER_INC'`	Increasing order

`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5ML.compare_values

Numerically compare two HDF5 values

Syntax

```
bEqual = H5ML.compare_values(value1,value2)
```

Description

`bEqual = H5ML.compare_values(value1,value2)` compares two values, where either or both values may be represented as a string. The values are compared numerically.

Function parameters:

`bEqual`	A logical value indicating whether the two values are equal
`value1`	The first value to be compared
`value2`	The second value to be compared

Examples

```
val = H5ML.get_constant_value('H5T_NATIVE_INT');
H5ML.compare_values(val,'H5T_NATIVE_INT')
```

H5ML.get_constant_names

Constants known by HDF5 library

Syntax

```
names = H5ML.get_constant_names()
```

Description

`names = H5ML.get_constant_names()` returns a list of known library constants, definitions, and enumerations. When these strings are supplied as actual parameters to HDF5 functions, they are automatically be converted to the appropriate numeric value.

Function parameters.

`names`	An alphabetized cell array of names

H5ML.get_constant_value

Value corresponding to a string

Syntax

```
value = H5ML.get_constant_value(constant)
```

Description

`value = H5ML.get_constant_value(constant)` returns the value corresponding to a given string. The string should correspond to an enumeration (for example, `'H5_ENUM_T'`) or a predefined identifier (for example, `'H5T_NATIVE_INT'`). Since the value corresponding to a given string is not guaranteed to remain the same, it is almost always preferable to use the `H5ML.compare_values()` function instead.

Function parameters:

`value`	The value corresponding to the supplied string
`constant`	A string that corresponds to a HDF5 enumeration or defined value.

Examples

```
a = H5ML.get_constant_value('H5T_NATIVE_INT');
```

H5ML.get_function_names

Functions provided by HDF5 library

Syntax

```
names = H5ML.get_function_names()
```

Description

`names = H5ML.get_function_names()` returns a list of supported library functions.

Function parameters:

`names`	An alphabetized cell array of names

H5ML.get_mem_datatype

Data type for dataset ID

Syntax

```
DTYPE_ID = H5ML.get_mem_datatype(LOCATION_ID)
```

Description

`DTYPE_ID = H5ML.get_mem_datatype(LOCATION_ID)` returns the ID of an HDF5 memory datatype for the dataset or attribute identified by `LOCATION_ID`. This HDF5 memory datatype is the default used by `H5D.read` or `H5D.write` when you specify `'H5ML_DEFAULT'` as a value of the memory data type parameter.

The identifier returned by `H5ML.get_mem_datatype` should eventually be closed by calling `H5T.close` to release resources.

Examples

```
file_id = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
dset_id = H5D.open(file_id,'/g1/g1.1/dset1.1.1');
datatype_id = H5ML.get_mem_datatype(dset_id)
H5T.close(datatype_id);
H5D.close(dset_id);
H5F.close(file_id);
```

H5ML.hoffset

Determine the offset of a field within a structure

Note: `H5ML.hoffset` is not recommended. Use `H5T` instead.

Syntax

Description

This function is used to determine the offset (in bytes) of a structure, `H5T.insert(file_type,'a', offset(1), dtype(1));`, within a field. It is used when constructing an HDF5 COMPOUND type. It is designed to correspond to the HDF5 HOFFSET macro. For more details about the operation of the HOFFSET macro, please consult the HDF5 documentation.

Function parameters:

`offset`	The byte offset of the field within the structure.
`structure`	The structure which contains the specified fieldname.
`fieldname`	The field for which the offset is determined.

Examples

This function is deprecated. It can only be used in workflows that do not include a field that is itself an HDF5 COMPOUND or of variable length. To handle these cases, the offsets should be computed directly. For example, in the case above, a file dataspace for such a compound could be created with:

```
dtype(1) = H5T.copy('H5T_NATIVE_INT');
```

```
dtype(2) = H5T.copy('H5T_NATIVE_DOUBLE');
dtype(3) = H5T.copy('H5T_NATIVE_FLOAT');

for j = 1:3, sz(j,1) = H5T.get_size(dtype(j)); end

% The first offset would always be zero and the size of the last
% field does not matter.
offset(1) = 0;
offset(2:3) = cumsum(sz(1:2));

file_type = H5T.create('H5T_COMPOUND',sum(sz));

H5T.insert(file_type,'a', offset(1), dtype(1));
H5T.insert(file_type,'b', offset(2), dtype(2));
H5T.insert(file_type,'c', offset(3), dtype(3));
```

See Also

H5T.get_size

H5ML.sizeof

Return the size (in bytes) of a variable as stored on disk

Note: `H5ML.sizeof` is not recommended. Use `H5T` instead.

Syntax

Description

This function is used to determine the size (in bytes) of a structure or other (simple) variable. It is designed to correspond to the C sizeof() operator as it is used during the creation of HDF5 datatypes, especially the HDF5 COMPOUND type.

Function parameters:

`size`	The size (in bytes) of the variable as it would be stored on disk.
`arg`	The variable for which the size is being sought.

Examples

This function is deprecated. It can only be used in workflows that do not include a field that is itself an HDF5 COMPOUND or of variable length. To handle these cases, the offsets should be computed directly. For example, in the case above, a file dataspace for such a compound could be created with:

```
dtype(1) = H5T.copy('H5T_NATIVE_INT');
dtype(2) = H5T.copy('H5T_NATIVE_DOUBLE');
dtype(3) = H5T.copy('H5T_NATIVE_FLOAT');

for j = 1:3, sz(j,1) = H5T.get_size(dtype(j)); end
```

```
% The first offset would always be zero and the size of the last
% field does not matter.
offset(1) = 0;
offset(2:3) = cumsum(sz(1:2));

file_type = H5T.create('H5T_COMPOUND',sum(sz));

H5T.insert(file_type,'a', offset(1), dtype(1));
H5T.insert(file_type,'b', offset(2), dtype(2));
H5T.insert(file_type,'c', offset(3), dtype(3));
```

See Also

H5T.get_size

H5O.close

Close object

Syntax

```
H5O.close(obj_id)
```

Description

`H5O.close(obj_id)` closes the object specified by `obj_id`. `obj_id` cannot be a dataspace, attribute, property list, or file.

H5O.copy

Copy object from source location to destination location

Syntax

```
H5O.copy(src_loc_id,src_name,dst_loc_id,dst_name,ocpypl_id,lcpl_id)
```

Description

`H5O.copy(src_loc_id,src_name,dst_loc_id,dst_name,ocpypl_id,lcpl_id)` copies the group, dataset or named datatype specified by `src_name` from the file or group specified by `src_loc_id` to the destination location `dst_loc_id`.

The destination location, as specified in `dst_loc_id`, may be a group in the current file or a location in a different file. If `dst_loc_id` is a file identifier, the copy will be placed in that file's root group.

The new copy will be created with the name `dst_name`. `dst_name` must not pre-exist in the destination location. If `dst_name` already exists at the location `dst_loc_id`, the operation will fail.

The new copy of the object is created with the object creation property and link creation property lists `ocpypl_id` and `lcpl_id`, respectively.

Examples

Copy the group `'/g3'` and all its datasets to a new group `'/g3.5'`.

```
srcFile = [matlabroot '/toolbox/matlab/demos/example.h5'];
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
ocpl = H5P.create('H5P_OBJECT_COPY');
lcpl = H5P.create('H5P_LINK_CREATE');
H5P.set_create_intermediate_group(lcpl,true);
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
gid = H5G.open(fid,'/');
```

```
H5O.copy(gid,'g3',gid,'g3.5',ocpl,lcpl);
H5G.close(gid);
H5P.close(ocpl);
H5P.close(lcpl);
H5F.close(fid);
```

H5O.get_comment

Get comment for object specified by object identifier

Syntax

```
comment = H5O.get_comment(obj_id)
```

Description

`comment = H5O.get_comment(obj_id)` retrieves the comment for the object specified by `obj_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'g4/world');
comment = H5O.get_comment(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5O.get_comment_by_name` | `H5O.set_comment` | `H5O.set_comment_by_name`

H5O.get_comment_by_name

Get comment for object specified by location and object name

Syntax

```
comment = H5O.get_comment_by_name(loc_id,name,lapl_id)
```

Description

`comment = H5O.get_comment_by_name(loc_id,name,lapl_id)` retrieves a comment where a location id and name together specify the object. A link access property list can affect the outcome if a link is traversed to access the object.

Examples

```
fid = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
comment = H5O.get_comment_by_name(fid,'g4/world','H5P_DEFAULT');
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.set_comment` | `H5O.set_comment_by_name`

H5O.get_info

Object metadata

Syntax

```
info = H5O.get_info(obj_id)
```

Description

`info = H5O.get_info(obj_id)` retrieves the metadata for an object specified by `obj_id`. For details about the object metadata, please refer to the HDF5 documentation.

Examples

Determine the number of attributes for a dataset.

```
fid = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
dsetId = H5D.open(fid,'/g1/g1.1/dset1.1.1');
info = H5O.get_info(dsetId);
info.num_attrs
```

Determine the type of objects in the root group.

```
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
root_info = H5G.get_info(gid);
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
for j = 0:root_info.nlinks-1
   obj_id = H5O.open_by_idx(fid,'/',idx_type,order,j,plist);
   obj_info = H5O.get_info(obj_id);
   switch(obj_info.type)
       case H5ML.get_constant_value('H5G_LINK')
           fprintf('Object #%d is a link.\n',j);
       case H5ML.get_constant_value('H5G_GROUP')
           fprintf('Object #%d is a group.\n',j);
```

```
        case H5ML.get_constant_value('H5G_DATASET')
            fprintf('Object #%d is a dataset.\n',j);
        case H5ML.get_constant_value('H5G_TYPE')
            fprintf('Object #%d is a named datatype.\n',j);
    end
    H5O.close(obj_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5D.open | H5F.open | H5G.open | H5T.open

H5O.link

Create hard link to specified object

Syntax

```
H5O.link(obj_id,new_loc_id,new_link_name,lcpl_id,lapl_id)
```

Description

`H5O.link(obj_id,new_loc_id,new_link_name,lcpl_id,lapl_id)` creates a hard link to an object, where `new_loc_id` and `new_link_name` specify the location. `lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

`H5O.link` is designed to add additional structure to an existing file so that, for example, an object can be shared among multiple groups.

Examples

Create a hard link from group `'/g2'` to the dataset `'/g1/ds1'`.

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
gid1 = H5G.create(fid,'/g1',plist_id);
space_id = H5S.create_simple(1,10,[]);
ds1 = H5D.create(gid1,'ds1','H5T_NATIVE_INT',space_id,plist_id);
gid2 = H5G.create(fid,'/g2',plist_id);
H5O.link(ds1,gid2,'ds2',plist_id,plist_id);
H5D.close(ds1);
H5S.close(space_id);
H5G.close(gid2); H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_hard` | `H5L.create_soft`

H5O.open

Open specified object

Syntax

```
obj_id = H5O.open(loc_id,relname,lapl_id)
```

Description

`obj_id = H5O.open(loc_id,relname,lapl_id)` opens an object specified by location identifier and relative path name. `lapl_id` is the link access property list associated with the link pointing to the object. If default link access properties are appropriate, this can be passed in as `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
obj_id = H5O.open(fid,'g3','H5P_DEFAULT');
H5O.close(obj_id);
H5F.close(fid);
```

See Also

`H5O.close` | `H5O.open_by_idx`

H5O.open_by_idx

Open object specified by index

Syntax

```
obj_id = H5O.open_by_idx(loc_id, group_name, idx_type, order, n,
lapl_id)
```

Description

`obj_id = H5O.open_by_idx(loc_id, group_name, idx_type, order, n, lapl_id)` opens the `n`-th object in the group specified by `loc_id` and `group_name`. `loc_id` specifies a file or group. `group_name` specifies the group relative to `loc_id` in which the object can be found.

Two parameters are used to establish the iteration: `index_type` and `order`. `index_type` specifies the type of index by which objects are ordered. Valid values include the following:

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the order in which the links are to be referenced for the purposes of this function. Valid values include the following:

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

`n` specifies the zero-based position of the object within the index. `lapl_id` specifies the link access property list to be used in accessing the object.

Examples

```
fid = H5F.open('example.h5');
```

```
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
obj_id = H5O.open_by_idx(fid,'g3',idx_type,order,0,'H5P_DEFAULT');
H5O.close(obj_id);
H5F.close(fid);
```

See Also

H5O.close | H5O.open

H5O.set_comment

Set comment for object specified by object identifier

Syntax

```
H5O.set_comment(obj_id,comment)
```

Description

`H5O.set_comment(obj_id,comment)` sets a comment for the object specified by `obj_id`.

Examples

```
plist = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,plist);
gid = H5G.create(fid,'/g1',plist);
H5O.set_comment(gid,'This is a group comment.');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.get_comment_by_name` | `H5O.set_comment_by_name`

H5O.set_comment_by_name

Set comment for object specified by location and object name

Syntax

```
H5O.set_comment_by_name(loc_id,rel_name,comment,lapl_id)
```

Description

`H5O.set_comment_by_name(loc_id,rel_name,comment,lapl_id)` sets a comment for an object specified by a location ID and a relative name. `lapl_id` is a link access property list identifier that can affect the outcome if links are traversed.

Examples

```
plist = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,plist);
gid = H5G.create(fid,'/g1',plist);
H5O.set_comment_by_name(fid,'g1','This is a group comment.',plist);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.get_comment_by_name` | `H5O.set_comment`

H5O.visit

Visit objects specified by object identifier

Syntax

```
[status,opdata_out] =
H5O.visit(obj_id,index_type,order,iter_func,opdata_in)
```

Description

`[status,opdata_out] = H5O.visit(obj_id,index_type,order,iter_func,opdata_in)` is a recursive iteration function to visit the object `object_id` and, if `object_id` is a group, all objects in and below it in an HDF5 file. This provides a mechanism for an application to perform a common set of operations across all of those objects or a dynamically selected subset.

If `object_id` is a group identifier, that group serves as the root of a recursive iteration. If `object_id` is a file identifier, that file's root group serves as the root of the recursive iteration. If `object_id` is any other type of object, such as a dataset or named data type, there is no iteration.

Two parameters are used to establish the iteration: `index_type` and `order`. The `index_type` parameter specifies the index used. If the links in a group have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.) `order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

'H5_ITER_INC'	Increasing order
'H5_ITER_DEC'	Decreasing order
'H5_ITER_NATIVE'	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

See Also

`H5O.visit_by_name`

H5O.visit_by_name

Visit objects specified by location and object name

Syntax

```
[status,opdata_out] =
H5O.visit_by_name(loc_id,obj_name,index_type,order,iter_func,opdata_in,lapl_id
```

Description

`[status,opdata_out] = H5O.visit_by_name(loc_id,obj_name,index_type,order,iter_func,opdata_in,lapl_id` specifies the object by the pairing of the location identifier and object name. `loc_id` specifies a file or an object in a file and `obj_name` specifies an object in the file with either an absolute name or relative to `loc_id`. A link access property list can affect the outcome if links are involved.

Two parameters are used to establish the iteration: `index_type` and `order`. The `index_type` parameter specifies the index to be used. If the links in a group have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin; if the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.) `order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order

`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`lapl_id` is a link access property list. When default link access properties are acceptable, `'H5P_DEFAULT'` can be used.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

See Also

`H5O.visit`

H5P.close

Close property list

Syntax

```
H5P.close(plist_id)
```

Description

`H5P.close(plist_id)` terminates access to the property list specified by `plist_id`.

See Also

`H5P.create`

H5P.copy

Copy of property list

Syntax

```
plist_copy = H5P.copy(plist_id)
```

Description

`plist_copy = H5P.copy(plist_id)` returns a copy of the property list specified by `plist_id`.

Examples

Make a copy of the file creation property list for `'example.h5'`.

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
fcpl2 = H5P.copy(fcpl);
```

H5P.create

Create new property list

Syntax

```
plist = H5P.create(class_id)
```

Description

`plist = H5P.create(class_id)` creates a new property list as an instance of the property list class specified by `class_id`. The `class_id` input can be one of the following strings or the corresponding constant value.

```
'H5P_ATTRIBUTE_CREATE'
'H5P_DATASET_ACCESS'
'H5P_DATASET_CREATE'
'H5P_DATASET_XFER'
'H5P_DATATYPE_CREATE'
'H5P_DATATYPE_ACCESS'
'H5P_FILE_MOUNT'
'H5P_FILE_CREATE'
'H5P_FILE_ACCESS'
'H5P_GROUP_CREATE'
'H5P_GROUP_ACCESS'
'H5P_LINK_CREATE'
'H5P_LINK_ACCESS'
'H5P_OBJECT_COPY'
'H5P_OBJECT_CREATE'
'H5P_STRING_CREATE'
```

`class_id` can also be an instance of a property list class.

Examples

```
fapl = H5P.create('H5P_FILE_ACCESS');
fid = H5F.open('example.h5','H5F_ACC_RDONLY',fapl);
```

See Also

H5ML.get_constant_value | H5P.close | H5P.get_class

H5P.get_class

Property list class

Syntax

```
plist_class = H5P.get_class(plist_id)
```

Description

`plist_class = H5P.get_class(plist_id)` returns the property list class for the property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
pclass = H5P.get_class(fcpl);
name = H5P.get_class_name(pclass);
```

See Also

`H5P.get_class_name`

H5P.close_class

Close property list class

Syntax

```
H5P.close_class(class)
```

Description

`H5P.close_class(class)` closes the property list class specified by `pclass_id`.

H5P.equal

Determine equality of property lists

Syntax

```
value = H5P.equal(plist1_id, plist2_id)
```

Description

`value = H5P.equal(plist1_id, plist2_id)` returns a positive number if the two property lists specified are equal, and zero if they are not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
fcpl = H5F.get_create_plist(fid);
if H5P.equal(fapl,fcpl)
    fprintf('property lists are equal\n');
else
    fprintf('property lists are not equal\n');
end
```

H5P.exist

Determine if specified property exists in property list

Syntax

```
value = H5P.exist(prop_id, name)
```

Description

`value = H5P.exist(prop_id, name)` returns a positive value if the property specified by the text string `name` exists within the property list or class specified by `prop_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
if H5P.exist(fapl,'sieve_buf_size')
    fprintf('sieve buffer size property exists\n');
else
    fprintf('sieve buffer size property does not exist\n');
end
```

H5P.get

Value of specified property in property list

Syntax

```
value = H5P.get(plist_id, name)
```

Description

`value = H5P.get(plist_id, name)` retrieves a copy of the value of the property specified by the text string `name` in the property list specified by `plist_id`. The `H5P.get` function returns the property as an array of `uint8` values. You might need to cast the value to an appropriate data type to get a meaningful result.

Examples

```
plist = H5P.create('H5P_FILE_ACCESS');
val = H5P.get(plist, 'rdcc_w0');
rdcc_w0 = typecast(val,'double');
```

It is recommended to use alternative functions like `H5P.get_chunk`, `H5P.get_layout`, `H5P.get_size` etc., where available, to get values for the common property names.

See Also

`H5P.set` | `typecast`

H5P.get_class_name

Name of property list class

Syntax

```
classname = H5P.get_class_name(pclass_id)
```

Description

`classname = H5P.get_class_name(pclass_id)` retrieves the name of the generic property list class. `classname` is a text string. If no class is found, the empty string is returned.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
pclass = H5P.get_class(fcpl);
name = H5P.get_class_name(pclass);
```

See Also

`H5P.get_class`

H5P.get_class_parent

Identifier for parent class

Syntax

```
pclass_obj_id = H5P.get_class_parent(pclass_id)
```

Description

`pclass_obj_id = H5P.get_class_parent(pclass_id)` returns an identifier to the parent class object of the property class specified by `pclass_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
fcpl_class = H5P.get_class(fcpl);
parent_class = H5P.get_class_parent(fcpl_class);
name = H5P.get_class_name(parent_class);
```

See Also

`H5P.get_class` | `H5P.get_class_name`

H5P.get_nprops

Query number of properties in property list or class

Syntax

```
nprops = H5P.get_nprops(id)
```

Description

`nprops = H5P.get_nprops(id)` returns the number of properties in the property list or class specified by `id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
nprops = H5P.get_nprops(fcpl);
```

H5P.get_size

Query size of property value in bytes

Syntax

```
sz = H5P.get_size(id,name)
```

Description

`sz = H5P.get_size(id,name)` returns the size, in bytes, of the property specified by the text string `name` in the property list or property class specified by `id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_size(fapl,'sieve_buf_size');
```

H5P.isa_class

Determine if property list is member of class

Syntax

```
output = H5P.isa_class(plist_id, pclass_id)
```

Description

`output = H5P.isa_class(plist_id, pclass_id)` returns a positive number if the property list specified by `plist_id` is a member of the class specified by `pclass_id`, zero if it is not, and a negative value to indicate an error.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
if H5P.isa_class(fcpl,'H5P_FILE_ACCESS')
    fprintf('fcpl is a file access property list\n');
else
    fprintf('fcpl is not a file access property list\n');
end
```

See Also

`H5P.get_class`

H5P.iterate

Iterate over properties in property list

Syntax

```
[output,idx_out] = H5P.iterate(id,idx_in,iter_func)
```

Description

`[output,idx_out] = H5P.iterate(id,idx_in,iter_func)` executes the operation `iter_func` on each property in the property object specified in `id`. The `id` input can be a property list or a property class. `idx_in` specifies the index of the next property to be processed. `output` is the value returned by the last call to `iter_func`. `idx_out` is the index of the last property processed. `iter_func` is a function handle.

The iterator function must have the following signature:

```
status = iter_func(id,prop_name)
```

`id` still identifies the property object passed into `H5P.iterate`, but `name` identifies the name of the current property.

H5P.set

Set property list value

Syntax

```
H5P.set(plist_id,name,value)
```

Description

`H5P.set(plist_id,name,value)` sets the value of the property specified by the text string `name` in the property list specified by `plist_id` to the value specified in `value`. The datatype of `value` must be `uint8`.

Examples

```
plist = H5P.create('H5P_FILE_ACCESS');
H5P.set(plist, 'rdcc_w0', typecast(0.8, 'uint8'));
```

It is recommended to use alternative functions like `H5P.set_chunk`, `H5P.set_layout`, `H5P.set_size`, etc., where available, to set values for the common property names.

See Also

`typecast`

H5P.get_btree_ratios

B-tree split ratios

Syntax

```
[left,middle,right] = H5P.get_btree_ratios(plist_id)
```

Description

`[left,middle,right] = H5P.get_btree_ratios(plist_id)` returns the B-tree split ratios for the dataset transfer property list specified by `plist_id`. The `left` output specifies the B-tree split ratio for left-most nodes. `right` corresponds to the right-most nodes and lone nodes, and `middle` corresponds to all other nodes.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
[left,middle,right] = H5P.get_btree_ratios(dxpl);
```

See Also

`H5P.set_btree_ratios`

H5P.get_chunk_cache

Raw data chunk cache parameters

Syntax

```
[rdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl_id)
```

Description

`[rdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl_id)` retrieves the number of chunk slots in the raw data chunk cache hash table (`rdcc_nslots`), the maximum possible number of bytes in the raw data chunk cache (`rdcc_nbytes`), and the preemption policy value (`rdcc_w0`) on a dataset access property list.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen3D');
dapl = H5D.get_access_plist(dset_id);
[rrdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_chunk_cache`

H5P.get_dxpl_multi

Data access property lists for multiple files

Note: `H5P.get_dxpl_multi` has been removed.

Syntax

```
memb_dxpl = H5P.get_dxpl_multi(dxpl_id)
```

Description

`memb_dxpl = H5P.get_dxpl_multi(dxpl_id)` returns an array of data access property lists for the multifile data transfer property list specified by `dxpl_id`.

H5P.get_edc_check

Determine if error detection is enabled

Syntax

```
check = H5P.get_edc_check(plist_id)
```

Description

`check = H5P.get_edc_check(plist_id)` queries the dataset transfer property list, specified by `plist_id`, to determine whether error detection is enabled for data read operations. Returns either `H5Z_ENABLE_EDC` or `H5Z_DISABLE_EDC`.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
check = H5P.get_edc_check(dxpl);
switch(check)
    case H5ML.get_constant_value('H5Z_ENABLE_EDC')
        fprintf('error detection enabled\n');
    case H5ML.get_constant_value('H5Z_DISABLE_EDC');
        fprintf('error detection disabled\n');
end
```

See Also

`H5P.set_edc_check`

H5P.get_hyper_vector_size

Number of I/O vectors

Syntax

```
sz = H5P.get_hyper_vector_size(dxpl_id)
```

Description

`sz = H5P.get_hyper_vector_size(dxpl_id)` returns the number of I/O vectors to be read/written in hyperslab I/O.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
sz = H5P.get_hyper_vector_size(dxpl);
```

See Also

`H5P.set_hyper_vector_size`

H5P.set_btree_ratios

Set B-tree split ratios for dataset transfer

Syntax

```
H5P.set_btree_ratios(plist_id,left,middle,right)
```

Description

`H5P.set_btree_ratios(plist_id,left,middle,right)` sets the B-tree split ratios for the dataset transfer property list specified by `plist_id`. The `left` argument specifies the B-tree split ratio for left-most nodes. `right` specifies the B-tree split ratio for right-most nodes and lone nodes. `middle` specifies the B-tree split ratio for all other nodes.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_btree_ratios(dxpl, 0.2, 0.6, 0.95);
```

See Also

`H5P.get_btree_ratios`

H5P.set_chunk_cache

Set raw data chunk cache parameters

Syntax

```
H5P.set_chunk_cache(dapl_id, rdcc_nslots, rdcc_nbytes, rdcc_w0)
```

Description

`H5P.set_chunk_cache(dapl_id, rdcc_nslots, rdcc_nbytes, rdcc_w0)` sets the number of elements (`rdcc_nslots`), the total number of bytes (`rdcc_nbytes`), and the preemption policy value (`rdcc_w0`) in the raw data chunk cache.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen3D');
dapl = H5D.get_access_plist(dset_id);
H5P.set_chunk_cache(dapl,500,1e6,0.7);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_chunk_cache`

H5P.set_dxpl_multi

Set data transfer property list for multifile driver

Note: `H5P.set_dxpl_multi` has been removed.

Syntax

```
H5P.set_dxpl_multi(dxpl_id, memb_dxpl)
```

Description

`H5P.set_dxpl_multi(dxpl_id, memb_dxpl)` sets the data transfer property list `dxpl_id` to use the multifile driver. `memb_dxpl` is an array of data access property lists.

H5P.set_edc_check

Enable error detection for dataset transfer

Syntax

```
H5P.set_edc_check(plist_id,check)
```

Description

`H5P.set_edc_check(plist_id,check)` sets the dataset transfer property list specified by `plist_id` to enable or disable error detection when reading data. `check` can have the value `H5Z_ENABLE_EDC` or `H5Z_DISABLE_EDC`.

Examples

Disable error detection for a default dataset transfer property list.

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_edc_check(dxpl,'H5Z_DISABLE_EDC');
```

See Also

`H5P.get_edc_check`

H5P.set_hyper_vector_size

Set number of I/O vectors for hyperslab I/O

Syntax

```
H5P.set_hyper_vector_size(dxpl_id,size)
```

Description

`H5P.set_hyper_vector_size(dxpl_id,size)` sets the number of I/O vectors to be accumulated in memory before being issued to the lower levels of the HDF5 library for reading or writing the actual data. `dxpl_id` is a dataset transfer property list identifier. `size` specifies the number of I/O vectors to accumulate in memory for I/O operations.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_hyper_vector_size(dxpl,2048);
```

See Also

`H5P.get_hyper_vector_size`

H5P.all_filters_avail

Determine availability of all filters

Syntax

```
value = H5P.all_filters_avail(dcpl_id)
```

Description

`value = H5P.all_filters_avail(dcpl_id)` returns a positive value if all of the filters set in the dataset creation property list `dcpl_id` are currently available, and zero if they are not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
dcpl = H5D.get_create_plist(dset_id);
if H5P.all_filters_avail(dcpl)
    fprintf('all filters available\n');
else
    fprintf('all filters not available\n');
end
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_filter`

H5P.fill_value_defined

Determine if fill value is defined

Syntax

```
fvstatus = H5P.fill_value_defined(plist_id)
```

Description

`fvstatus = H5P.fill_value_defined(plist_id)` determines whether a fill value is defined in the dataset creation property list `plist_id`. The `fvstatus` output can have any of the following values: `H5D_FILL_VALUE_UNDEFINED`, `H5D_FILL_VALUE_DEFAULT`, or `H5D_FILL_VALUE_USER_DEFINED`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
dcpl = H5D.get_create_plist(dset_id);
fvstatus = H5P.fill_value_defined(dcpl);
switch(fvstatus)
    case H5ML.get_constant_value('H5D_FILL_VALUE_UNDEFINED')
        fprintf('fill value undefined\n');
    case H5ML.get_constant_value('H5D_FILL_VALUE_DEFAULT')
        fprintf('fill value set to default\n');
    case H5ML.get_constant_value('H5D_FILL_VALUE_USER_DEFINED')
        fprintf('fill value is user defined\n');
end
```

See Also

`H5P.get_fill_value` | `H5P.set_fill_value`

H5P.get_alloc_time

Return timing of storage space allocation

Syntax

```
alloc_time = H5P.get_alloc_time(plist_id)
```

Description

`alloc_time = H5P.get_alloc_time(plist_id)` retrieves the timing for storage space allocation from the dataset creation property list specified by `plist_id`. The `alloc_time` output can have any of the following values: `H5D_ALLOC_TIME_DEFAULT`, `H5D_ALLOC_TIME_EARLY`, `H5D_ALLOC_TIME_INCR`, or `H5D_ALLOC_TIME_LATE`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
alloc_time = H5P.get_alloc_time(dcpl);
switch(alloc_time)
    case H5ML.get_constant_value('H5D_ALLOC_TIME_DEFAULT')
        fprintf('allocation time is default\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_EARLY')
        fprintf('allocation time is dataset creation time\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_INCR')
        fprintf('allocation time is incremental\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_LATE')
        fprintf('allocation time is when data is first written\n');
end
```

H5P.get_chunk

Return size of chunks

Syntax

```
[rank,h5_chunk_dims] = H5P.get_chunk(plist_id)
```

Description

`[rank,h5_chunk_dims] = H5P.get_chunk(plist_id)` retrieves the size of chunks for the raw data of a chunked layout dataset for the dataset creation property list specified by `plist_id`.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_chunk_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g4/time');
dcpl = H5D.get_create_plist(dset_id);
[rank,chunk_dims] = H5P.get_chunk(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_chunk`

H5P.get_external

Return information about external file

Syntax

```
[name,offset,size] = H5P.get_external(plist_id,idx)
```

Description

`[name,offset,size] = H5P.get_external(plist_id,idx)` returns information about the external file specified by the dataset creation property list `plist_id`. The `idx` specifies the external file index, which is a number from zero to `N-1`, where `N` is the value returned by `H5P.get_external_count`. The `name` output returns the name of the external file (limited by 2048 characters). The `offset` output returns the location in bytes, from the beginning of the external file, where the data starts. The `size` output returns the size of the external data.

See Also

`H5P.get_external_count`

H5P.get_external_count

Return count of external files

Syntax

```
num_files = H5P.get_external_count(plist_id)
```

Description

`num_files = H5P.get_external_count(plist_id)` returns the number of external files for the dataset creation property list, `plist_id`.

See Also

`H5P.get_external`

H5P.get_fill_time

Return time when fill values are written to dataset

Syntax

```
fill_time = H5P.get_fill_time(plist_id)
```

Description

`fill_time = H5P.get_fill_time(plist_id)` returns the time when fill values are written to the dataset specified by the dataset creation property list `plist_id`. The `fill_time` output is one of the following values: `H5D_FILL_TIME_IFSET`, `H5D_FILL_TIME_ALLOC`, or `H5D_FILL_TIME_NEVER`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
fill_time = H5P.get_fill_time(dcpl);
switch(fill_time)
    case H5ML.get_constant_value('H5D_FILL_TIME_IFSET')
        fprintf('upon allocation if and only if fill value set by user\n');
    case H5ML.get_constant_value('H5D_FILL_TIME_ALLOC')
        fprintf('written when storage space is allocated\n');
    case H5ML.get_constant_value('H5D_FILL_TIME_NEVER')
        fprintf('fill values are never written\n');
end
```

See Also

`H5P.get_fill_time` | `H5P.set_fill_value`

H5P.get_fill_value

Return dataset fill value

Syntax

```
value = H5P.get_fill_value(plist_id,type_id)
```

Description

`value = H5P.get_fill_value(plist_id,type_id)` returns the dataset fill value defined in the dataset creation property list `plist_id`. The `type_id` input specifies the datatype of the returned fill value.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
type_id = H5T.copy('H5T_NATIVE_INT');
fill_value = H5P.get_fill_value(dcpl,type_id);
```

See Also

`H5P.get_fill_time` | `H5P.set_fill_time` | `H5P.set_fill_value`

H5P.get_filter

Return information about filter in pipeline

Syntax

```
[filter,flags,cd_values,name] = H5P.get_filter(plist_id,index)
[filter,flags,cd_values,name,filter_config] =
H5P.get_filter(plist_id,index)
```

Description

`[filter,flags,cd_values,name] = H5P.get_filter(plist_id,index)` returns information about the filter, specified by its filter index, in the filter pipeline, specified by the property list with which it is associated. This interface corresponds to the 1.6 version of `H5Pget_filter` in the HDF5 library.

`[filter,flags,cd_values,name,filter_config] = H5P.get_filter(plist_id,index)` returns information about the filter, specified by its filter index, in the filter pipeline, specified by the property list with which it is associated. It also returns information about the filter. Consult the HDF5 documentation for `H5Zget_filter_info` for information about `filter_config`. This interface corresponds to the 1.8 version of `H5Pget_filter` in the HDF5 library.

See Also

`H5P.get_filter_by_id` | `H5P.get_nfilters` | `H5P.modify_filter` | `H5P.remove_filter`

H5P.get_filter_by_id

Return information about specified filter

Syntax

```
[flags,cd_values,name,filter_config] =
H5P.get_filter_by_id(plist_id,idx)
```

Description

`[flags,cd_values,name,filter_config] = H5P.get_filter_by_id(plist_id,idx)` returns information about the filter specified by the filter id, `idx`.

See Also

`H5P.get_filter` | `H5P.get_nfilters` | `H5P.modify_filter` | `H5P.remove_filter`

H5P.get_layout

Determine layout of raw data for dataset

Syntax

```
layout = H5P.get_layout(dcpl)
```

Description

`layout = H5P.get_layout(dcpl)` returns the layout of the raw data for the dataset specified by the dataset creation property list, `dcpl`. Possible values are: `H5D_COMPACT`, `H5D_CONTIGUOUS`, or `H5D_CHUNKED`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
layout = H5P.get_layout(dcpl);
switch(layout)
    case H5ML.get_constant_value('H5D_COMPACT')
        fprintf('layout is compact\n');
    case H5ML.get_constant_value('H5D_CONTIGUOUS')
        fprintf('layout is contiguous\n');
    case H5ML.get_constant_value('H5D_CHUNKED')
        fprintf('layout is chunked\n');
end
```

See Also

`H5P.set_layout`

H5P.get_nfilters

Return number of filters in pipeline

Syntax

```
num_filters = H5P.get_nfilters(plist_id)
```

Description

`num_filters = H5P.get_nfilters(plist_id)` returns the number of filters defined in the filter pipeline associated with the dataset creation property list, `plist_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g4/world');
dcpl = H5D.get_create_plist(dset_id);
num_filters = H5P.get_nfilters(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_filter` | `H5P.get_filter_by_id` | `H5P.modify_filter` | `H5P.remove_filter`

H5P.modify_filter

Modify filter in pipeline

Syntax

```
H5P.modify_filter(plist_id,filter_id,flags,cd_values)
```

Description

`H5P.modify_filter(plist_id,filter_id,flags,cd_values)` modifies the specified filter in the filter pipeline. `plist_id` is a property list identifier. `flags` is a bit vector specifying certain general properties of the filter. `cd_values` specifies auxiliary data for the filter.

See Also

`H5P.get_filter` | `H5P.get_filter_by_id` | `H5P.get_nfilters` | `H5P.remove_filter`

H5P.remove_filter

Remove filter from property list

Syntax

```
H5P.remove_filter(plist_id,filter)
```

Description

`H5P.remove_filter(plist_id,filter)` removes the specified filter from the filter pipeline. `plist_id` is the dataset creation property list identifier.

See Also

`H5P.get_filter` | `H5P.get_filter_by_id` | `H5P.get_nfilters` | `H5P.modify_filter`

H5P.set_alloc_time

Set timing for storage space allocation

Syntax

```
H5P.set_alloc_time(plist_id,alloc_time)
```

Description

`H5P.set_alloc_time(plist_id,alloc_time)` sets the timing for the allocation of storage space for a dataset's raw data. `plist_id` is a dataset creation property list. `alloc_time` can have any of the following values: `H5D_ALLOC_TIME_DEFAULT`, `H5D_ALLOC_TIME_EARLY`, `H5D_ALLOC_TIME_INC`, or `H5D_ALLOC_TIME_LATE`.

Examples

Create a 1000x500 double precision dataset with late allocation time.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [1000 500];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
alloc_time = H5ML.get_constant_value('H5D_ALLOC_TIME_LATE');
H5P.set_alloc_time(dcpl,alloc_time);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_alloc_time`

H5P.set_chunk

Set chunk size

Syntax

```
H5P.set_chunk(plist_id,h5_chunk_dims)
```

Description

`H5P.set_chunk(plist_id,h5_chunk_dims)` sets the size of the chunks used to store a chunked layout dataset. `plist_id` is a dataset creation property list identifier. `h5_chunk_dims` is an array specifying the size, in dataset elements, of each chunk.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_chunk_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a two dimensional double precision dataset that has an initial size of [512 1024], but is also unlimited in both dimensions and has a chunk size of [512 1024].

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
unlimited = H5ML.get_constant_value('H5S_UNLIMITED');
dims = [512 1024];
h5_dims = fliplr(dims);
h5_maxdims = [unlimited unlimited];
space_id = H5S.create_simple(2,[1024 512],h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [512 1024];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
```

```
H5D.close(dset_id);
H5F.close(fid);
```

See Also

H5P.get_chunk

H5P.set_deflate

Set compression method and compression level

Syntax

```
H5P.set_deflate(plist_id,level)
```

Description

`H5P.set_deflate(plist_id,level)` sets the compression method for the dataset creation property list specified by `plist_id` to `H5D_COMPRESS_DEFLATE`. `level` specifies the compression level as a value from 0 and 9, inclusive. Lower values results in less compression.

Examples

Create a two dimensional double precision dataset that has an initial size of [512 1024], but is also unlimited in both dimensions and has a chunk size of [512 1024] and a compression level of 5.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
unlimited = H5ML.get_constant_value('H5S_UNLIMITED');
dims = [512 1024];
h5_dims = fliplr(dims);
h5_maxdims = [unlimited unlimited];
space_id = H5S.create_simple(2,[1024 512],h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [512 1024];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_deflate(dcpl,5);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

H5P.set_external

Add additional file to external file list

Syntax

```
H5P.set_external(plist_id,name,offset,nbytes)
```

Description

`H5P.set_external(plist_id,name,offset,nbytes)` adds the external file specified by `name` to the list of external files in the dataset creation property list, `plist_id`. The `offset` argument specifies the location, in bytes, where the data starts relative to the beginning of the file. `nbytes` is the number of bytes reserved in the file for the data. `nbytes` may also be given as `'H5F_UNLIMITED'`, in which case the external file may be of unlimited size.

Examples

Create a dataset with an unlimited size external file.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
H5P.set_external(dcpl,'myexternalfile.dat',0,'H5F_UNLIMITED');
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
data = rand(dims);
dxpl = 'H5P_DEFAULT';
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',dxpl,data);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5ML.get_constant_value` | `H5P.get_external`

H5P.set_fill_time

Set time when fill values are written to dataset

Syntax

```
H5P.set_fill_time(plist_id,fill_time)
```

Description

`H5P.set_fill_time(plist_id,fill_time)` sets the timing for writing fill values to a dataset in the dataset creation property list `plist_id`. The timing can be specified by one of the following values: `H5D_FILL_TIME_IFSET`, `H5D_FILL_TIME_ALLOC`, or `H5D_FILL_TIME_NEVER`.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
fill_time = H5ML.get_constant_value('H5D_FILL_TIME_ALLOC');
H5P.set_fill_time(dcpl,fill_time);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_fill_time` | `H5P.get_fill_value` | `H5P.set_fill_value`

H5P.set_fill_value

Set fill value for dataset creation property list

Syntax

```
H5P.set_fill_value(plist_id,type_id,value)
```

Description

`H5P.set_fill_value(plist_id,type_id,value)` sets the fill value for a the dataset creation property list specified by `plist_id`. The `value` argument specifies the fill value. `type_id` specifies the datatype of the fill value. Setting `value` to an empty array indicates that the fill value is to be undefined.

Examples

Create a double precision dataset with a fill value of -999.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
fill_time = H5ML.get_constant_value('H5D_FILL_TIME_ALLOC');
H5P.set_fill_time(dcpl,fill_time);
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
H5P.set_fill_value(dcpl,type_id,-999);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

H5P.set_filter

Add filter to filter pipeline

Syntax

```
H5P.set_filter(plist_id,filter,flags,cd_values)
```

Description

`H5P.set_filter(plist_id,filter,flags,cd_values)` adds the specified filter and corresponding properties to the end of an output filter pipeline. `plist_id` is a property list identifier. `filter` is a filter identifier and should correspond to one of the following values:

```
H5P_FILTER_DEFLATE
H5P_FILTER_SHUFFLE
H5P_FILTER_FLETCHER32
```

`flags` is a bit vector specifying properties of the filter. `cd_values` is an array that contains auxiliary data for the filter.

See Also

`H5P.set_deflate` | `H5P.set_fletcher32` | `H5P.set_shuffle`

H5P.set_fletcher32

Set Fletcher32 checksum filter in dataset creation

Syntax

```
H5P.set_fletcher32(plist_id)
```

Description

`H5P.set_fletcher32(plist_id)` sets the Fletcher32 checksum filter in the dataset creation property list specified by `plist_id`. The dataset creation property list must also have chunking enabled.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [10 20];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_fletcher32(dcpl);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_deflate` | `H5P.set_shuffle`

H5P.set_layout

Set type of storage for dataset

Syntax

```
H5P.set_layout(dcpl,layout)
```

Description

`H5P.set_layout(dcpl,layout)` sets the type of storage used to store the raw data for the dataset creation property list, `dcpl`. The `layout` argument specifies the type of storage layout for raw data: `H5D_COMPACT`, `H5D_CONTIGUOUS`, or `H5D_CHUNKED`.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
layout = H5ML.get_constant_value('H5D_CONTIGUOUS');
H5P.set_layout(dcpl,layout);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_layout` | `H5P.set_chunk`

H5P.set_nbit

Set N-Bit filter

Syntax

```
H5P.set_nbit(plist_id)
```

Description

`H5P.set_nbit(plist_id)` sets the N-Bit filter, `H5Z_FILTER_NBIT`, in the dataset creation property list `plist_id`.

H5P.set_scaleoffset

Set Scale-Offset filter

Syntax

```
H5P.set_scaleoffset(plistId,scaleType,scaleFactor)
```

Description

`H5P.set_scaleoffset(plistId,scaleType,scaleFactor)` sets the Scale-Offset filter, `H5Z_FILTER_SCALEOFFSET`, for a dataset. For integer data types, the parameter `scaleType` should be set to the enumerated value `H5Z_SO_INT`. For floating-point data types, the `scaleType` should be the enumerated value `H5Z_SO_FLOAT_DSCALE`. Chunking must already be enabled on the dataset creation property list.

See Also

`H5P.set_chunk`

H5P.set_shuffle

Set shuffle filter

Syntax

```
H5P.set_shuffle(plist_id)
```

Description

`H5P.set_shuffle(plist_id)` sets the shuffle filter, `H5Z_FILTER_SHUFFLE`, in the dataset creation property list `plist_id`. Compression must be enabled on the dataset creation property list in order to use the shuffle filter, and best results are usually obtained when the shuffle filter is set immediately prior to setting the deflate filter.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [10 20];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_shuffle(dcpl);
H5P.set_deflate(dcpl,5);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_deflate`

H5P.get_alignment

Retrieve alignment properties

Syntax

```
[threshold,alignment] = H5P.get_alignment(plist_id)
```

Description

`[threshold,alignment] = H5P.get_alignment(plist_id)` retrieves the current settings for alignment properties from the file access property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
[threshold,alignment] = H5P.get_alignment(fapl);
H5P.close(fapl);
H5F.close(fid);
```

H5P.get_driver

Low-level file driver

Syntax

```
driver_id = H5P.get_driver(plist_id)
```

Description

`driver_id = H5P.get_driver(plist_id)` returns the identifier of the low-level file driver associated with the file access property list or data transfer property list specified by `plist_id`. See HDF5 documentation for a list of valid return values.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
driver_id = H5P.get_driver(fapl);
if ( driver_id == H5ML.get_constant_value('H5FD_SEC2') )
    fprintf('File driver is H5FD_SEC2.\n');
end
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5ML.get_constant_value`

H5P.get_family_offset

Offset for family file driver

Syntax

```
offset = H5P.get_family_offset(fapl_id)
```

Description

`offset = H5P.get_family_offset(fapl_id)` retrieves the value of offset from the file access property list, `fapl_id`. `offset` is the offset of the data in the HDF5 file that is stored on disk in the selected member file in a family of files.

See Also

`H5P.set_family_offset`

H5P.get_fapl_core

Information about core file driver properties

Syntax

```
[increment,backing_store] = H5P.get_fapl_core(fapl_id)
```

Description

`[increment,backing_store] = H5P.get_fapl_core(fapl_id)` queries the `H5FD_CORE` driver properties as set by `H5P.set_fapl_core`. The `fapl_id` argument specifies a file access property list. The return value `increment` specifies the size, in bytes, of memory increments. `backing_store` is a Boolean flag indicating whether to write the file contents to disk when the file is closed.

See Also

`H5P.set_fapl_core`

H5P.get_fapl_family

File access property list information

Syntax

```
[memb_size,memb_fapl_id] = H5P.get_fapl_family(fapl_id)
```

Description

`[memb_size,memb_fapl_id] = H5P.get_fapl_family(fapl_id)` returns the size in bytes of each file member and the identifier of the file access property list for use with the family driver specified by `fapl_id`.

See Also

`H5P.set_fapl_family`

H5P.get_fapl_multi

Information about multifile access property list

Syntax

```
[memb_map,memb_fapl,memb_name,memb_addr,relax] =
H5P.get_fapl_multi(fapl_id)
```

Description

`[memb_map,memb_fapl,memb_name,memb_addr,relax] = H5P.get_fapl_multi(fapl_id)` returns information about the multifile access property list specified by `fapl_id`. The `memb_map` output maps memory usage types to other memory usage types. `memb_fapl` is a property list for each memory usage type. `memb_name` is the name generator for names of member files. `relax` is a Boolean value that, when non-zero, allows read-only access to incomplete file sets.

See Also

`H5P.set_fapl_multi`

H5P.get_fclose_degree

File close degree

Syntax

```
degree = H5P.get_fclose_degree(fapl_id)
```

Description

`degree = H5P.get_fclose_degree(fapl_id)` returns the current setting of the file close degree property `fc_degree` in the file access property list specified by `fapl_id`. Possible return values are: `H5F_CLOSE_DEFAULT`, `H5F_CLOSE_WEAK`, `H5F_CLOSE_SEMI`, or `H5F_CLOSE_STRONG`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
degree = H5P.get_fclose_degree(fapl);
switch(degree)
    case H5ML.get_constant_value('H5F_CLOSE_DEFAULT')
        fprintf('file close degree is default\n');
    case H5ML.get_constant_value('H5F_CLOSE_WEAK')
        fprintf('file close degree is weak\n');
    case H5ML.get_constant_value('H5F_CLOSE_SEMI')
        fprintf('close degree is semi\n');
    case H5ML.get_constant_value('H5F_CLOSE_STRONG')
        fprintf('close degree is strong\n');
end
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_fclose_degree`

H5P.get_libver_bounds

Library version bounds settings

Syntax

```
[low,high] = H5P.get_libver_bounds(fapl_id)
```

Description

`[low,high] = H5P.get_libver_bounds(fapl_id)` gets bounds on library version bounds settings that control the format versions used when creating objects in the file with access property list `fapl_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
[low,high] = H5P.get_libver_bounds(fapl);
```

See Also

`H5F.get_access_plist` | `H5P.set_libver_bounds`

H5P.get_gc_references

Garbage collection references setting

Syntax

```
gc_ref = H5P.get_gc_references(fapl_id)
```

Description

`gc_ref = H5P.get_gc_references(fapl_id)` returns the current setting for the garbage collection references property from the file access property list specified by `fapl_id`. If `gc_ref` is `1`, garbage collection is on; if `0`, garbage collection is off.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
gc_ref = H5P.get_gc_references(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_gc_references`

H5P.get_mdc_config

Metadata cache configuration

Syntax

```
config_struct = H5P.get_mdc_config(plist_id)
```

Description

`config_struct = H5P.get_mdc_config(plist_id)` returns the current metadata cache configuration from the indicated file access property list.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
config = H5P.get_mdc_config(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_mdc_config`

H5P.get_meta_block_size

Metadata block size setting

Syntax

```
sz = H5P.get_meta_block_size(fapl_id)
```

Description

`sz = H5P.get_meta_block_size(fapl_id)` returns the current minimum size, in bytes, of new metadata block allocations.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_meta_block_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_meta_block_size`

H5P.get_multi_type

Type of data property for MULTI driver

Syntax

```
type = H5P.get_multi_type(fapl_id)
```

Description

`type = H5P.get_multi_type(fapl_id)` returns the type of data setting from the file access or data transfer property list, `fapl_id`.

This function should only be used with an HDF5 file written as a set of files with the MULTI file driver.

See Also

`H5P.set_multi_type`

H5P.get_sieve_buf_size

Maximum data sieve buffer size

Syntax

```
sz = H5P.get_sieve_buf_size(fapl_id)
```

Description

`sz = H5P.get_sieve_buf_size(fapl_id)` returns the current maximum size of the data sieve buffer.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_sieve_buf_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_sieve_buf_size`

H5P.get_small_data_block_size

Small data block size setting

Syntax

```
sz = H5P.get_small_data_block_size(fapl_id)
```

Description

`sz = H5P.get_small_data_block_size(fapl_id)` returns the current setting for the size of the small data block. `fapl_id` is a file access property list identifier.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_small_data_block_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_small_data_block_size`

H5P.set_alignment

Set alignment properties for file access property list

Syntax

```
H5P.set_alignment(fapl_id,threshold,alignment)
```

Description

`H5P.set_alignment(fapl_id,threshold,alignment)` sets the alignment properties of the file access property list specified by `fapl_id` so that any file object greater than or equal in size to threshold (in bytes) is aligned on an address which is a multiple of alignment.

In most cases the default values of threshold and alignment result in the best performance.

See Also

`H5P.get_alignment`

H5P.set_family_offset

Set offset property for family of files

Syntax

```
H5P.set_family_offset(fapl_id,offset)
```

Description

`H5P.set_family_offset(fapl_id,offset)` sets offset property in the file access property list specified by `fapl_id` for low-level access to a file in a family of files. `offset` identifies a user-determined location from the beginning of the HDF5 file in bytes.

See Also

`H5P.get_family_offset`

H5P.set_fapl_core

Modify file access to use H5FD_CORE driver

Syntax

```
H5P.set_fapl_core(fapl_id,increment,backing_store)
```

Description

`H5P.set_fapl_core(fapl_id,increment,backing_store)` modifies the file access property list to use the H5FD_CORE driver. `increment` specifies the increment by which allocated memory is to be increased each time more memory is required. `backing_store` is a Boolean flag that, when non-zero, indicates the file contents should be written to disk when the file is closed.

Examples

Create a file image in memory only.

```
plist = 'H5P_DEFAULT';
ndatasets = 20;
block_size = 1024*1024;
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_core(fapl,2^16,false);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,fapl);
space_id = H5S.create_simple(1, block_size, []);
type_id = H5T.copy('H5T_IEEE_F64LE');
data = zeros(block_size,1);
for j = 1:ndatasets
 dsname = sprintf( 'dset%02d', j);
 fprintf( 'Writing dataset %s...\n',dsname);
 dsid = H5D.create(fid,dsname,type_id,space_id,'H5P_DEFAULT');
 H5D.write(dsid,'H5ML_DEFAULT',space_id,space_id,plist,data);
 H5D.close(dsid);
end
H5P.close(fapl);
H5S.close(space_id);
```

```
H5T.close(type_id);
H5F.close(fid);
dir('myfile.h5');
```

See Also

H5P.get_fapl_core

H5P.set_fapl_family

Set file access to use family driver

Syntax

```
H5P.set_fapl_family(fapl_id,memb_size,memb_fapl_id)
```

Description

`H5P.set_fapl_family(fapl_id,memb_size,memb_fapl_id)` sets the file access property list, specified by `fapl_id`, to use the family driver. `memb_size` is the size in bytes of each file member. `memb_fapl_id` is the identifier of the file access property list to be used for each family member.

Examples

```
plist = 'H5P_DEFAULT';
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_family(fapl, 8192, plist);
fid = H5F.create('family%d.h5','H5F_ACC_TRUNC','H5P_DEFAULT',fapl);
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [50 25];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,[]);
dset_id = H5D.create(fid,'DS',type_id,space_id,plist)
data = reshape(1:prod(dims),dims);
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5P.close(fapl);
H5T.close(type_id);
H5S.close(space_id);
H5D.close(dset_id);
dir('*.h5');
h5disp('family%d.h5');
```

See Also

`H5P.get_fapl_family`

H5P.set_fapl_log

Set use of logging driver

Syntax

```
H5P.set_fapl_log(fapl_id,logfile,flags,buf_size)
```

Description

`H5P.set_fapl_log(fapl_id,logfile,flags,buf_size)` modifies the file access property list, `fapl_id`, to use the logging driver H5FD_LOG. `logfile` is the name of the file in which the logging entries are to be recorded. `flags` is a bit mask that specifies the types of activity to log. See the HDF5 documentation for a list of available flag settings. `buf_size` specifies the size of the logging buffer.

H5P.set_fapl_multi

Set use of multifile driver

Syntax

```
H5P.set_fapl_multi(fapl_id,relax)
H5P.set_fapl_multi(fapl_id,memb_map,memb_fapl,memb_name,memb_addr,relax)
```

Description

`H5P.set_fapl_multi(fapl_id,relax)` sets the file access property list, `fapl_id`, to access HDF5 files created with the multi-driver with default values provided by the HDF5 library. `relax` is a Boolean value that allows read-only access to incomplete file sets when set to `1`.

`H5P.set_fapl_multi(fapl_id,memb_map,memb_fapl,memb_name,memb_addr,relax)` sets the file access property list to use the multifile driver. `memb_map` maps memory usage types to other memory usage types. `memb_fapl` contains a property list for each memory usage type. `memb_name` is a name generator for names of member files. `memb_addr` specifies the offsets within the virtual address space at which each type of data storage begins.

See Also

`H5P.get_fapl_multi`

H5P.set_fapl_sec2

Set file access for sec2 driver

Syntax

```
H5P.set_fapl_sec2(fapl_id)
```

Description

`H5P.set_fapl_sec2(fapl_id)` modifies the file access property list, `fapl_id`, to use the H5FD_SEC2 driver.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_sec2(fapl);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

H5P.set_fapl_split

Set file access for emulation of split file driver

Syntax

```
H5P.set_fapl_split(fapl_id,meta_ext,meta_plist_id,raw_ext,raw_plist_id)
```

Description

`H5P.set_fapl_split(fapl_id,meta_ext,meta_plist_id,raw_ext,raw_plist_id)` is a compatibility function that enables the multi-file driver to emulate the split driver from HDF5 Releases 1.0 and 1.2. `meta_ext` is a text string that specifies the metadata filename extension. `meta_plist_id` is a file access property list identifier for the metadata file. `raw_ext` is a text string that specifies the raw data filename extension. `raw_plist_id` is the file access property list identifier for the raw data file.

H5P.set_fapl_stdio

Set file access for standard I/O driver

Syntax

```
H5P.set_fapl_stdio(fapl_id)
```

Description

`H5P.set_fapl_stdio(fapl_id)` modifies the file access property list, `fapl_id`, to use the standard I/O driver, H5FD_STDIO.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_stdio(fapl);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

H5P.set_fclose_degree

Set file access for file close degree

Syntax

```
H5P.set_fclose_degree(fapl_id,degree)
```

Description

`H5P.set_fclose_degree(fapl_id,degree)` sets the file close degree property in the file access property list, `fapl_id`, to the value specified by `degree`. The `degree` argument can have any of the following values:

`'H5F_CLOSE_WEAK'`

`'H5F_CLOSE_SEMI'`

`'H5F_CLOSE_STRONG'`

`'H5F_CLOSE_DEFAULT'`

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fclose_degree(fapl,'H5F_CLOSE_STRONG');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_fclose_degree`

H5P.set_gc_references

Set garbage collection references flag

Syntax

```
H5P.set_gc_references(fapl_id,gc_ref)
```

Description

`H5P.set_gc_references(fapl_id,gc_ref)` sets the flag for garbage collecting references for the file specified by the file access property list identifier, `fapl_id`. The `gc_ref` argument is a flag setting reference garbage collection to on (`1`) or off (`0`).

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_gc_references(fapl,1);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_gc_references`

H5P.set_libver_bounds

Set library version bounds for objects

Syntax

```
H5P.set_libver_bounds(fapl_id,low,high)
```

Description

`H5P.set_libver_bounds(fapl_id,low,high)` sets bounds on library versions, and indirectly format versions, to be used when creating objects in the file with access property list `fapl_id`. The `low` argument must be set to either of `'H5F_LIBVER_EARLIEST'`, `'H5F_LIBVER_18'` or `'H5F_LIBVER_LATEST'`. The `high` argument must be set to `'H5F_LIBVER_18'` or `'H5F_LIBVER_LATEST'`.

Examples

Create an HDF5 file where objects are created using the latest available format for each object.

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_libver_bounds(fapl,'H5F_LIBVER_LATEST','H5F_LIBVER_LATEST');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
```

See Also

`H5ML.get_constant_value` | `H5P.get_libver_bounds`

H5P.set_mdc_config

Set initial metadata cache configuration

Syntax

```
H5P.set_mdc_config(plist_id,config_struct)
```

Description

`H5P.set_mdc_config(plist_id,config_struct)` sets the initial metadata cache configuration in the indicated file access property list to the supplied values. Before using this function, you should retrieve the current configuration using `H5P.get_mdc_config`.

Many of the fields in the structure, `config_struct`, are intended to be used only in close consultation with the HDF5 Group itself.

See Also

`H5P.get_mdc_config`

H5P.set_meta_block_size

Set minimum metadata block size

Syntax

```
H5P.set_meta_block_size(fapl_id,size)
```

Description

`H5P.set_meta_block_size(fapl_id,size)` sets the minimum metadata block size for the file access property list specified by `fapl_id`. The `size` argument specifies minimum size, in bytes, of metadata block allocations.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_meta_block_size(fapl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_meta_block_size`

H5P.set_multi_type

Specify type of data accessed with MULTI driver

Syntax

```
H5P.set_multi_type(fapl_id,type)
```

Description

`H5P.set_multi_type(fapl_id,type)` sets the type of data property in the file access or data transfer property list `fapl_id`. The `type` argument can have any of the following values: `H5FD_MEM_SUPER`, `H5FD_MEM_BTREE`, `H5FD_MEM_DRAW`, `H5FD_MEM_GHEAP`, `H5FD_MEM_LHEAP`, or `H5FD_MEM_OHDR`.

See Also

`H5P.get_multi_type`

H5P.set_sieve_buf_size

Set maximum size of data sieve buffer

Syntax

```
H5P.set_sieve_buf_size(fapl_id,buffer_size)
```

Description

`H5P.set_sieve_buf_size(fapl_id,buffer_size)` sets `buffer_size`, the maximum size in bytes of the data sieve buffer, which is used by file drivers that are capable of using data sieving. `fapl_id` is a file access property list identifier.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_sieve_buf_size(fapl,131072);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_sieve_buf_size`

H5P.set_small_data_block_size

Set size of block reserved for small data

Syntax

```
H5P.set_small_data_block_size(fapl_id,size)
```

Description

`H5P.set_small_data_block_size(fapl_id,size)` sets the maximum size, in bytes, of a contiguous block reserved for small data. `fapl_id` is a file access property list identifier.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_small_data_block_size(fapl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_small_data_block_size`

H5P.get_istore_k

Return 1/2 rank of indexed storage B-tree

Syntax

```
ik = H5P.get_istore_k(plist_id)
```

Description

`ik = H5P.get_istore_k(plist_id)` returns the chunked storage B-tree 1/2 rank of the file creation property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
ik = H5P.get_istore_k(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_istore_k`

H5P.get_sizes

Return size of offsets and lengths

Syntax

```
[sizeof_addr,sizeof_size] = H5P.get_sizes(fcpl)
```

Description

`[sizeof_addr,sizeof_size] = H5P.get_sizes(fcpl)` returns the size of the offsets and lengths used in an HDF5 file. `fcpl` specifies a file creation property list.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[soaddr, sosize] = H5P.get_sizes(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_sizes`

H5P.get_sym_k

Return size of B-tree 1/2 rank and leaf node 1/2 size

Syntax

```
[ik,lk] = H5P.get_sym_k(plist_id)
```

Description

`[ik,lk] = H5P.get_sym_k(plist_id)` returns the size of the symbol table B-tree 1/2 rank, `ik`, and the symbol table leaf node 1/2 size, `lk`. The `plist_id` argument is a file creation property list identifier.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[ik, lk] = H5P.get_sym_k(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_sym_k`

H5P.get_userblock

Return size of user block

Syntax

```
sz = H5P.get_userblock(plist_id)
```

Description

`sz = H5P.get_userblock(plist_id)` returns the size of a user block in a file creation property list. `plist_id` is a property list identifier.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
sz = H5P.get_userblock(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_userblock`

H5P.get_version

Return version information for file creation property list

Syntax

```
[superblock,freelist,stab,shhdr] = H5P.get_version(fcpl)
```

Description

`[superblock,freelist,stab,shhdr] = H5P.get_version(fcpl)` returns the version of the super block, the global freelist, the symbol table, and the shared object header. Retrieving this information requires the file creation property list.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[super,freelist,stab,shhdr] = H5P.get_version(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

H5P.set_istore_k

Set size of parameter for indexing chunked datasets

Syntax

```
H5P.set_istore_k(plist_id,ik)
```

Description

`H5P.set_istore_k(plist_id,ik)` sets the size of the parameter used to control the B-trees for indexing chunked datasets for the file creation property list specified by `plist_id`. The `ik` argument is one half the rank of a tree that stores chunked raw data.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_istore_k(fcpl,64);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_istore_k`

H5P.set_sizes

Set byte size of offsets and lengths

Syntax

```
H5P.set_sizes(plist_id,sizeof_addr,sizeof_size)
```

Description

`H5P.set_sizes(plist_id,sizeof_addr,sizeof_size)` sets the byte size of the offsets and lengths used to address objects in an HDF5 file. `plist_id` is a file creation property list.

See Also

`H5P.get_sizes`

H5P.set_sym_k

Set size of parameters used to control symbol table nodes

Syntax

```
H5P.set_sym_k(plist_id,ik,lk)
```

Description

`H5P.set_sym_k(plist_id,ik,lk)` sets the size of parameters used to control the symbol table nodes for the file access property list, `plist_id`. The `ik` argument is one half the rank of a tree that stores a symbol table for a group. `lk` is one half of the number of symbols that can be stored in a symbol table node.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_sym_k(fcpl,32,8);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_sym_k`

H5P.set_userblock

Set user block size

Syntax

```
H5P.set_userblock(plist_id,size)
```

Description

`H5P.set_userblock(plist_id,size)` sets the user block size of the file creation property list, `plist_id`.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_userblock(fcpl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_userblock`

H5P.get_attr_creation_order

Return tracking order and indexing settings

Syntax

```
crt_order_flags = H5P.get_attr_creation_order(ocpl_id)
```

Description

`crt_order_flags = H5P.get_attr_creation_order(ocpl_id)` retrieves tracking and indexing settings for attribute creation order. If `crt_order_flags` is zero, then the attribute creation order is neither tracked or indexed. Otherwise the creation order flags should be one of the following constant values:

```
H5P_CRT_ORDER_TRACKED
H5P_CRT_ORDER_INDEXED
```

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
flags = H5P.get_attr_creation_order(dcpl);
switch ( flags )
    case 0
        fprintf('neither tracked nor indexed\n');
    case H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED')
        fprintf('tracked\n');
    case H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED')
        fprintf('indexed\n');
end
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5ML.get_constant_value` | `H5P.set_attr_creation_order`

H5P.get_attr_phase_change

Retrieve attribute phase change thresholds

Syntax

```
[max_compact,min_dense] = H5P.get_attr_phase_change(ocpl_id)
```

Description

`[max_compact,min_dense] = H5P.get_attr_phase_change(ocpl_id)` retrieves attribute phase change thresholds for the dataset or group with creation property list `ocpl_id`.

`max_compact` is the maximum number of attributes to be stored in compact storage (default is 8).

`min_dense` is the minimum number of attributes to be stored in dense storage (default is 6).

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
[max_compact,min_dense] = H5P.get_attr_phase_change(dcpl);
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_attr_phase_change`

H5P.get_copy_object

Return properties to be used when object is copied

Syntax

```
copy_options = H5P.get_copy_object(ocpl_id)
```

Description

`copy_options = H5P.get_copy_object(ocpl_id)` retrieves the properties currently specified in the object copy property list `ocpl_id`, which will be invoked when a new copy is made of an existing object.

Examples

```
ocpl = H5P.create('H5P_OBJECT_COPY');
options = H5P.get_copy_object(ocpl);
```

See Also

`H5P.set_copy_object`

H5P.set_attr_creation_order

Set tracking of attribute creation order

Syntax

```
H5P.set_attr_creation_order(gcplId,crt_order_flags)
```

Description

`H5P.set_attr_creation_order(gcplId,crt_order_flags)` sets tracking and indexing of attribute creation order. The creation order flags should be either `H5P_CRT_ORDER_TRACKED` or a bitwise-or of `H5P_CRT_ORDER_TRACKED` and `H5P_CRT_ORDER_INDEXED`.

The default behavior is that attribute creation order is neither tracked nor indexed.

Examples

```
dcpl = H5P.create('H5P_DATASET_CREATE');
order = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
H5P.set_attr_creation_order(dcpl,order);
```

See Also

`bitor` | `H5ML.get_constant_value` | `H5P.get_attr_creation_order`

H5P.set_attr_phase_change

Set attribute storage phase change thresholds

Syntax

```
H5P.set_attr_phase_change(ocpl_id,max_compact,min_dense)
```

Description

`H5P.set_attr_phase_change(ocpl_id,max_compact,min_dense)` sets attribute storage phase change thresholds for the group or dataset with creation order property list `ocpl_id`.

`max_compact` is the maximum number of attributes to be stored in compact storage (default is 8).

`min_dense` is the minimum number of attributes to be stored in dense storage (default is 6).

See Also

`H5P.get_attr_phase_change`

H5P.set_copy_object

Set properties to be used when objects are copied

Syntax

```
H5P.set_copy_object(ocp_plist_id,copy_options)
```

Description

`H5P.set_copy_object(ocp_plist_id,copy_options)` sets the properties in the object copy property list `ocp_plist_id` that will be invoked when a new copy is made of an existing object. `ocp_plist_id` is the object copy property list and specifies the properties governing the copying of the object.

Several flags, described below, are available for inclusion in the object copy property list:

`H5O_COPY_SHALLOW_HIERARCHY_FLAG`	Copy only immediate members of a group. Default behavior, without flag: Recursively copy all objects below the group.
`H5O_COPY_EXPAND_SOFT_LINK_FLAG`	Expand soft links into new objects. Default behavior, without flag: Keep soft links as they are.
`H5O_COPY_EXPAND_EXT_LINK_FLAG`	Expand external link into new objects. Default behavior, without flag: Keep external links as they are.
`H5O_COPY_EXPAND_REFERENCE_FLAG`	Copy objects that are pointed to by references. Default behavior, without flag: Update only the values of object references.
`H5O_COPY_WITHOUT_ATTR_FLAG`	Copy object without copying attributes. Default behavior, without flag: Copy object along with all its attributes.

Examples

```
ocp_plist_id = H5P.create ('H5P_OBJECT_COPY');
```

```
option1 = H5ML.get_constant_value('H5O_COPY_EXPAND_SOFT_LINK_FLAG');
option2 = H5ML.get_constant_value('H5O_COPY_EXPAND_REFERENCE_FLAG');
copy_options = bitor(option1,option2);
H5P.set_copy_object(ocp_plist_id, copy_options);
```

H5P.get_create_intermediate_group

Determine creation of intermediate groups

Syntax

```
bool = H5P.get_create_intermediate_group(lcpl_id)
```

Description

`bool = H5P.get_create_intermediate_group(lcpl_id)` determines whether the link creation property list `lcpl_id` is set to enable creating missing intermediate groups.

Examples

```
lcpl = H5P.create('H5P_LINK_CREATE');
if H5P.get_create_intermediate_group(lcpl)
    fprintf('set to enable creating intermediate groups\n');
else
    fprintf('not set to enable creating intermediate groups\n');
end
```

See Also

`H5P.set_create_intermediate_group`

H5P.get_link_creation_order

Query if link creation order is tracked

Syntax

```
crt_order_flags = H5P.get_link_creation_order(gcpl_id)
```

Description

`crt_order_flags = H5P.get_link_creation_order(gcpl_id)` queries whether link creation order is tracked or indexed in a group with creation property list identifier `gcpl_id`. The creation order flags should be one of the following constant values:

`H5P_CRT_ORDER_TRACKED`

`H5P_CRT_ORDER_INDEXED`

Examples

```
tracked = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
indexed = H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED');
gcpl = H5P.create('H5P_GROUP_CREATE');
order = H5P.get_link_creation_order(gcpl);
if bitand(order,tracked)
    fprintf('order is tracked\n');
end
if bitand(order,indexed)
    fprintf('order is indexed\n');
end
```

See Also

`bitand` | `H5ML.get_constant_value` | `H5P.set_link_creation_order`

H5P.get_link_phase_change

Query settings for conversion between groups

Syntax

```
[max_compact,min_dense] = H5P.get_link_phase_change(gcpl_id)
```

Description

`[max_compact,min_dense] = H5P.get_link_phase_change(gcpl_id)` retrieves the settings for conversion between compact and dense groups.

`max_compact` is the maximum number of links to store as header messages in the group header before converting the group to the dense format. Groups that are in the compact format and exceed this number of links are automatically converted to the dense format.

`min_dense` is the minimum number of links to store in the dense format. Groups which are in dense format and in which the number of links falls below this number are automatically converted back to the compact format.

Examples

```
gcpl = H5P.create('H5P_GROUP_CREATE');
[max_compact, min_dense] = H5P.get_link_phase_change(gcpl);
```

See Also

`H5P.set_link_phase_change`

H5P.set_create_intermediate_group

Set creation of intermediate groups

Syntax

```
H5P.set_create_intermediate_group(lcpl_id,flag)
```

Description

`H5P.set_create_intermediate_group(lcpl_id,flag)` specifies in the link creation property list `lcpl_id` whether to create missing intermediate groups.

Examples

Enable the creation of intermediate groups.

```
fid = H5F.create('myfile.h5');
lcpl = H5P.create('H5P_LINK_CREATE');
H5P.set_create_intermediate_group(lcpl,1);
gid = H5G.create(fid,'/a/b/c/d',lcpl,'H5P_DEFAULT','H5P_DEFAULT');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5P.get_create_intermediate_group`

H5P.set_link_creation_order

Set creation order tracking and indexing

Syntax

```
H5P.set_link_creation_order(gcplId,crt_order_flags)
```

Description

`H5P.set_link_creation_order(gcplId,crt_order_flags)` sets creation order tracking and indexing for links in the group with group creation property list `gcpl_id`.

The creation order flags should be one of the following constant values:

`H5P_CRT_ORDER_TRACKED`

`H5P_CRT_ORDER_INDEXED`

If only `H5P_CRT_ORDER_TRACKED` is set, HDF5 will track link creation order in any group created with the group creation property list `gcpl_id`. If both `H5P_CRT_ORDER_TRACKED` and `H5P_CRT_ORDER_INDEXED` are set, HDF5 will track link creation order in the group and index links on that property.

Examples

```
tracked = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
indexed = H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED');
order = bitor(tracked,indexed);
gcpl = H5P.create('H5P_GROUP_CREATE');
H5P.set_link_creation_order(gcpl,order);
```

See Also

`H5ML.get_constant_value` | `H5P.get_link_creation_order`

H5P.set_link_phase_change

Set parameters for group conversion

Syntax

```
H5P.set_link_phase_change(gcpl_id,max_compact,min_dense)
```

Description

`H5P.set_link_phase_change(gcpl_id,max_compact,min_dense)` sets the parameters for conversion between compact and dense groups.

`max_compact` is the maximum number of links to store as header messages in the group header before converting the group to the dense format. Groups that are in the compact format and exceed this number of links are automatically converted to the dense format.

`min_dense` is the minimum number of links to store in the dense format. Groups which are in dense format and in which the number of links falls below this number are automatically converted back to the compact format.

Examples

```
gcpl = H5P.create('H5P_GROUP_CREATE');
H5P.set_link_phase_change(gcpl,10,8);
```

See Also

`H5P.get_link_phase_change`

H5P.get_char_encoding

Return character encoding

Syntax

```
encoding = H5P.get_char_encoding(propertyList)
```

Description

`encoding = H5P.get_char_encoding(propertyList)` retrieves the character encoding used to encode strings or object names that are created with the property list `propertyList`. The values returned correspond to either `H5T_CSET_ASCII` or `H5T_CSET_UTF8`.

See Also

`H5ML.get_constant_value` | `H5P.set_char_encoding`

H5P.set_char_encoding

Set character encoding used to encode strings

Syntax

```
H5P.set_char_encoding(propList,encoding)
```

Description

`H5P.set_char_encoding(propList,encoding)` sets the character encoding used to encode strings or object names that are created with the property list `propList`. The values of `encoding` should either be `H5T_CSET_ASCII` or `H5T_CSET_UTF8`.

See Also

`H5ML.get_constant_value` | `H5P.get_char_encoding`

H5R.create

Create reference

Syntax

```
ref = H5R.create(loc_id,name,ref_type,space_id)
```

Description

`ref = H5R.create(loc_id,name,ref_type,space_id)` creates the reference, `ref`, of the type specified in `ref_type`, pointing to the object specified by name located at `loc_id`. The `ref_type` argument can be either `'H5R_OBJECT'`, or `'H5R_DATASET_REGION'`. The `space_id` argument should be `-1`, if `ref_type` is `'H5R_OBJECT'`.

Examples

Create a double-precision dataset and a reference dataset.

```
fid = H5F.create('myfile.h5');
type1_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [10 5];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space1_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset1_id = H5D.create(fid,'my_double',type1_id,space1_id,dcpl);
type2_id = 'H5T_STD_REF_OBJ';
space2_id = H5S.create('H5S_SCALAR');
dset2_id = H5D.create(fid,'my_ref',type2_id,space2_id,dcpl);
ref_data = H5R.create(fid,'my_double','H5R_OBJECT',-1);
dxpl = 'H5P_DEFAULT';
H5D.write(dset2_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',dxpl,ref_data);
H5D.close(dset1_id);
H5D.close(dset2_id);
H5F.close(fid);
```

See Also

H5D.create

H5R.dereference

Open object specified by reference

Syntax

```
output = H5R.dereference(dataset,ref_type,ref)
```

Description

`output = H5R.dereference(dataset,ref_type,ref)` returns an identifier to the object specified by `ref` in the dataset specified by `dataset`.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
deref_dset_id = H5R.dereference(dset_id,'H5R_OBJECT',ref_data(:,1));
H5D.close(dset_id);
H5D.close(deref_dset_id);
H5F.close(fid);
```

See Also

`H5I.get_name` | `H5R.create`

H5R.get_name

Name of referenced object

Syntax

```
name = H5R.get_name(loc_id,ref_type,ref)
```

Description

`name = H5R.get_name(loc_id,ref_type,ref)` retrieves the name for the object identified by `ref`. The `loc_id` argument is the identifier for the dataset containing the reference or for the group containing that dataset. `ref_type` specifies the type of the reference `ref`. Valid values for `ref_type` are `'H5R_OBJECT'` or `'H5R_DATASET_REGION'`.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
name = H5R.get_name(dset_id,'H5R_OBJECT',ref_data(:,1));
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5I.get_name`

H5R.get_obj_type

Type of referenced object

Syntax

```
obj_type = H5R.get_obj_type(id,ref_type,ref)
```

Description

`obj_type = H5R.get_obj_type(id,ref_type,ref)` returns the type of object that an object reference points to. Valid values for `ref_type` are: `H5R_OBJECT` or `H5R_DATASET_REGION`. Valid return values correspond to the following values.

`'H5O_TYPE_GROUP'`	Object is a group.
`'H5O_TYPE_DATASET'`	Object is a dataset.
`'H5O_TYPE_NAMED_DATATYPE'`	Object is a named datatype.

This function corresponds to the 1.8 interface version of `H5Rget_obj_type` in the HDF5 library C API.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
obj_type = H5R.get_obj_type(fid,'H5R_OBJECT',ref_data(:,1));
switch(obj_type)
    case H5ML.get_constant_value('H5O_TYPE_GROUP')
        fprintf('group\n');
    case H5ML.get_constant_value('H5O_TYPE_DATASET')
        fprintf('dataset\n');
    case H5ML.get_constant_value('H5O_TYPE_NAMED_DATATYPE')
        fprintf('named datatype\n');
```

```
end
H5D.close(dset_id);
H5F.close(fid);
```

See Also

H5ML.get_constant_value

H5R.get_region

Copy of data space of specified region

Syntax

```
space_id = H5R.get_region(dataset,ref_type,ref)
```

Description

`space_id = H5R.get_region(dataset,ref_type,ref)` returns a data space with the specified region selected. `dataset` is used to identify the file containing the referenced region and can be any identifier for any object in the file.

Examples

```
space = 'H5S_ALL';
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/region_reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_DSETREG',space,space,plist);
space_id = H5R.get_region(fid,'H5R_DATASET_REGION',ref_data(:,1));
H5S.close(space_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5S.copy

Create copy of data space

Syntax

```
output = H5S.copy(space_id)
```

Description

`output = H5S.copy(space_id)` creates a new data space, which is an exact copy of the dataspace identified by `space_id`. The `output` argument is a data space identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.1');
space1_id = H5D.get_space(dset_id);
space2_id = H5S.copy(space1_id);
[~,dims1] = H5S.get_simple_extent_dims(space1_id)
[~,dims2] = H5S.get_simple_extent_dims(space2_id)
```

See Also

`H5D.get_space` | `H5S.get_simple_extent_dims`

H5S.create

Create new data space

Syntax

```
space_id = H5S.create(space_type)
```

Description

`space_id = H5S.create(space_type)` creates a new dataspace of the type specified by `space_type`, which can be specified by one of the following strings.

```
'H5S_SCALAR'
'H5S_SIMPLE'
'H5S_NULL'
```

`space_id` is the identifier for the new dataspace.

Examples

Create a scalar dataspace.

```
space_id = H5S.create('H5S_SCALAR');
numpoints = H5S.get_simple_extent_npoints(space_id);
```

See Also

`H5S.get_simple_extent_npoints`

H5S.close

Close data space

Syntax

```
H5S.close(space_id)
```

Description

`H5S.close(space_id)` releases and terminates access to a data space. `space_id` is a data space identifier.

See Also

`H5A.get_space` | `H5D.get_space`

H5S.create_simple

Create new simple data space

Syntax

```
space_id = H5S.create_simple(rank,h5_dims,h5_maxdims)
```

Description

`space_id = H5S.create_simple(rank,h5_dims,h5_maxdims)` creates a new simple data space and opens it for access. `rank` is the number of dimensions used in the data space. `h5_dims` is an array specifying the size of each dimension of the dataset. `h5_maxdims` is an array specifying the upper limit on the size of each dimension. `space_id` is a data space identifier.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a data space for a dataset with 10 rows and 5 columns.

```
dims = [10 5];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
```

Create a data space for a dataset with 10 rows and 5 columns such that the dataset is extendible along the column dimension.

```
dims = [10 5];
h5_dims = fliplr(dims);
maxdims = [10 H5ML.get_constant_value('H5S_UNLIMITED')];
```

```
h5_maxdims = fliplr(maxdims);
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
```

See Also

H5ML.get_constant_value | H5S.close | H5S.create

H5S.extent_copy

Copy extent from source to destination data space

Syntax

```
H5S.extent_copy(dst_id,src_id)
```

Description

`H5S.extent_copy(dst_id,src_id)` copies the extent from the source data space, `src_id`, to the destination data space, `dst_id`.

Examples

```
space_id1 = H5S.create('H5S_SIMPLE');
dims = [100 200];
h5_dims = fliplr(dims);
maxdims = [100 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
H5S.set_extent_simple(space_id1,2,h5_dims,h5_maxdims);
space_id2 = H5S.create('H5S_SIMPLE');
H5S.extent_copy(space_id2,space_id1);
```

See Also

`H5S.create` | `H5S.get_simple_extent_dims` | `H5S.set_extent_simple`

H5S.get_select_bounds

Bounding box of data space selection

Syntax

```
[start,finish] = H5S.get_select_bounds(space_id)
```

Description

`[start,finish] = H5S.get_select_bounds(space_id)` returns the coordinates of the bounding box containing the current selection. `start` contains the starting coordinates of the bounding box and `finish` contains the coordinates of the diagonally opposite corner.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([30 40]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_OR',start,[],[],block);
[start, finish] = H5S.get_select_bounds(space_id);
matlab_start = fliplr(start);
matlab_finish = fliplr(finish);
```

See Also

`H5S.create_simple` | `H5S.select_hyperslab`

H5S.get_select_elem_npoints

Number of element points in selection

Syntax

```
numpoints = H5S.get_select_elem_npoints(space_id)
```

Description

`numpoints = H5S.get_select_elem_npoints(space_id)` returns the number of element points in the current data space selection.

Examples

Select the corner points of a data space.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
coords = fliplr(coords);
coords = coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',coords)
numpoints = H5S.get_select_elem_npoints(space_id);
```

See Also

`H5S.select_elements`

H5S.get_select_elem_pointlist

Element points in data space selection

Syntax

```
points =
H5S.get_select_elem_pointlist(space_id,startpoint,numpoints)
```

Description

`points = H5S.get_select_elem_pointlist(space_id,startpoint,numpoints)` returns the list of element points in the current data space selection. `startpoint` specifies the element point to start with and `numpoints` specifies the total number of points.

`points` is a two-dimensional array of 0-based values specifying the coordinates of the elements. If `m` is the rank of the dataspace, then `points` will have size [`m` x `numpoints`].

Note: The ordering of the coordinate points is the same as the HDF5 library C API.

Examples

Determine the first two points in the current selection.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
coords = fliplr(coords);
coords = coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',coords);
points = H5S.get_select_elem_pointlist(space_id,0,2);
```

H5S.get_select_hyper_blocklist

List of hyperslab blocks

Syntax

```
blocklist =
H5S.get_select_hyper_blocklist(space_id,startblock,numblocks)
```

Description

`blocklist = H5S.get_select_hyper_blocklist(space_id,startblock,numblocks)` returns a list of the hyperslab blocks currently selected. `space_id` is a dataspace identifier. `startblock` specifies the block to start with and `numblocks` specifies the number of hyperslab blocks to retrieve.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([20 30]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_NOTB',start,[],[],block);
nblocks = H5S.get_select_hyper_nblocks(space_id);
for j = 1:nblocks
    hblocks{j} = H5S.get_select_hyper_blocklist(space_id,j-1,1);
end
```

See Also

`H5S.get_select_hyper_nblocks` | `H5S.select_hyperslab`

H5S.get_select_hyper_nblocks

Number of hyperslab blocks

Syntax

```
num_blocks = H5S.get_select_hyper_nblocks(space_id)
```

Description

`num_blocks = H5S.get_select_hyper_nblocks(space_id)` returns the number of hyperslab blocks in the current data space selection.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([20 30]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_NOTB',start,[],[],block);
nblocks = H5S.get_select_hyper_nblocks(space_id);
```

See Also

`H5S.get_select_hyper_blocklist` | `H5S.select_hyperslab`

H5S.get_select_npoints

Number of elements in data space selection

Syntax

```
num_points = H5S.get_select_npoints(space_id)
```

Description

`num_points = H5S.get_select_npoints(space_id)` returns the number of elements in the current data space selection.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
op = 'H5S_SELECT_SET';
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
n = H5S.get_select_npoints(space_id);
```

See Also

`H5S.create_simple` | `H5S.select_hyperslab`

H5S.get_select_type

Type of data space selection

Syntax

```
sel_type = H5S.get_select_type(space_id)
```

Description

`sel_type = H5S.get_select_type(space_id)` returns the data space selection type. Valid return values correspond to the following enumerated constants:

```
H5S_SEL_NONE
H5S_SEL_POINTS
H5S_SEL_HYPERSLABS
H5S_SEL_ALL
```

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
sel_type = H5S.get_select_type(space_id);
switch(sel_type)
    case H5ML.get_constant_value('H5S_SEL_NONE')
        fprintf('no selection\n');
    case H5ML.get_constant_value('H5S_SEL_POINTS');
        fprintf('point selection\n');
    case H5ML.get_constant_value('H5S_SEL_HYPERSLABS');
        fprintf('hyperslab selection\n');
end
```

See Also

`H5ML.get_constant_value` | `H5S.select_elements` | `H5S.select_hyperslab`

H5S.get_simple_extent_dims

Data space size and maximum size

Syntax

```
[numdims,h5_dims,h5_maxdims] = H5S.get_simple_extent_dims(space_id)
```

Description

`[numdims,h5_dims,h5_maxdims] = H5S.get_simple_extent_dims(space_id)` returns the number of dimensions in the data space, the size of each dimension, and the maximum size of each dimension.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.2');
space_id = H5D.get_space(dset_id);
[ndims,h5_dims] = H5S.get_simple_extent_dims(space_id);
matlab_dims = fliplr(h5_dims);
```

H5S.get_simple_extent_ndims

Data space rank

Syntax

```
output = H5S.get_simple_extent_ndims(space_id)
```

Description

`output = H5S.get_simple_extent_ndims(space_id)` returns the dimensionality (also called the rank) of a data space.

H5S.get_simple_extent_npoints

Number of elements in data space

Syntax

```
output = H5S.get_simple_extent_npoints(space_id)
```

Description

`output = H5S.get_simple_extent_npoints(space_id)` returns the number of elements in the data space specified by `space_id`.

H5S.get_simple_extent_type

Data space class

Syntax

```
space_type = H5S.get_simple_extent_type(space_id)
```

Description

`space_type = H5S.get_simple_extent_type(space_id)` returns the data space class of the data space specified by `space_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
space_type = H5S.get_simple_extent_type(space_id);
switch(space_type)
    case H5ML.get_constant_value('H5S_SCALAR')
        fprintf('scalar\n');
    case H5ML.get_constant_value('H5S_SIMPLE')
        fprintf('simple\n');
    case H5ML.get_constant_value('H5S_NULL')
        fprintf('none\n');
end
```

See Also

`H5D.get_space` | `H5ML.get_constant_value` | `H5S.create`

H5S.is_simple

Determine if data space is simple

Syntax

```
output = H5S.is_simple(space_id)
```

Description

`output = H5S.is_simple(space_id)` returns a positive value if the data space specified by `space_id` is a simple data space, zero if it is not, and a negative value to indicate failure.

Examples

Create a new data space and verify that it is simple.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
val = H5S.is_simple(space_id);
```

Create a null data space and verify that it is not simple.

```
space_id = H5S.create('H5S_NULL');
val = H5S.is_simple(space_id);
```

See Also

`H5S.create` | `H5S.create_simple`

H5S.offset_simple

Set offset of simple data space

Syntax

```
H5S.offset_simple(space_id,offset)
```

Description

`H5S.offset_simple(space_id,offset)` specifies the offset of the simple data space specified by `space_id`. This function allows the same shaped selection to be moved to different locations within a data space without requiring it to be redefined.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
offset = fliplr([3 5]);
H5S.offset_simple(space_id,offset)
[start,finish] = H5S.get_select_bounds(space_id);
start = fliplr(start);
finish = fliplr(finish);
```

See Also

`H5S.get_select_bounds` | `H5S.select_hyperslab`

H5S.select_all

Select entire extent of data space

Syntax

```
H5S.select_all(space_id)
```

Description

`H5S.select_all(space_id)` selects the entire extent of the data space specified by `space_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
num_points1 = H5S.get_select_npoints(space_id);
H5S.select_none(space_id);
num_points2 = H5S.get_select_npoints(space_id);
H5S.select_all(space_id);
num_points3 = H5S.get_select_npoints(space_id);
```

H5S.select_elements

Specify coordinates to include in selection

Syntax

```
H5S.select_elements(space_id,op,h5_coord)
```

Description

`H5S.select_elements(space_id,op,h5_coord)` selects the array elements to be included in the selection for the data space specified by `space_id`. The `op` argument determines how the new selection is to be combined with the previously existing selection for the data space and can be specified by one of the following string values.

```
'H5S_SELECT_SET'
'H5S_SELECT_APPEND'
'H5S_SELECT_PREPEND'
```

`h5_coord` is a two-dimensional array of 0-based values specifying the coordinates of the elements being selected. If `m` is the rank of the data space and if n is the number of points, then `h5_coord` should be an `m`-by-`n` array.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_coord` parameter assumes coordinates have C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Select the corner points of a data space. In this case, `h5_coord` should have size 2x4.

```
dims = [100 200];
h5_dims = fliplr(dims);
```

```
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
h5_coords = fliplr(coords);
h5_coords = h5_coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',h5_coords);
```

See Also

H5S.create_simple | H5S.get_select_elem_npoints | H5S.get_select_elem_pointlist

H5S.select_hyperslab

Select hyperslab region

Syntax

```
H5S.select_hyperslab(space_id,op,h5_start,h5_stride,h5_count,h5_block)
```

Description

`H5S.select_hyperslab(space_id,op,h5_start,h5_stride,h5_count,h5_block)` selects a hyperslab region to add to the current selected region for the data space specified by `space_id`. The `op` argument determines how the new selection is to be combined with the previously existing selection for the data space. Possible values include: `H5S_SELECT_SET`, `H5S_SELECT_OR`, `H5S_SELECT_AND`, `H5S_SELECT_XOR`, `H5S_SELECT_NOTA`, or `H5S_SELECT_NOTB`.

The `h5_start` array determines the starting coordinates of the hyperslab to select. The `h5_count` array determines how many blocks to select from the data space, in each dimension. The `h5_stride` array specifies how many elements to move in each dimension. The `h5_block` array determines the size of the element block selected from the data space.

If `h5_stride` is specified as `[]`, then a contiguous hyperslab is selected, as if each value in `h5_stride` were set to 1. If `h5_count` is specified as `[]`, the number of blocks selected along each dimension defaults to 1. If `h5_block` is specified as `[]`, then the block size defaults to a single element in each dimension, as if each value in the block array were set to 1.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
```

```
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
```

See Also

H5S.create_simple

H5S.select_none

Reset selection region to include no elements

Syntax

```
H5S.select_none(space_id)
```

Description

`H5S.select_none(space_id)` resets the selection region for the data space `space_id` to include no elements.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
num_points1 = H5S.get_select_npoints(space_id);
H5S.select_none(space_id);
num_points2 = H5S.get_select_npoints(space_id);
```

H5S.select_valid

Determine validity of selection

Syntax

```
output = H5S.select_valid(space_id)
```

Description

`output = H5S.select_valid(space_id)` returns a positive value if the selection of the data space `space_id` is within the extent of that data space, and zero if it is not. A negative value indicates failure.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([90 190]); count = [11 11];
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],count,[]);
valid = H5S.select_valid(space_id);
```

See Also

`H5S.create_simple` | `H5S.select_hyperslab`

H5S.set_extent_none

Remove extent from data space

Syntax

```
H5S.set_extent_none(space_id)
```

Description

`H5S.set_extent_none(space_id)` removes the extent from a data space and sets the type to `H5S_NO_CLASS`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer2D');
space_id = H5D.get_space(dset_id);
H5S.set_extent_none(space_id);
extent_type = H5S.get_simple_extent_type(space_id);
switch(extent_type)
    case H5ML.get_constant_value('H5S_SCALAR')
        fprintf('scalar\n');
    case H5ML.get_constant_value('H5S_SIMPLE')
        fprintf('simple\n');
    case H5ML.get_constant_value('H5S_NO_CLASS')
        fprintf('no class\n');
end
```

See Also

`H5S.get_simple_extent_dims`

H5S.set_extent_simple

Set size of data space

Syntax

```
H5S.set_extent_simple(space_id,rank,h5_dims,h5_maxdims)
```

Description

`H5S.set_extent_simple(space_id,rank,h5_dims,h5_maxdims)` sets the size of the data space identified by `space_id`. The `rank` argument is the number of dimensions used in the data space. `h5_dims` is an array specifying the size of each dimension of the dataset. `h5_maxdims` is an array specifying the upper limit on the size of each dimension.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
space_id = H5S.create('H5S_SIMPLE');
dims = [100 200];
h5_dims = fliplr(dims);
maxdims = [100 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
H5S.set_extent_simple(space_id,2,h5_dims, h5_maxdims);
```

See Also

`H5ML.get_constant_value` | `H5S.create` | `H5S.get_simple_extent_dims`

H5T.close

Close data type

Syntax

```
H5T.close(type_id)
```

Description

`H5T.close(type_id)` releases the data type specified by `type_id`.

See Also

`H5A.get_type` | `H5D.get_type`

H5T.commit

Commit transient data type

Syntax

```
H5T.commit(loc_id,name,type_id)
H5T.commit(loc_id,name,type_id,lcpl_id,tcpl_id,tapl_id)
```

Description

`H5T.commit(loc_id,name,type_id)` commits a transient data type to a file, creating a new named data type. `loc_id` is a file or group identifier. `name` is the name of the data type and `type_id` is the data type id. This interface corresponds to the 1.6.x version of `H5Tcommit`.

`H5T.commit(loc_id,name,type_id,lcpl_id,tcpl_id,tapl_id)` commits a transient data type to a file, creating a new named data type. `loc_id` is a file or group identifier. `name` is the name of the data type and `type_id` is the data type id. `lcpl_id`, `tcpl_id`, and `tapl_id` are link creation, data type creation, and data type access property list identifiers. This interface corresponds to the 1.8.x version of `H5Tcommit`.

Examples

Create a named variable-length data type.

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
vlen_type_id = H5T.vlen_create(base_type_id);
H5T.commit(fid,'MyVlen',vlen_type_id);
H5T.close(vlen_type_id);
H5T.close(base_type_id);
H5F.close(fid);
```

See Also

`H5T.close` | `H5T.committed`

H5T.committed

Determine if data type is committed

Syntax

```
output = H5T.committed(type_id)
```

Description

`output = H5T.committed(type_id)` returns a positive value to indicate that the data type has been committed, and zero to indicate that it has not. A negative value indicates failure.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
is_committed = H5T.committed(type_id);
```

See Also

`H5T.commit`

H5T.copy

Copy data type

Syntax

```
output_type_id = H5T.copy(type_id)
```

Description

`output_type_id = H5T.copy(type_id)` copies the existing data type identifier, a dataset identifier specified by `type_id`, or a predefined data type such as `'H5T_NATIVE_DOUBLE'`. `output_type_id` is a data type identifier.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
type_size = H5T.get_size(type_id);
```

See Also

`H5T.get_size`

H5T.create

Create new data type

Syntax

```
output = H5T.create(class_id,size)
```

Description

`output = H5T.create(class_id,size)` creates a new data type of the class specified by `class_id`, with the number of bytes specified by `size`. The `output` argument is a data type identifier.

Examples

Create a signed 32-bit enumerated data type.

```
type_id = H5T.create('H5T_ENUM',4);
H5T.set_order(type_id,'H5T_ORDER_LE');
H5T.set_sign(type_id,'H5T_SGN_2');
H5T.enum_insert(type_id,'black',0);
H5T.enum_insert(type_id,'white',1);
```

See Also

`H5T.set_order` | `H5T.set_sign`

H5T.detect_class

Determine of data type contains specific class

Syntax

```
output = H5T.detect_class(type_id,class_id)
```

Description

`output = H5T.detect_class(type_id,class_id)` returns a positive value if the data type specified in `type_id` contains any data types of the data type class specified in `class_id`, or zero to indicate that it does not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen');
type_id = H5D.get_type(dset_id);
has_double = H5T.detect_class(type_id,'H5T_FLOAT');
```

See Also

`H5D.get_type`

H5T.equal

Determine equality of data types

Syntax

```
output = H5T.equal(type1_id,type2_id)
```

Description

`output = H5T.equal(type1_id,type2_id)` returns a positive number if the data type identifiers refer to the same data type, and zero to indicate that they do not. A negative value indicates failure. Either of the input values could be a string corresponding to an HDF5 data type.

Examples

Determine if the data type of a dataset is a 32-bit little endian integer.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer2D');
dtype_id = H5D.get_type(dset_id);
if H5T.equal(dtype_id,'H5T_STD_I32LE')
    fprintf('32-bit little endian integer\n');
end
```

See Also

`H5D.get_type`

H5T.get_class

Data type class identifier

Syntax

```
class_id = H5T.get_class(type_id)
```

Description

`class_id = H5T.get_class(type_id)` returns the data type class identifier of the data type specified by `type_id`.

Valid class identifiers include:

```
H5T_INTEGER
H5T_FLOAT
H5T_STRING
H5T_BITFIELD
H5T_OPAQUE
H5T_COMPOUND
H5T_ENUM
H5T_VLEN
H5T_ARRAY
```

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
class_id = H5T.get_class(type_id);
switch(class_id)
    case H5ML.get_constant_value('H5T_INTEGER')
```

```
        fprintf('Integer\n');
    case H5ML.get_constant_value('H5T_FLOAT')
        fprintf('Floating point\n');
    case H5ML.get_constant_value('H5T_STRING')
        fprintf('String\n');
    case H5ML.get_constant_value('H5T_BITFIELD')
        fprintf('Bitfield\n');
    case H5ML.get_constant_value('H5T_OPAQUE')
        fprintf('Opaque\n');
    case H5ML.get_constant_value('H5T_COMPOUND')
        fprintf('Compound'\n');
    case H5ML.get_constant_value('H5T_ENUM')
        fprintf('Enumerated\n');
    case H5ML.get_constant_value('H5T_VLEN')
        fprintf('Variable length\n');
    case H5ML.get_constant_value('H5T_ARRAY')
        fprintf('Array\n');
end
```

See Also

H5ML.get_constant_value

H5T.get_create_plist

Copy of data type creation property list

Syntax

```
plist_id = H5T.get_create_plist(datatype_id)
```

Description

`plist_id = H5T.get_create_plist(datatype_id)` returns a property list identifier for the data type creation property list associated with the data type specified by `datatype_id`.

See Also

`H5D.get_create_plist` | `H5F.get_create_plist`

H5T.get_native_type

Native data type of dataset data type

Syntax

```
output = H5T.get_native_type(type_id,direction)
```

Description

`output = H5T.get_native_type(type_id,direction)` returns the equivalent native data type for the dataset data type specified in `type_id`. The `direction` argument indicates the order in which the library searches for a native data type match and must be either `'H5T_DIR_ASCEND'` or `'H5T_DIR_DESCEND'`.

H5T.get_size

Size of data type in bytes

Syntax

```
type_size = H5T.get_size(type_id)
```

Description

`type_size = H5T.get_size(type_id)` returns the size of a data type in bytes. `type_id` is a data type identifier.

Examples

Determine the size of the data type for a specific dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/bitfield2D');
type_id = H5D.get_type(dset_id);
type_size = H5T.get_size(type_id);
```

See Also

`H5D.get_type` | `H5T.set_size`

H5T.get_super

Base data type

Syntax

```
super_type_id = H5T.get_super(type_id)
```

Description

`super_type_id = H5T.get_super(type_id)` returns the base data type from which the data type type specified by `type_id` is derived.

Examples

Retrieve the base data type for an enumerated dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
dtype_id = H5D.get_type(dset_id);
super_type_id = H5T.get_super(dtype_id);
```

H5T.lock

Lock data type

Syntax

```
H5T.lock(type_id)
```

Description

`H5T.lock(type_id)` locks the data type specified by `type_id`, making it read-only and non-destructible.

H5T.open

Open named data type

Syntax

```
type_id = H5T.open(loc_id,name)
```

Description

`type_id = H5T.open(loc_id,name)` opens a named data type at the location specified by `loc_id` and returns an identifier for the data type. `loc_id` is either a file or group identifier.

This function corresponds to the `H5Topen1` function in the HDF5 library C API.

See Also

`H5A.open` | `H5D.open` | `H5G.open` | `H5O.open` | `H5T.close`

H5T.array_create

Create array data type object

Syntax

```
array_type_id = H5T.array_create(base_id,h5_dims)
array_type_id = H5T.array_create(base_id,rank,h5_dims,perms)
```

Description

`array_type_id = H5T.array_create(base_id,h5_dims)` creates a new array data type object. This interface corresponds to the 1.8 library version of `H5Tarray_create`.

`array_type_id = H5T.array_create(base_id,rank,h5_dims,perms)` creates a new array data type object. This interface corresponds to the 1.6 library version of `H5Tarray_create`. The `perms` parameter is not used at this time and can be omitted.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a 100-by-200 double precision array data type.

```
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
array_type = H5T.array_create(base_type_id,h5_dims);
```

See Also

`H5T.get_array_dims` | `H5T.get_array_ndims`

H5T.get_array_dims

Array dimension extents

Syntax

```
dimsizes = H5T.get_array_dims(type_id)
[ndims,dimsizes,perm] = H5T.get_array_dims(type_id)
```

Description

`dimsizes = H5T.get_array_dims(type_id)` returns the sizes of the dimensions and the dimension permutations of the specified array data type object. This interface corresponds to the 1.8 version of `H5Tget_array_dims`.

`[ndims,dimsizes,perm] = H5T.get_array_dims(type_id)` corresponds to the 1.6 version of the interface. It is strongly deprecated.

Note: The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/array2D');
type_id = H5D.get_type(dset_id);
h5_dims = H5T.get_array_dims(type_id);
dims = fliplr(h5_dims);
```

See Also

`H5T.array_create` | `H5T.get_array_ndims`

H5T.get_array_ndims

Rank of array data type

Syntax

```
output = H5T.get_array_ndims(type_id)
```

Description

`output = H5T.get_array_ndims(type_id)` returns the rank, the number of dimensions, of an array data type object.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/array2D');
type_id = H5D.get_type(dset_id);
ndims = H5T.get_array_ndims(type_id);
```

See Also

`H5T.get_array_dims`

H5T.get_cset

Character set of string data type

Syntax

```
cset = H5T.get_cset(type_id)
```

Description

`cset = H5T.get_cset(type_id)` returns the character set type of the data type specified by `type_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/string');
type_id = H5D.get_type(dset_id);
cset = H5T.get_cset(type_id);
switch(cset)
  case H5ML.get_constant_value('H5T_CSET_ASCII')
      fprintf('ASCII\n');
  case H5ML.get_constant_value('H5T_CSET_UTF8')
      fprintf('UTF-8\n');
end
```

See Also

`H5T.set_cset`

H5T.get_ebias

Exponent bias of floating-point type

Syntax

```
output = H5T.get_ebias(type_id)
```

Description

`output = H5T.get_ebias(type_id)` returns the exponent bias of a floating-point type. `type_id` is data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
ebias = H5T.get_ebias(type_id);
```

See Also

`H5T.set_ebias`

H5T.get_fields

Floating-point data type bit field information

Syntax

```
[spos,epos,esize,mpos,msize] = H5T.get_fields(type_id)
```

Description

`[spos,epos,esize,mpos,msize] = H5T.get_fields(type_id)` returns information about the locations of the various bit fields of a floating point data type. `type_id` is a data type identifier. `spos` is the floating-point sign bit. `epos` is the exponent bit-position. `esize` is the size of the exponent in bits. `mpos` is the mantissa bit-position. `msize` is the size of the mantissa in bits.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
[spos, epos, esize, mpos, msize] = H5T.get_fields(type_id);
```

H5T.get_inpad

Internal padding type for floating-point data types

Syntax

```
pad_type = H5T.get_inpad(type_id)
```

Description

`pad_type = H5T.get_inpad(type_id)` returns the internal padding type for unused bits in floating-point data types. `type_id` is a data type identifier. `pad_type` can be `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
pad_type = H5T.get_inpad(type_id);
switch(pad_type)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('pad zero\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('pad one\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('pad background\n');
end
```

See Also

`H5T.set_inpad`

H5T.get_norm

Mantissa normalization type

Syntax

```
norm_type = H5T.get_norm(type_id)
```

Description

`norm_type = H5T.get_norm(type_id)` returns the mantissa normalization of a floating-point data type. `type_id` is a data type identifier. `norm_type` can be `H5T_NORM_IMPLIED`, `H5T_NORM_MSBSET`, or `H5T_NORM_NONE`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
norm_type = H5T.get_norm(type_id);
switch(norm_type)
    case H5ML.get_constant_value('H5T_NORM_IMPLIED')
        fprintf('MSB of mantissa is not stored, always 1\n');
    case H5ML.get_constant_value('H5T_NORM_MSBSET');
        fprintf('MSB of mantissa is always 1\n');
    case H5ML.get_constant_value('H5T_NORM_NONE')
        fprintf('mantissa is not normalized\n');
end
```

See Also

`H5T.set_norm`

H5T.get_offset

Bit offset of first significant bit

Syntax

```
offset = H5T.get_offset(type_id)
```

Description

`offset = H5T.get_offset(type_id)` returns the offset of the first significant bit. `type_id` is a data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
offset = H5T.get_offset(type_id);
```

See Also

`H5T.set_offset`

H5T.get_order

Byte order of atomic data type

Syntax

```
output = H5T.get_order(type_id)
```

Description

`output = H5T.get_order(type_id)` returns the byte order of an atomic data type. `type_id` is a data type identifier. Possible return values are the constant values corresponding to the following strings:

```
'H5T_ORDER_LE'
'H5T_ORDER_BE'
'H5T_ORDER_VAX'
```

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.2');
type_id = H5D.get_type(dset_id);
switch(H5T.get_order(type_id))
    case H5ML.get_constant_value('H5T_ORDER_LE')
        fprintf('little endian\n');
    case H5ML.get_constant_value('H5T_ORDER_BE')
        fprintf('big endian\n');
    case H5ML.get_constant_value('H5T_ORDER_VAX')
        fprintf('vax\n');
end
```

See Also

`H5ML.get_constant_value` | `H5T.set_order`

H5T.get_pad

Padding type of least and most-significant bits

Syntax

```
[lsb,msb] = H5T.get_pad(type_id)
```

Description

`[lsb,msb] = H5T.get_pad(type_id)` returns the padding type of the least and most-significant bit padding. `type_id` is a data type identifier. `lsb` is the least-significant bit padding type. `msb` is the most-significant bit padding type. Values for `lsb` and `msb` can be `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
[lsb,msb] = H5T.get_pad(type_id);
switch(lsb)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('lsb pad type is zeros\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('lsb pad type is ones\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('lsb pad type is background\n');
end
switch(msb)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('msb pad type is zeros\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('msb pad type is ones\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('msb pad type is background\n');
end
```

See Also

H5T.set_pad

H5T.get_precision

Precision of atomic data type

Syntax

```
output = H5T.get_precision(type_id)
```

Description

`output = H5T.get_precision(type_id)` returns the precision of an atomic data type. `type_id` is a data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
numbits = H5T.get_precision(type_id);
```

See Also

`H5T.set_precision`

H5T.get_sign

Sign type for integer data type

Syntax

```
sign_type = H5T.get_sign(type_id)
```

Description

`sign_type = H5T.get_sign(type_id)` returns the sign type for an integer type. `type_id` is a data type identifier. Valid types are: `H5T_SGN_NONE` or `H5T_SGN_2`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
sign_type = H5T.get_sign(type_id);
switch(sign_type)
    case H5ML.get_constant_value('H5T_SGN_NONE')
        fprintf('Unsigned integer type.\n');
    case H5ML.get_constant_value('H5T_SGN_2');
        fprintf('Signed integer type.\n');
end
```

See Also

`H5T.set_sign`

H5T.get_strpad

Storage mechanism for string data type

Syntax

```
output = H5T.get_strpad(type_id)
```

Description

`output = H5T.get_strpad(type_id)` returns the storage mechanism (padding type) for a string data type. Possible values are:

`H5T_STR_NULLPAD`	Pad with zeros
`H5T_STR_NULLTERM`	Null-terminate
`H5T_STR_SPACEPAD`	Pad with spaces

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/string');
type_id = H5D.get_type(dset_id);
padding = H5T.get_strpad(type_id);
switch(padding)
    case H5ML.get_constant_value('H5T_STR_NULLTERM')
        fprintf('null-terminated\n');
    case H5ML.get_constant_value('H5T_STR_NULLPAD')
        fprintf('padded with zeros\n');
    case H5ML.get_constant_value('H5T_STR_SPACEPAD')
        fprintf('padded with spaces\n');
end
```

See Also

`H5T.set_strpad`

H5T.set_cset

Set character dataset for string data type

Syntax

```
H5T.set_cset(type_id, cset)
```

Description

`H5T.set_cset(type_id, cset)` sets the character encoding used to create strings. The only valid type is `H5T_CSET_ASCII`.

Examples

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,10);
encoding = H5ML.get_constant_value('H5T_CSET_ASCII');
H5T.set_cset(type_id,encoding);
```

See Also

`H5T.get_cset`

H5T.set_ebias

Set exponent bias of floating-point data type

Syntax

```
H5T.set_ebias(type_id,ebias)
```

Description

`H5T.set_ebias(type_id,ebias)` sets the exponent bias of a floating-point type. `type_id` is a data type identifier. `ebias` is an exponent bias value.

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
H5T.set_size(type_id,32);
H5T.set_ebias(type_id,99);
```

See Also

`H5T.get_ebias`

H5T.set_fields

Set sizes and locations of floating-point bit fields

Syntax

```
H5T.set_fields(type_id,spos,epos,esize,mpos,msize)
```

Description

`H5T.set_fields(type_id,spos,epos,esize,mpos,msize)` sets the locations and sizes of the various floating-point bit fields. `spos` is the sign position. `epos` is the exponent in bits. `esize` is the size of exponent in bits. `mpos` is the mantissa bit position. `msize` is the size of the mantissa in bits.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
H5T.set_fields(type_id,30,24,6,0,2);
```

See Also

`H5T.get_fields`

H5T.set_inpad

Specify how unused internal bits are to be filled

Syntax

```
H5T.set_inpad(type_id,pad)
```

Description

`H5T.set_inpad(type_id,pad)` sets how unused internal bits of a floating point type are filled. `type_id` is the identifier of the data type. `inpad` specifies how to fill the bits: `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND` (leave background alone).

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
pad_type = H5ML.get_constant_value('H5T_PAD_ZERO');
H5T.set_inpad(type_id,pad_type);
```

See Also

`H5T.get_inpad`

H5T.set_norm

Set mantissa normalization of floating-point data type

Syntax

```
H5T.set_norm(type_id,norm)
```

Description

`H5T.set_norm(type_id,norm)` sets the mantissa normalization of a floating-point data type. Valid normalization types are: `H5T_NORM_IMPLIED`, `H5T_NORM_MSBSET`, or `H5T_NORM_NONE`.

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
norm_type = H5ML.get_constant_value('H5T_NORM_MSBSET');
H5T.set_norm(type_id,norm_type);
```

See Also

`H5T.get_norm`

H5T.set_offset

Set bit offset of first significant bit

Syntax

```
H5T.set_offset(type_id,offset)
```

Description

`H5T.set_offset(type_id,offset)` sets the bit offset of the first significant bit. `type_id` is the identifier of the data type. `offset` specifies the number of bits of padding that appear.

Examples

```
type_id = H5T.copy('H5T_NATIVE_INT');
H5T.set_offset(type_id,16);
```

See Also

`H5T.get_offset`

H5T.set_order

Set byte ordering of atomic data type

Syntax

```
H5T.set_order(type_id,type_order)
```

Description

`H5T.set_order(type_id,type_order)` sets the byte ordering of an atomic data type. `type_order` can be one of the following values:

`H5T_ORDER_LE`

`H5T_ORDER_BE`

`H5T_ORDER_VAX`

Examples

Create a big endian 32-bit integer type.

```
type_id = H5T.copy('H5T_NATIVE_INT');
order = H5ML.get_constant_value('H5T_ORDER_BE');
H5T.set_order(type_id,order);
```

See Also

`H5ML.get_constant_value` | `H5T.get_order`

H5T.set_pad

Set padding type for least and most significant bits

Syntax

```
H5T.set_pad(type_id,lsb,msb)
```

Description

`H5T.set_pad(type_id,lsb,msb)` sets the padding type for the least and most-significant bits. `type_id` is the identifier of the data type. `lsb` specifies the padding type for least-significant bits; `msb` for most-significant bits. Valid padding types are `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND` (leave background alone).

Examples

```
type_id = H5T.copy('H5T_NATIVE_INT');
lsb = H5ML.get_constant_value('H5T_PAD_ONE');
msb = H5ML.get_constant_value('H5T_PAD_ZERO');
H5T.set_pad(type_id,lsb,msb);
```

See Also

`H5T.get_pad`

H5T.set_precision

Set precision of atomic data type

Syntax

```
H5T.set_precision(type_id,prec)
```

Description

`H5T.set_precision(type_id,prec)` sets the precision of an atomic data type. `type_id` is a data type identifier. `prec` specifies the number of bits of precision for the data type.

H5T.set_sign

Set sign property for integer data type

Syntax

```
H5T.set_sign(type_id,sign)
```

Description

`H5T.set_sign(type_id,sign)` sets the sign property for an integer type. `type_id` is a data type identifier. `sign` specifies the sign type. Valid values are `H5T_SGN_NONE` or `H5T_SGN_2`.

Examples

```
type_id = H5T.copy('H5T_NATIVE_LONG');
sgn = H5ML.get_constant_value('H5T_SGN_NONE');
H5T.set_sign(type_id,sgn);
```

See Also

`H5T.get_sign`

H5T.set_size

Set size of data type in bytes

Syntax

```
H5T.set_size(type_id,type_size)
```

Description

`H5T.set_size(type_id,type_size)` sets the total size in bytes for the data type specified by `type_id`. The string `'H5T_VARIABLE'` can also be used if a variable length string is desired.

Examples

Create a variable length string with null termination.

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,'H5T_VARIABLE');
H5T.set_strpad(type_id,'H5T_STR_NULLTERM');
```

See Also

`H5T.get_size`

H5T.set_strpad

Set storage mechanism for string data type

Syntax

```
H5T.set_strpad(type_id,storage_type)
```

Description

`H5T.set_strpad(type_id,storage_type)` defines the storage mechanism for the string data type identified by `type_id`. The storage type may be one of the following values.

`'H5T_STR_NULLTERM'`	Null terminated
`'H5T_STR_NULLPAD'`	Padded with zeros
`'H5T_STR_SPACEPAD'`	Padded with spaces

Examples

Create a ten-character string data type with space padding.

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,10);
H5T.set_strpad(type_id,'H5T_STR_SPACEPAD');
```

See Also

`H5T.get_strpad`

H5T.get_member_class

Data type class for compound data type member

Syntax

```
output = H5T.get_member_class(type_id,membno)
```

Description

`output = H5T.get_member_class(type_id,membno)` returns the data type class of the compound data type member specified by `membno`. The `type_id` argument is the data type identifier of a compound object.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
member_name = H5T.get_member_name(type_id,0);
member_class = H5T.get_member_class(type_id,0);
```

See Also

`H5T.get_member_name`

H5T.get_member_index

Index of compound or enumeration type member

Syntax

```
idx = H5T.get_member_index(type_id,name)
```

Description

`idx = H5T.get_member_index(type_id,name)` returns the index of a field of a compound data type or an element of an enumeration data type. `type_id` is a data type identifier and name is a text string that identifies the target field or element.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
idx = H5T.get_member_index(type_id,'b');
```

See Also

`H5T.get_member_name`

H5T.get_member_name

Name of compound or enumeration type member

Syntax

```
name = H5T.get_member_name(type_id,membno)
```

Description

`name = H5T.get_member_name(type_id,membno)` returns the name of a field of a compound data type or an element of an enumeration data type. `type_id` is a data type identifier. `membno` is a zero-based index of the field or element whose name is to be retrieved.

Examples

Determine the name of the first field of a compound dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
dtype_id = H5D.get_type(dset_id);
member_name = H5T.get_member_name(dtype_id,0);
```

See Also

`H5T.get_member_index`

H5T.get_member_offset

Offset of field of compound data type

Syntax

```
output = H5T.get_member_offset(type_id,membno)
```

Description

`output = H5T.get_member_offset(type_id,membno)` returns the byte offset of the field specified by `membno` in the compound data type specified by `type_id`. Note that zero (0) is a valid offset.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
idx = H5T.get_member_offset(type_id,1);
```

See Also

`H5T.get_member_name`

H5T.get_member_type

Data type of specified member

Syntax

```
type_id = H5T.get_member_type(type_id,membno)
```

Description

`type_id = H5T.get_member_type(type_id,membno)` returns the data type of the member specified by `membno` in the data type specified by `type_id`.

Examples

Get the size of the data type of the first member of a compound data type.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
compound_type_id = H5D.get_type(dset_id);
member_type_id = H5T.get_member_type(compound_type_id,0);
type_size = H5T.get_size(member_type_id);
```

See Also

`H5D.get_type`

H5T.get_nmembers

Number of elements in enumeration type

Syntax

```
output = H5T.get_nmembers(type_id)
```

Description

`output = H5T.get_nmembers(type_id)` retrieves the number of fields in a compound data type or the number of members of an enumeration data type. `type_id` is a data type identifier.

Examples

Determine the number of fields in a compound dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
dtype_id = H5D.get_type(dset_id);
nmembers = H5T.get_nmembers(dtype_id);
```

H5T.insert

Add member to compound data type

Syntax

```
H5T.insert(type_id,name,offset,member_datatype)
```

Description

`H5T.insert(type_id,name,offset,member_datatype)` adds another member to the compound data type specified by `type_id`. The `name` argument is a text string that specifies the name of the new member, which must be unique in the compound data type. `offset` specifies where you want to insert the new member and `member_datatype` specifies the data type identifier of the new member.

Examples

```
type_id = H5T.create('H5T_COMPOUND',16);
H5T.insert(type_id,'first',0,'H5T_NATIVE_DOUBLE');
H5T.insert(type_id,'second',8,'H5T_NATIVE_INT');
H5T.insert(type_id,'third',12,'H5T_NATIVE_UINT');
```

See Also

`H5T.create`

H5T.pack

Recursively remove padding from compound data type

Syntax

```
H5T.pack(type_id)
```

Description

`H5T.pack(type_id)` recursively removes padding from within a compound data type to make it more efficient (space-wise) to store that data. `type_id` is a data type identifier.

H5T.enum_create

Create new enumeration data type

Syntax

```
output = H5T.enum_create(parent_id)
```

Description

`output = H5T.enum_create(parent_id)` creates a new enumeration data type based on the specified base data type, `parent_id`, which must be an integer type. `output` is a data type identifier for the new enumeration data type.

Examples

```
parent_id = H5T.copy('H5T_NATIVE_UINT');
type_id = H5T.enum_create(parent_id);
H5T.enum_insert(type_id,'red',1);
H5T.enum_insert(type_id,'green',2);
H5T.enum_insert(type_id,'blue',3);
H5T.close(type_id);
H5T.close(parent_id);
```

See Also

`H5T.enum_insert`

H5T.enum_insert

Insert enumeration data type member

Syntax

```
H5T.enum_insert(type_id,name,value)
```

Description

`H5T.enum_insert(type_id,name,value)` inserts a new enumeration data type member into the enumeration data type specified by `type_id`. The `name` argument is a text string that specifies the name of the new member of the enumeration and `value` is the value of the member.

Examples

```
parent_id = H5T.copy('H5T_NATIVE_UINT');
type_id = H5T.enum_create(parent_id);
H5T.enum_insert(type_id,'red',1);
H5T.enum_insert(type_id,'green',2);
H5T.enum_insert(type_id,'blue',3);
H5T.close(type_id);
H5T.close(parent_id);
```

See Also

`H5T.enum_create`

H5T.enum_nameof

Name of enumeration data type member

Syntax

```
name = H5T.enum_nameof(type_id,value)
```

Description

`name = H5T.enum_nameof(type_id,value)` returns the symbol name corresponding to a member of an enumeration data type. `type_id` specifies the enumeration data type. `value` identifies the member of the enumeration.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
name0 = H5T.enum_nameof(type_id,int32(0));
name1 = H5T.enum_nameof(type_id,int32(1));
```

See Also

`H5T.enum_valueof`

H5T.enum_valueof

Value of enumeration data type member

Syntax

```
value = H5T.enum_valueof(type_id,member_name)
```

Description

`value = H5T.enum_valueof(type_id,member_name)` returns the value corresponding to a specified member of an enumeration data type. `type_id` specifies the enumeration data type and `member_name` specifies the member.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
num_members = H5T.get_nmembers(type_id);
for j = 1:num_members
    member_name{j} = H5T.get_member_name(type_id,j-1);
    member_value(j) = H5T.enum_valueof(type_id,member_name{j});
end
```

See Also

`H5T.get_member_name` | `H5T.get_nmembers`

H5T.get_member_value

Value of enumeration data type member

Syntax

```
value = H5T.get_member_value(type_id,membno)
```

Description

`value = H5T.get_member_value(type_id,membno)` returns the value of the enumeration data type member specified by `membno`. The `type_id` argument is the data type identifier for the enumeration data type.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
num_members = H5T.get_nmembers(type_id);
for j = 1:num_members
    member_name{j} = H5T.get_member_name(type_id,j-1);
    member_value(j) = H5T.get_member_value(type_id,j-1);
end
```

See Also

`H5T.get_member_name` | `H5T.get_nmembers`

H5T.get_tag

Tag associated with opaque data type

Syntax

```
tag = H5T.get_tag(type_id)
```

Description

`tag = H5T.get_tag(type_id)` returns the tag associated with the opaque data type specified by `type_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/opaque');
dtype_id = H5D.get_type(dset_id);
tag = H5T.get_tag(dtype_id);
```

See Also

`H5T.set_tag`

H5T.set_tag

Tag opaque data type with description

Syntax

```
H5T.set_tag(type_id,tag)
```

Description

`H5T.set_tag(type_id,tag)` tags the opaque data type specified by `type_id`, with the descriptive ASCII string identifier, `tag`.

Examples

Create an opaque data type with a length of 4 bytes and a particular tag.

```
type_id = H5T.create('H5T_OPAQUE',4);
H5T.set_tag(type_id,'Created by MATLAB.');
```

See Also

`H5T.create` | `H5T.get_tag`

H5T.is_variable_str

Determine if data type is variable-length string

Syntax

```
output = H5T.is_variable_str(type_id)
```

Description

`output = H5T.is_variable_str(type_id)` returns a positive value if the data type specified by `type_id` is a variable-length string and zero if it is not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/VLstring2D');
type_id = H5D.get_type(dset_id);
if H5T.is_variable_str(type_id) > 0
    fprintf('variable length string\n');
end
```

See Also

`H5D.get_type` | `H5T.get_size` | `H5T.vlen_create`

H5T.vlen_create

Create new variable-length data type

Syntax

```
vlen_type_id = H5T.vlen_create(base_id)
```

Description

`vlen_type_id = H5T.vlen_create(base_id)` creates a new variable-length (VL) data type. `base_id` specifies the base type of the data type to create.

Examples

Create a variable length data type for 64-bit floating-point numbers.

```
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
vlen_type_id = H5T.vlen_create(base_type_id);
```

See Also

`H5T.is_variable_str`

H5Z.filter_avail

Determine if filter is available

Syntax

```
output = H5Z.filter_avail(filter_id)
```

Description

`output = H5Z.filter_avail(filter_id)` determines whether the filter specified by the filter identifier is available to the application. `filter_id` can be specified by one of the following strings or its numeric equivalent.

```
'H5Z_FILTER_DEFLATE'
'H5Z_FILTER_SHUFFLE'
'H5Z_FILTER_FLETCHER32'
'H5Z_FILTER_SZIP'
'H5Z_FILTER_NBIT'
'H5Z_FILTER_SCALEOFFSET'
```

Examples

Determine if the shuffle filter is available.

```
bool = H5Z.filter_avail('H5Z_FILTER_SHUFFLE');
```

See Also

`H5ML.get_constant_value`

H5Z.get_filter_info

Information about filter

Syntax

```
filter_config_flags = H5Z.get_filter_info(filter)
```

Description

`filter_config_flags = H5Z.get_filter_info(filter)` retrieves information about the filter specified by its identifier. At present, the information returned is the filter's configuration flags, indicating whether the filter is configured to decode data, to encode data, neither, or both. `filter_config_flags` should be used with the HDF5 constant values `H5Z_FILTER_CONFIG_ENCODE_ENABLED` and `H5Z_FILTER_CONFIG_DECODE_ENABLED` in a bitwise AND operation. If the resulting value is 0, then the encode or decode functionality is not available.

Examples

Determine if encoding is enabled for the deflate filter.

```
flags = H5Z.get_filter_info('H5Z_FILTER_DEFLATE');
functionality = H5ML.get_constant_value('H5Z_FILTER_CONFIG_ENCODE_ENABLED');
enabled = bitand(flags,functionality) > 0;
```

See Also

`bitand` | `H5ML.get_constant_value` | `H5Z.filter_avail`

hadamard

Hadamard matrix

Syntax

```
H = hadamard(n)
```

Description

`H = hadamard(n)` returns the Hadamard matrix of order `n`.

Definitions

Hadamard matrices are matrices of `1`'s and `-1`'s whose columns are orthogonal,

```
H'*H = n*I
```

where `[n n]=size(H)` and `I = eye(n,n)` ,.

They have applications in several different areas, including combinatorics, signal processing, and numerical analysis, [1], [2].

An `n`-by-`n` Hadamard matrix with `n > 2` exists only if `rem(n,4) = 0`. This function handles only the cases where `n`, `n/12`, or `n/20` is a power of 2.

Examples

The command `hadamard(4)` produces the 4-by-4 matrix:

```
1     1     1     1
1    -1     1    -1
1     1    -1    -1
1    -1    -1     1
```

References

[1] Ryser, H. J., *Combinatorial Mathematics*, John Wiley and Sons, 1963.

[2] Pratt, W. K., *Digital Signal Processing*, John Wiley and Sons, 1978.

See Also

`compan` | `hankel` | `toeplitz`

Introduced before R2006a

handle class

Abstract class for deriving handle classes

Description

`classdef` *`MyHandleClass`* `<` `handle` makes *`MyHandleClass`* a subclass of the `handle` class.

The `handle` class is the superclass for all classes that follow handle semantics. A *handle* is a reference to an object. If you copy an object's handle variable, MATLAB copies only the handle. Both the original and copy refer to the same object. For example, if a function modifies a handle object passed as an input argument, the modification affects the original input object.

In contrast, nonhandle objects (that is, value objects) are not references. Functions must return modified value objects to change the object outside of the function's workspace.

See “Modifying Objects” for information on passing objects to functions.

If you want to create a class that defines events, you must derive that class from the `handle` class.

The `handle` class is an abstract class, so you cannot create an instance of this class directly. You use the `handle` class to derive other classes, which can be concrete classes whose instances are handle objects. See “Handle Classes” for information on using handle classes.

Methods

addlistener	Create event listener
delete	Delete handle object
findobj	Find handle objects

findprop	Find `meta.property` object
isvalid	Determine valid handles
notify	Notify listeners that event is occurring
relationaloperators	Determine equality or sort handle objects

Events

`ObjectBeingDestroyed`	Triggered when the handle object is about to be destroyed, but before calling the `delete` method. Listener callbacks execute before MATLAB destroys the handle object.

Destroying Handle Objects

MATLAB destroys handle objects when there are no references to the object. You can explicitly remove a handle object by calling its `delete` method. The `handle` class enables you to control what happens when handle objects are destroyed, either implicitly when no references exist or explicitly when you delete the object.

Define `delete` Method in Subclass

Subclasses of `handle` can define their own `delete` method. MATLAB calls this method when deleting an object of the subclass, enabling your subclass to perform any specific actions necessary before deleting an object.

For more information, see “Handle Class Destructor”.

Create Listener for `ObjectBeingDestroyed` Event

Any code can respond to the pending deletion of a handle object by defining a listener for that object’s `ObjectBeingDestroyed` event. MATLAB triggers this event before calling the object’s `delete` method.

For more information on using events and listeners, see “Events and Listeners — Syntax and Techniques”.

Attributes

`Abstract`	`true`
`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

To learn about attributes of classes, see Class Attributes in the MATLAB Object-Oriented Programming documentation.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Derive Class from `handle`

Derive `MyClass` from `handle`:

```
classdef MyClass < handle
   properties
      Prop1
   end
   methods
      function obj = MyClass(arg)
         obj.Prop1 = arg;
      end
   end
end
```

More About

- Class Attributes

- “Handle Classes”
- “Supporting Both Handle and Value Subclasses”
- “Handle Objects”

addlistener

Class: handle

Create event listener

Syntax

```
lh = addlistener(Hsource,EventName,callback)
lh = addlistener(Hsource,property,EventName,callback)
```

Description

`lh = addlistener(Hsource,EventName,callback)` creates a listener for the specified event when triggered on the source object.

If `Hsource` is not scalar, the listener responds to the named event on any object in the `Hsource` array.

`lh = addlistener(Hsource,property,EventName,callback)` creates a listener for one of the predefined property events. There are four predefined property events:

Event Name	Event Occurs
`PreSet`	Immediately before the property value is set, before calling its set access method
`PostSet`	Immediately after the property value is set
`PreGet`	Immediately before a property value query is serviced, before calling its get access method
`PostGet`	Immediately after returning the property value to the query

Tips

- To remove a listener, delete the listener object returned by `addlistener`. For example,

```
delete(lh)
```

calls the handle class delete method to delete the object from the workspace and remove the listener.

- Redefining or clearing the variable containing the handle of the listener (for example, `lh`) does not delete the listener. The event object (`Hsource`) still has a reference to the `event.listener` object. `addlistener` ties the listener's lifecycle to the object that is the source of the event.
- To define a listener that is not tied to the event object, use the `event.listener` constructor directly to create the listener.

Input Arguments

`Hsource` — Event source

handle array

Event source is the object that is source of the event, or an array of source objects, specified as a handle array.

`EventName` — Name of event

character array

Name of event that is triggered on the source objects, specified as a case-sensitive, quoted string. For property events, the event name is one of the four predefined property events.

`property` — Name of property

string

Name of the property whose property event triggers your listener, specified as one of these values:

- A scalar `meta.property` object
- An array or a cell array of `meta.property` objects
- A string or a cell array of strings, where each string is the name of a property defined for the objects in `Hsource`

  You can listen to property events on dynamic properties only If `Hsource` is scalar. If `Hsource` is non-scalar, then the properties must belong to the class of `Hsource` and can not include dynamic properties (which are not part of the class definition).

The class defining the source property must set the `GetObservable` and `SetObservable` property attributes to enable you to listen to the property events.

`callback` — Listener callback
function handle

Listener callback specified as a function handle

Data Types: `function_handle`

Output Arguments

`lh` — listener
`event.listener` or `event.proplistener`

Listener object created by `addlistener`, specified as the handle to an `event.listener` or an `event.proplistener` object.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Alternatives

When you need the lifecycle of the listener object to be independent of the source object lifecycle, use the `even.listener` or `event.proplistener` to create listeners.

See Also
`handle.notify` | `event.listener`

Related Examples
- "Learn to Use Events and Listeners"

- “Creating Property Listeners”
- “Limiting Listener Scope — Constructing event.listener Objects Directly”

delete

Class: handle

Delete handle object

Syntax

```
delete(h)
```

Description

`delete(h)` deletes a handle object, but does not clear the handle variable from the workspace. The handle variable is not valid once the handle object has been deleted.

Subclasses of `handle` can implement a method named `delete` to perform cleanup tasks just before MATLAB destroys an object of the class. MATLAB calls the `delete` method of any `handle` object when the object is destroyed when the subclass `delete` method meets certain criteria.

For information on implementing a class destructor for a subclass of `handle`, see "Handle Class Destructor".

Input Arguments

h — Objects to destroy

handle array

Objects to destroy, specified as a handle array.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`isvalid`

More About

- “Handle Class Destructor”
- “Handle Objects”

findobj

Class: handle

Find handle objects

Syntax

```
Hmatch = findobj(H)
Hmatch = findobj(H,property,value,...,property,value)
Hmatch = findobj(H,'-not',property,value)
Hmatch = findobj(H,'-regexp',property,value)
Hmatch = findobj(H,property,value,'-logicaloperator',property,value)
Hmatch = findobj(H,'-function',fh)
Hmatch = findobj(H,'-function',property,fh)
Hmatch = findobj(H,'-property',property)
Hmatch = findobj(H,'-method',methodname)
Hmatch = findobj(H,'-event',eventname)
Hmatch = findobj(H,'-depth',d,___)
Hmatch = findobj(H,'flat',___)
```

Description

`Hmatch = findobj(H)` returns the objects listed in `H` and all of their descendants.

`Hmatch = findobj(H,property,value,...,property,value)` finds `handle` objects that have the specified property set to the specified value.

`Hmatch = findobj(H,'-not',property,value)` inverts the expression in the following `property value` pair. That is, find objects whose specified property is not equal to `value`.

`Hmatch = findobj(H,'-regexp',property,value)` treats the content of the `value` argument as regular expressions.

`Hmatch = findobj(H,property,value,'-logicaloperator',property,value)` applies the logical operator to the name/value pairs. Supported logical operators include:

- `'-or'` (default if you do not specify an operator)
- `'-and'`
- `'-xor'`

`Hmatch = findobj(H,'-function',fh)` calls the function handle `fh` on the objects in `H` and returns the objects for which the function returns `true`.

`Hmatch = findobj(H,'-function',property,fh)` calls the function handle `fh` on the specified property's value for the objects in `H` and returns the objects for which the function returns `true`. The function must return a scalar logical value.

`Hmatch = findobj(H,'-property',property)` finds all object in `H` having the named property.

`Hmatch = findobj(H,'-method',methodname)` finds objects that have the specified method name.

`Hmatch = findobj(H,'-event',eventname)` finds objects that have the specified method name.

`Hmatch = findobj(H,'-depth',d, ___ )` specifies how many levels in the instance hierarchies under the objects in `H` to search.

`Hmatch = findobj(H,'flat', ___ )` searches only the level of the objects in array `H`.

Tips

- `findobj` has access only to public properties
- If there are no matches, `findobj` returns an empty array of the same class as the input array `H`.
- Logical operator precedence follows MATLAB precedence rules. For more information, see "Operator Precedence".
- Control precedence by grouping within cell arrays

Input Arguments

`H` — Objects to search from

handle array

Objects to search from, specified as an array of object handles. Unless the you specify the `'-depth'` or `'flat'` options, `findobj` searches the objects in the input array `H` and child objects in the instance hierarchy.

`property` — Property name
string

Property name, specified as a case-sensitive, quoted string.

`value` — Property value
any value

Property value, specified as a value or MATLAB expression.

`methodname` — Method name
string

Method name, specified as a case-sensitive quoted string.

`eventname` — Event name
string

Event name, specified as a case-sensitive quoted string.

`d` — Depth of search
integer >= 0

Depth of search, specified as an integer indicating the number of levels below any given object in the input array `H`.

- `d = n` — Search `n` levels of the hierarchy below each object in `H`
- `d = 0` — Search only the same level as the objects in `H`.
- `d = inf` — Search all levels below objects in `H`. This is the default.

`fh` — Function handle
function handle

Function handle, specifying the function that is evaluated for each object in the input array `H`. This function must return a scalar, logical value indicating whether there is a match (`true`) or not (`false`).

Output Arguments

`Hmatch` — Objects found by search

handle array

Objects found by search, returned as a handle array.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Object with Specific Property Value

Find the object with a specific property value. Given the `handle` class, `BasicHandle`:

```
classdef BasicHandle < handle
   properties
      Prop1
   end
   methods
      function obj = BasicHandle(val)
         if nargin > 0
            obj.Prop1 = val;
         end
      end
   end
end
```

Create an array of `BasicHandle` objects:

```
h(1) = BasicHandle(7);
h(2) = BasicHandle(11);
h(3) = BasicHandle(27);
```

Find the handle of the object whose `Prop1` property has a value of 7:

```
h7 = findobj(h,'Prop1',7);
h7.Prop1

ans =

     7
```

Object with Specific Property Name

Find the object with a specific dynamic property. Given the `button` class:

```
classdef button < dynamicprops
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         if nargin > 0
            if length(pos) == 4
               obj.UiHandle = uicontrol('Position',pos,...
                'Style','pushbutton');
            else
               error('Improper position')
            end
         end
      end
   end
end
```

Create an array of button objects, only one element of which defines a dynamic property. Use `findobj` to get the handle of the object with the dynamic property named `ButtonCoord`:

```
b(1) = button([20 40 80 20]);
b(1).addprop('ButtonCoord');
b(1).ButtonCoord = [2,3];
b(2) = button([120 40 80 20]);
b(3) = button([220 40 80 20]);

h =  findobj(b,'-property','ButtonCoord');
h.ButtonCoord

ans =
```

2 3

See Also

handle.findprop

findprop

Class: handle

Find `meta.property` object

Syntax

```
mp = findprop(h,property)
```

Description

`mp = findprop(h,property)` returns the `meta.property` object associated with the named property of the object `h`. `property` can be a property defined by the class of `h` or a dynamic property defined only for the object `h`.

Input Arguments

`h` — handle object

handle

Handle object, specified as a scalar handle.

`property` — Name of property

string

Name of property, specified as a case-sensitive, quoted string.

Output Arguments

`mp` — `meta.property` object

`meta.property`

`meta.property` object that is associated with the named property. If `findprop` does not find the property on the object `h`, `findprop` returns a 0-by-1 empty `meta.property` object.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

View Property Attribute Settings

Use `findprop` to view property attribute settings:

```
mp = findprop(containers.Map,'Count')
```

```
mp =

  property with properties:

                  Name: 'Count'
           Description: 'Number of pairs in the collection'
    DetailedDescription: ''
              GetAccess: 'public'
              SetAccess: 'private'
              Dependent: 1
               Constant: 0
               Abstract: 0
              Transient: 1
                 Hidden: 0
         GetObservable: 0
         SetObservable: 0
              AbortSet: 0
             GetMethod: []
             SetMethod: []
            HasDefault: 0
         DefiningClass: [1x1 meta.class]
```

See Also

`handle.findobj` | `meta.property`

More About

- "Getting Information About Properties"
- "Dynamic Properties — Adding Properties to an Instance"

isvalid

Class: handle

Determine valid handles

Syntax

```
B = isvalid(H)
```

Description

`B = isvalid(H)` returns a logical array in which each element is `true` if the corresponding element in `H` is a valid handle. A handle variable becomes invalid if the object has been deleted.

You cannot override the `isvalid` method in `handle` subclasses.

Input Arguments

`H` — Input array

handle array

Input array, specified as an array of object handles.

Output Arguments

`B` — Result of validity test

logical

Result of validity test, returned as a logical array the same size as `H` in which each element is `true` if the corresponding element in `H` is a valid handle.

Attributes

`Access`	`public`

```
Sealed                                  true
```

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Test for Valid Handles

This example tests a handle array for valid members:

```
H = plot(rand(5));
delete(H(3:4))
B = isvalid(H)
```

```
B =

     1
     1
     0
     0
     1
```

See Also

`handle.delete` | `isgraphics`

More About

- "Testing Handle Validity"

notify

Class: handle

Notify listeners that event is occurring

Syntax

```
notify(H,EventName)
notify(H,EventName,data)
```

Description

`notify(H,EventName)` notifies listeners that the named event is taking place on the handle objects in `H`.

`notify(H,EventName,data)` includes user-defined event data.

Input Arguments

`H` — Event source

handle array

Event source, specified as a handle array. All of the objects in `H` must defined the named event.

`EventName` — Name of event

string

Name of event, specified as a case-sensitive, quoted string that is defined by the class of `H`.

`data` — User-defined event data

subclass of `event.EventData`

User-defined event data, specified as an object that is a subclass of the `event.EventData` class. For information on defining event data, see "Define Event-Specific Data".

Note: Listener callbacks should not reuse the same event data object in subsequent calls to `notify`. Always create a new event data object to pass to `notify`.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`handle.addlistener`

More About

- "Events and Listeners — Syntax and Techniques"

relationaloperators

Class: handle

Determine equality or sort handle objects

Syntax

```
tf = eq(H1,H2)
tf = ne(H1,H2)
tf = lt(H1,H2)
tf = le(H1,H2)
tf = gt(H1,H2)
tf = ge(H1,H2)
```

Description

`tf = eq(H1,H2)` Equal. (`H1 == H2`)

`tf = ne(H1,H2)` Not equal. (`H1 ~= H2`)

`tf = lt(H1,H2)` Less than. (`H1 < H2`)

`tf = le(H1,H2)` Less than or equal. (`H1 <= H2`)

`tf = gt(H1,H2)` Greater than. (`H1 > H2`)

`tf = ge(H1,H2)` Greater than or equal. (`H1 >= H2`)

For each pair of input arrays (`H1` and `H2`), the operation returns a logical array of the same size. Each element in the returned array is an element-wise equality or comparison test result. These methods perform scalar expansion in the same way as the MATLAB built-in relational operators. See `relationaloperators` for more information.

The following guidelines apply to handle comparison:

- Copies of a handle variable always compare as equal.
- The repeated comparison of any two handles always yields the same result in the same MATLAB session.

- Different handles are always not equal.
- The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.
- If the input arrays belong to different classes (including the case where one input array belongs to a non-handle class such as `double`) then the comparison is always false.
- If you make a comparison between a handle object and an object of a dominant class, the method of the dominant class is invoked. You should generally test only like objects because a dominant class might not define one of these methods.
- An error occurs if the input arrays are not the same size and neither is scalar.

Use `isequal` when you want to determine if different handle objects have the same data in all object properties.

Input Arguments

`H1` — Left argument to operator

handle array

Left argument to operator, specified as a handle array.

`H2` — Right argument to operator

handle array

Right argument to operator, specified as a handle array.

Output Arguments

`tf` — Result of comparison

logical array

Result of comparison, returned as a logical array of the same size as the input arrays, where each element is an element-wise equality or comparison test result

- `1` — relationship is true
- `0` — relationship if not true

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`isequal`

hankel

Hankel matrix

Syntax

```
H = hankel(c)
H = hankel(c,r)
```

Description

`H = hankel(c)` returns the square Hankel matrix whose first column is `c` and whose elements are zero below the first anti-diagonal.

`H = hankel(c,r)` returns a Hankel matrix whose first column is `c` and whose last row is `r`. If the last element of `c` differs from the first element of `r`, the last element of `c` prevails.

Definitions

A Hankel matrix is a matrix that is symmetric and constant across the anti-diagonals, and has elements `h(i,j) = p(i+j-1)`, where vector `p = [c r(2:end)]` completely determines the Hankel matrix.

Examples

A Hankel matrix with anti-diagonal disagreement is

```
c = 1:3; r = 7:10;
h = hankel(c,r)
h =
    1    2    3    8
    2    3    8    9
    3    8    9   10
```

```
p = [1 2 3 8 9 10]
```

See Also

`hadamard` | `kron` | `toeplitz`

Introduced before R2006a

hasnext

Determine if ValueIterator has one or more values available

Syntax

```
tf = hasnext(ValIter)
```

Description

`tf = hasnext(ValIter)` returns logical `1` (`true`) if `ValueIter` has one or more values available; otherwise, it returns logical `0` (`false`).

Examples

Get Values from ValueIterator in Reduce Function

Use the `hasnext` and `getnext` functions in a `while` loop within the reduce function to iteratively get values from the `ValueIterator` object. For example,

```
function MeanDistReduceFun(sumLenKey, sumLenIter, outKVStore)
    sumLen = [0 0];
    while hasnext(sumLenIter)
        sumLen = sumLen + getnext(sumLenIter);
    end
    add(outKVStore, 'Mean', sumLen(1)/sumLen(2));
end
```

Always call `hasnext` before `getnext` to confirm availability of a value. `mapreduce` returns an error if you call `getnext` with no remaining values in the `ValueIterator` object.

Input Arguments

`ValIter` — Intermediate value iterator

`ValueIterator` object

Intermediate value iterator, specified as a `ValueIterator` object. The `mapreduce` function automatically creates this object during execution. The second input to the reduce function specifies the variable name for the `ValueIterator` object, which is the variable name to use with the `hasnext` and `getnext` functions.

For more information, see Using ValueIterator Objects.

More About

- Using ValueIterator Objects
- "Build Effective Algorithms with MapReduce"

See Also

`getnext` | `mapreduce`

Introduced in R2014b

hasFrame

Class: VideoReader

Determine if frame available to read

Syntax

```
tf = hasFrame(obj)
```

Description

`tf = hasFrame(obj)` returns logical `1` (`true`) if there is a video frame available to read from the file. Otherwise, it returns logical `0` (`false`).

Input Arguments

`obj` — Object associated with video file to read

`VideoReader` object

Object associated with the video file to read, specified as a `VideoReader` object.

Examples

Read and Play Back Movie File

Read and play back the sample movie file, `xylophone.mp4`.

Construct a `VideoReader` object to read data from the sample file. Then, determine the width and height of the video.

```
xyloObj = VideoReader('xylophone.mp4');

vidWidth = xyloObj.Width;
vidHeight = xyloObj.Height;
```

Create a movie structure array, `mov`.

```
mov = struct('cdata',zeros(vidHeight,vidWidth,3,'uint8'),...
    'colormap',[]);
```

Read one frame at a time until the end of the video is reached.

```
k = 1;
while hasFrame(xyloObj)
    mov(k).cdata = readFrame(xyloObj);
    k = k+1;
end
```

Size a figure based on the video's width and height. Then, play back the movie once at the video's frame rate.

```
hf = figure;
set(hf,'position',[150 150 vidWidth vidHeight]);

movie(hf,mov,1,xyloObj.FrameRate);
```

- "Read Video Files"

See Also

`movie` | `VideoReader`

hdf5info

Information about HDF5 file

Note: `hdf5info` will be removed in a future version. Use `h5info` instead.

Syntax

```
fileinfo = hdf5info(filename)
fileinfo = hdf5info(...,'ReadAttributes',BOOL)
[...] = hdf5info(..., 'V71Dimensions', BOOL)
```

Description

`fileinfo = hdf5info(filename)` returns a structure `fileinfo` whose fields contain information about the contents of the HDF5 file `filename`. `filename` is a string that specifies the name of the HDF5 file.

`fileinfo = hdf5info(...,'ReadAttributes',BOOL)` specifies whether `hdf5info` returns the values of the attributes or just information describing the attributes. By default, `hdf5info` reads in attribute values (`BOOL` = `true`).

`[...] = hdf5info(..., 'V71Dimensions', BOOL)` specifies whether to report the dimensions of data sets and attributes as they were returned in previous versions of `hdf5info` (MATLAB 7.1 [R14SP3] and earlier). If `BOOL` is true, `hdf5info` swaps the first two dimensions of the data set. This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, swapping these dimensions may not correctly reflect the intent of the data in the file and may invalidate metadata. When `BOOL` is false (the default), `hdf5info` returns data dimensions that correctly reflect the data ordering as it is written in the file —each dimension in the output variable matches the same dimension in the file.

Note If you use the `'V71Dimensions'` parameter and intend on passing the `fileinfo` structure returned to the `hdf5read` function, you should also specify the

`'V71Dimensions'` parameters with `hdf5read`. If you do not, `hdf5read` uses the new behavior when reading the data set and certain metadata returned by `hdf5info` does not match the actual data returned by `hdf5read`.

Examples

```
fileinfo = hdf5info('example.h5');
```

To get more information about the contents of the HDF5 file, look at the `GroupHierarchy` field in the `fileinfo` structure returned by `hdf5info`.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: [1x64 char]
          Name: '/'
        Groups: [1x2 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [1x2 struct]
```

To probe further into the file hierarchy, keep examining the `Groups` field.

See also

`hdf5read`, `hdf5write`

hdf5read

Read HDF5 file

Note: `hdf5read` will be removed in a future version. Use `h5read` instead.

Syntax

```
data = hdf5read(filename,datasetname)
attr = hdf5read(filename,attributename)
[data, attr] = hdf5read(...,'ReadAttributes',BOOL)
data = hdf5read(hinfo)
[...] = hdf5read(..., 'V71Dimensions', BOOL)
```

Description

`data = hdf5read(filename,datasetname)` reads all the data in the data set `datasetname` that is stored in the HDF5 file `filename` and returns it in the variable `data`. To determine the names of data sets in an HDF5 file, use the `hdf5info` function.

The return value, `data`, is a multidimensional array. `hdf5read` maps HDF5 data types to native MATLAB data types, whenever possible. If it cannot represent the data using MATLAB data types, `hdf5read` uses one of the HDF5 data type objects. For example, if an HDF5 file contains a data set made up of an enumerated data type, `hdf5read` uses the `hdf5.h5enum` object to represent the data in the MATLAB workspace. The `hdf5.h5enum` object has data members that store the enumerations (names), their corresponding values, and the enumerated data.

Note: `hdf5read` performs best when reading numeric datasets. If you need to read string, compound, or variable length datasets, MathWorks strongly recommends that you use the low-level HDF5 interface function, `H5D.read`. To read a subset of a dataset, you must use the low-level interface.

`attr = hdf5read(filename,attributename)` reads all the metadata in the attribute `attributename`, stored in the HDF5 file `filename`, and returns it in the variable `attr`. To determine the names of attributes in an HDF5 file, use the `hdf5info` function.

`[data, attr] = hdf5read(...,'ReadAttributes',BOOL)` reads all the data, as well as all of the associated attribute information contained within that data set. By default, `BOOL` is false.

`data = hdf5read(hinfo)` reads all of the data in the data set specified in the structure `hinfo` and returns it in the variable `data`. The `hinfo` structure is extracted from the output returned by `hdf5info`, which specifies an HDF5 file and a specific data set.

`[...] = hdf5read(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets read from the file. If `BOOL` is true, `hdf5read` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data dimensions correctly reflect the data ordering as it is written in the file — each dimension in the output variable matches the same dimension in the file.

Examples

Use `hdf5info` to get information about an HDF5 file and then use `hdf5read` to read a data set, using the information structure (`hinfo`) returned by `hdf5info` to specify the data set.

```
hinfo = hdf5info('example.h5');
dset = hdf5read(hinfo.GroupHierarchy.Groups(2).Datasets(1));
```

See Also

`hdf5info` | `hdf5write`

Introduced before R2006a

hdf5write

Write data to file in HDF5 format

Note: `hdf5write` will be removed in a future version. Use `h5write` instead.

Syntax

```
hdf5write(filename,location,dataset)
hdf5write(filename,details,dataset)
hdf5write(filename,details,attribute)
hdf5write(filename, details1, dataset1, details2, dataset2,...)
hdf5write(filename,...,'WriteMode',mode,...)
hdf5write(..., 'V71Dimensions', BOOL)
```

Description

`hdf5write(filename,location,dataset)` writes the data `dataset` to the HDF5 file, `filename`. If `filename` does not exist, `hdf5write` creates it. If `filename` exists, `hdf5write` overwrites the existing file, by default, but you can also append data to an existing file using an optional syntax.

`location` defines where to write the data set in the file. HDF5 files are organized in a hierarchical structure similar to a UNIX directory structure. `location` is a string that resembles a UNIX path.

`hdf5write` maps the data in `dataset` to HDF5 data types according to rules outlined below.

`hdf5write(filename,details,dataset)` writes `dataset` to `filename` using the values in the `details` structure. For a data set, the `details` structure can contain the following fields.

Field Name	Description	Data Type
`Location`	Location of the data set in the file	Character array

Field Name	Description	Data Type
`Name`	Name to attach to the data set	Character array

`hdf5write(filename,details,attribute)` writes the metadata `attribute` to `filename` using the values in the `details` structure. For an attribute, the `details` structure can contain following fields.

Field Name	Description	Data Type
`AttachedTo`	Location of the object this attribute modifies	Structure array
`AttachType`	Identifies what kind of object this attribute modifies; possible values are `'group'` and `'dataset'`	Character array
`Name`	Name to attach to the data set	Character array

`hdf5write(filename, details1, dataset1, details2, dataset2,...)` writes multiple data sets and associated attributes to `filename` in one operation. Each data set and attribute must have an associated `details` structure.

`hdf5write(filename,...,'WriteMode',mode,...)` specifies whether `hdf5write` overwrites the existing file (the default) or appends data sets and attributes to the file. Possible values for mode are `'overwrite'` and `'append'`.

`hdf5write(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets written to the file. If `BOOL` is true, `hdf5write` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data written to the file correctly reflects the data ordering of the data sets — each dimension in the file's data sets matches the same dimension in the corresponding MATLAB variable.

Data Type Mappings

The following table lists how `hdf5write` maps the data type from the workspace into an HDF5 file. If the data in the workspace that is being written to the file is a MATLAB

data type, `hdf5write` uses the following rules when translating MATLAB data into HDF5 data objects.

MATLAB Data Type	HDF5 Data Set or Attribute
Numeric	Corresponding HDF5 native data type. For example, if the workspace data type is `uint8`, the `hdf5write` function writes the data to the file as 8-bit integers. The size of the HDF5 dataspace is the same size as the MATLAB array.
String	Single, null-terminated string
Cell array of strings	Multiple, null-terminated strings, each the same length. Length is determined by the length of the longest string in the cell array. The size of the HDF5 dataspace is the same size as the cell array.
Cell array of numeric data	Numeric array, the same dimensions as the cell array. The elements of the array must all have the same size and type. The data type is determined by the first element in the cell array.
Structure array	HDF5 compound type. Individual fields in the structure employ the same data translation rules for individual data types. For example, a cell array of strings becomes a multiple, null-terminated strings.
HDF5 objects	If the data being written to the file is composed of HDF5 objects, `hdf5write` uses the same data type when writing to the file. For all HDF5 objects, except `HDF5.h5enum` objects, the dataspace has the same dimensions as the array of HDF5 objects passed to the function. For `HDF5.h5enum` objects, the size and dimensions of the data set in the HDF5 file is the same as the object's Data field.

Examples

Write a 5-by-5 data set of `uint8` values to the root group.

```
hdf5write('myfile.h5', '/dataset1', uint8(magic(5)))
```

Write a 2-by-2 string data set in a subgroup.

```
dataset = {'north', 'south'; 'east', 'west'};
hdf5write('myfile2.h5', '/group1/dataset1.1', dataset);
```

Write a data set and attribute to an existing group.

```
dset = single(rand(10,10));
dset_details.Location = '/group1/dataset1.2';
dset_details.Name = 'Random';

attr = 'Some random data';
attr_details.Name = 'Description';
attr_details.AttachedTo = '/group1/dataset1.2/Random';
attr_details.AttachType = 'dataset';

hdf5write('myfile2.h5', dset_details, dset, ...
          attr_details, attr, 'WriteMode', 'append');
```

Write a data set using objects.

```
dset = hdf5.h5array(magic(5));
hdf5write('myfile3.h5', '/g1/objects', dset);
```

See Also

`hdf5read` | `hdf5info`

Introduced before R2006a

hdfan

Gateway to HDF multifile annotation (AN) interface

Syntax

```
[out1,...,outN] = hdfan(funcstr,input1,...,inputN)
```

Description

`hdfan` is the MATLAB gateway to the HDF multifile annotation (AN) interface.

`[out1,...,outN] = hdfan(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the AN function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between AN functions in the HDF library and valid values for `funcstr`. For example, `hdfan('endaccess',annot_id)` corresponds to the C library call `ANendaccess(annot_id)`.

Access Functions

Access functions initialize the interface and provide and terminate access to annotations.

Value of `funcstr`	Function Syntax	Description
`'start'`	`AN_id = hdfan('start',file_id)`	Initializes the multifile annotation interface.
`'select'`	`annot_id = hdfan('select',AN_id, index,annot_type)`	Selects and returns the identifier for the annotation identified by the given index value and annotation type.
`'end'`	`status = hdfan('end',AN_id)`	Terminates access to the multifile annotation interface.
`'create'`	`annot_id = hdfan('create',AN_id,tag,r`	Creates a data annotation for the object identified by the specified tag and reference number. `annot_type` can be `'data_label'` or `'data_desc'`.

Value of `funcstr`	Function Syntax	Description
`'createf'`	`annot_id = hdfan('createf',AN_id,anno`	Creates a file label or file description annotation. `annot_type` can be `'file_label'` or `'file_desc'`.
`'endaccess'`	`status = hdfan('endaccess',annot_id`	Terminates access to an annotation.

Read/Write Functions

Read/write functions read and write file or object annotations.

Value of `funcstr`	Function Syntax	Description
`'writeann'`	`status = hdfan('writeann',annot_id,`	Writes the annotation corresponding to the given annotation identifier.
`'readann'`	`[annot_string,status] = hdfan('readann',annot_id)`	Reads the annotation corresponding to the given annotation identifier;
	`[annot_string,status] = hdfan('readann',annot_id,m`	Reads the annotation corresponding to the given annotation identifier. `annot_string` will not be longer than `max_str_length`.

General Inquiry Functions

General inquiry functions return information about the annotations in a file.

Value of `funcstr`	Function Syntax	Description
`'numann'`	`num_annot = hdfan('numann',AN_id,annot`	Gets number of annotations of specified type corresponding to given tag/ref pair.
`'annlist'`	`[ann_list,status] = hdfan('annlist',AN_id,anno`	Gets the list of annotations of given type in the file corresponding to a given tag/ref pair.

Value of funcstr	Function Syntax	Description
'annlen'	length = hdfan('annlen',annot_id)	Gets the length of annotation corresponding to the given annotation identifier.
'fileinfo'	[nfl,nfd,ndl,ndd,status] = hdfan('fileinfo',AN_id)	Gets number of file label, file description, data label, and data description annotations in the file corresponding to AN_id.
'get_tagref'	[tag,ref,status] = hdfan('get_tagref',AN_id,i	Gets the tag/ref pair for the specified annotation type and index.
'id2tagref'	[tag,ref,status] = hdfan('id2tagref',annot_id	Gets the tag/ref pair corresponding to the specified annotation identifier.
'tagref2id'	annot_id = hdfan('tagref2id',AN_id,ta	Gets the annotation identifier corresponding to the specified tag/ref pair.
'atype2tag'	tag = hdfan('atype2tag',annot_ty	Gets the tag corresponding to the specified annotation type.
'tag2atype'	annot_type = hdfan('tag2atype',tag)	Gets the annotation type corresponding to the specified tag.

Input/Output Arguments

A status or identifier output of -1 indicates that the operation failed.

In general, the input argument annot_type can be one of these strings:

- 'file_label'
- 'file_desc'
- 'data_label'
- 'data_desc'

AN_id refers to the multifile annotation interface identifier.

annot_id refers to an individual annotation identifier.

You must terminate access to all opened identifiers using either `hdfan('end',AN_id)` or `hdfan('endaccess',annot_id)`. Otherwise, the HDF library might not properly write all data to the file.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfdf24

Gateway to HDF 24-bit raster image (DF24) interface

Syntax

```
[out1,...,outN] = hdfdf24(funcstr,input1,...,inputN)
```

Description

`hdfdf24` is the MATLAB gateway to the HDF 24-bit raster image interface.

`[out1,...,outN] = hdfdf24(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the DF24 function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between DF24 functions in the HDF library and valid values for `funcstr`. For example, `hdfdf24('lastref')` corresponds to the C library call `DF24lastref()`.

Write Functions

Write functions create raster image sets and store them in new files or append them to existing files.

Value of `funcstr`	Function Syntax	Description
`'addimage'`	`status = hdfdf24('addimage',filenam`	Appends a 24-bit raster image to a file.
`'putimage'`	`status = hdfdf24('putimage',filenam`	Writes a 24-bit raster image to file by overwriting all existing data.
`'setcompress'`	`status = hdfdf24('setcompress',comp`	Sets the compress method for the next raster image written to the file. `compress_type` can be `'none'`, `'rle'`, `'jpeg'`, or `'imcomp'`. If

Value of `funcstr`	Function Syntax	Description
		`compress_type` is `'jpeg'`, then two additional parameters must be specified: `quality` (a scalar between 0 and 100) and `force_baseline` (either 0 or 1). Other compression types do not have additional parameters.
`'setdims'`	`status = hdfdf24('setdims',width,he`	Sets the dimensions for the next raster image written to the file.
`'setil'`	`status = hdfdf24('setil',interlace)`	Sets the interlace format of the next raster image written to the file. `interlace` can be `'pixel'`, `'line'`, or `'component'`.
`'lastref'`	`ref = hdfdf24('lastref')`	Reports the last reference number assigned to a 24-bit raster image.

Read Functions

Read functions determine the dimensions and interlace format of an image set, read the actual image data, and provide sequential or random read access to any raster image set.

Value of `funcstr`	Function Syntax	Description
`'getdims'`	`[width,height,interlace,st = hdfdf24('getdims',filename`	Retrieves the dimensions before reading the next raster image. `interlace` can be `'pixel'`, `'line'`, or `'component'`.
`'getimage'`	`[RGB,status] = hdfdf24('getimage',filenam`	Reads the next 24-bit raster image.
`'reqil'`	`status = hdfdf24('reqil',interlace)`	Specifies the interlace format before reading the next raster image. `interlace` can be `'pixel'`, `'line'`, or `'component'`.

Value of `funcstr`	Function Syntax	Description
`'readref'`	`status = hdfdf24('readref',filename`	Reads 24-bit raster image with the specified raster number.
`'restart'`	`status = hdfdf24('restart')`	Returns to the first 24-bit raster image in the file.
`'nimages'`	`num_images = hdfdf24('nimages',filename`	Reports the number of 24-bit raster images in a file.

Input/Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

HDF uses C-style ordering of elements, in which elements along the last dimension vary fastest. MATLAB uses FORTRAN-style ordering, in which elements along the first dimension vary fastest. `hdfdf24` does not automatically convert from C-style ordering to MATLAB style ordering, which means that MATLAB image arrays need to be permuted when using `hdfdf24` to read or write from HDF files. The exact permutation depends on the interlace format specified by, for example, `hdfdf24('setil',...)`. The following calls to `permute` converts HDF arrays to MATLAB arrays, according to the specified interlace format.

```
RGB = permute(RGB,[3 2 1]);
```
`'pixel'` interlace

```
RGB = permute(RGB,[3 1 2]);
```
`'line'` interlace

```
RGB = permute(RGB,[2 1 3]);
```
`'component'` interlace

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfdfr8

Gateway to HDF 8-bit raster image (DFR8) interface

Syntax

```
[out1,...,outN] = hdfdfr8(funcstr,input1,...,inputN)
```

Description

`hdfdfr8` is the MATLAB gateway to the HDF 8-bit raster image (DFR8) interface.

`[out1,...,outN] = hdfdfr8(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the DFR8 function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between DFR8 functions in the HDF library and valid values for `funcstr`. For example, `hdfdfr8('setpalette',map)` corresponds to the C library call `DFR8setpalette(map)`.

Write Functions

Write functions create raster image sets and store them in new files or append them to existing files.

Value of `funcstr`	Function Syntax	Description
`'writeref'`	`status = hdfdfr8('writeref',filenam`	Stores the raster image using the specified reference number.
`'setpalette'`	`status = hdfdfr8('setpalette',color`	Sets palette for multiple 8-bit raster images.
`'addimage'`	`status = hdfdfr8('addimage',filenam`	Appends an 8-bit raster image to a file. `compress` can be `'none'`, `'rle'`, `'jpeg'`, or `'imcomp'`.
`'putimage'`	`status = hdfdfr8('putimage',filenam`	Writes an 8-bit raster image to an existing file or creates the file. `compress` can be `'none'`, `'rle'`, `'jpeg'`, or `'imcomp'`.

Value of funcstr	Function Syntax	Description
'setcompress'	status = hdfdfr8('setcompress',comp	Sets the compression type. compress_type can be 'none', 'rle', 'jpeg', or 'imcomp'. If compress_type is 'jpeg', then two additional parameters must be passed in: quality (a scalar between 0 and 100) and force_baseline (either 0 or 1). Other compression types do not have additional parameters.

Read Functions

Read functions determine the dimension and palette assignment for an image set, read the actual image data, and provide sequential or random read access to any raster image set.

Value of funcstr	Function Syntax	Description
'getdims'	[width,height,hasmap,statu = hdfdfr8('getdims',filename	Retrieves dimensions for an 8-bit raster image.
'getimage'	[X,map,status] = hdfdfr8('getimage',filenam	Retrieves an 8-bit raster image and its palette.
'readref'	status = hdfdfr8('readref',filename	Gets the next raster image with the specified reference number.
'restart'	status = hdfdfr8('restart')	Ignores information about last file accessed and restarts from beginning.
'nimages'	num_images = hdfdfr8('nimages',filename	Returns number of raster images in a file.
'lastref'	ref = hdfdfr8('lastref')	Returns reference number of last element accessed.

Input/Output Arguments

A status or identifier output of -1 indicates that the operation failed.

HDF uses C-style ordering of elements, in which elements along the last dimension vary fastest.MATLAB uses FORTRAN-style ordering, in which elements along the first dimension vary fastest. `hdfdfr8` does not automatically convert from C-style ordering to MATLAB style ordering, which means that MATLAB image and colormap matrices must be transposed when using `hdfdfr8` to read or write from HDF files.

Functions in `hdfdfr8` that read and write palette information expect to use `uint8` data in the range [0,255], while MATLAB colormaps contain double-precision values in the range [0,1]. Therefore, HDF palettes must be converted to `double` and scaled to be used as MATLAB colormaps.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfh

Gateway to HDF H interface

Syntax

```
[out1,...,outN] = hdfh(funcstr,input1,...,inputN)
```

Description

`hdfh` is the MATLAB gateway to the HDF H interface.

`[out1,...,outN] = hdfh(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the H function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between H functions in the HDF library and valid values for `funcstr`. For example, `hdfh('close',file_id)` corresponds to the C library call `Hclose(file_id)`.

Functions

Value of `funcstr`	Function Syntax	Description
`'appendable'`	`status = hdfh('appendable',access_id)`	Specifies that the element can be appended to.
`'close'`	`status = hdfh('close',file_id)`	Closes the access path to the file.
`'deldd'`	`status = hdfh('deldd',file_id,tag,ref)`	Deletes a tag and reference number from the data descriptor list.
`'dupdd'`	`status = hdfh('dupdd',file_id,tag,ref,`	
`'endaccess'`	`status = hdfh('endaccess',access_id)`	Terminates access to a data object by disposing of the access identifier.
`'fidinquire'`	`[filename,access_mode,attach,` `= hdfh('fidinquire',file_id)`	Returns information about specified file.

Value of `funcstr`	Function Syntax	Description
`'find'`	`[tag,ref,offset,length,status = hdfh('find',file_id,... search_tag,search_ref,search_`	Locates the next object to be searched for in an HDF file. `search_type` can be `'new'` or `'continue'`. The `dir` input can be `'forward'` or `'backward'`.
`'getelement'`	`[data,status] = hdfh('getelement',file_id,tag`	Reads the data element for the specified tag and reference number.
`'getfileversio`	`[major,minor,release,info,sta = hdfh('getfileversion',file_id`	Returns version information for an HDF file.
`'getlibversion`	`[major,minor,release,info,sta = hdfh('getlibversion')`	Returns version information for the current HDF library.
`'inquire'`	`[file_id,tag,ref,length,offse special,status] = hdfh('inquire',access_id)`	Returns access information about a data element.
`'ishdf'`	`tf = hdfh('ishdf',filename)`	Determines if a file is an HDF file.
`'length'`	`length = hdfh('length',file_id,tag,ref`	Returns the length of a data object specified by the tag and reference number.
`'newref'`	`ref = hdfh('newref',file_id)`	Returns a reference number that can be used with any tag to product a unique tag/reference number pair.
`'nextread'`	`status = hdfh('nextread',access_id,tag`	Searches for the next data descriptor that matches the specified tag and reference number. `origin` can be `'start'` or `'current'`.
`'number'`	`num = hdfh('number',file_id,tag)`	Returns the number of instances of a tag in a file.

Value of `funcstr`	Function Syntax	Description
`'offset'`	`offset = hdfh('offset',file_id,tag,ref`	Returns the offset of a data element in the file.
`'open'`	`file_id = hdfh('open',filename,access,r`	Provides an access path to an HDF file by reading all the data descriptor blocks into memory.
`'putelement'`	`count = hdfh('putelement',file_id,tag`	Writes a data element or replaces an existing data element in an HDF file. `X` must be a `uint8` array.
`'read'`	`X = hdfh('read',access_id,length)`	Reads the next segment in a data element.
`'seek'`	`status = hdfh('seek',access_id,offset,`	Sets the access pointer to an offset within a data element. `origin` can be `'start'` or `'current'`.
`'startread'`	`access_id = hdfh('startread',file_id,tag,`	
`'startwrite'`	`access_id = hdfh('startwrite',file_id,tag`	
`'sync'`	`status = hdfh('sync',file_id)`	
`'trunc'`	`length = hdfh('trunc',access_id,trunc_`	Truncates the specified data object to the given length.
`'write'`	`count = hdfh('write',access_id,X)`	Writes the next data segment to a specified data element. `X` must be a `uint8` array.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

Limitations

- `hdfh` does not support these functions in the NCSA H interface:
  - `Hcache`
  - `Hendbitaccess`
  - `Hexist`
  - `Hflushdd`
  - `Hgetbit`
  - `Hputbit`
  - `Hsetlength`
  - `Hshutdown`
  - `Htagnewref`

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfhd

Gateway to HDF HD interface

Syntax

```
[out1,...,outN] = hdfhd(funcstr,input1,...,inputN)
```

Description

`hdfhd` is the MATLAB gateway to the HDF HD interface.

`[out1,...,outN] = hdfhd(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the HD function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between HD functions in the HDF library and valid values for `funcstr`.

Functions

Value of `funcstr`	Function Syntax	Description
`'gettagsname'`	`tag_name = hdfhd('gettagsname',tag)`	Gets the name of the specified tag.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfhe

Gateway to HDF HE interface

Syntax

```
[out1,...,outN] = hdfhe(funcstr,input1,...,inputN)
```

Description

`hdfhe` is the MATLAB gateway to the HDF HE interface.

This is a stub page.

`[out1,...,outN] = hdfhe(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the HE function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between HE functions in the HDF library and valid values for `funcstr`.

Functions

Value of `funcstr`	Function Syntax	Description
`'clear'`	`hdfhe('clear')`	Clears all information on reported errors from the error stack.
`'print'`	`hdfhe('print',level)`	Prints information in error stack. If `level` is 0, then the entire error stack is printed.
`'string'`	`error_text = hdfhe('string',error_c`	Returns the error message associated with the specified error code.
`'value'`	`error_code = hdfhe('value',stack_of`	Returns an error code from the specified level of the error stack. A

Value of `funcstr`	Function Syntax	Description
		`stack_offset` value of 1 gets the most recent error code.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

Limitations

- `hdfhe` does not support these functions:
  - `HEpush`
  - `HEreport`

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfhx

Gateway to HDF external data (HX) interface

Syntax

```
[out1,...,outN] = hdfhx(funcstr,input1,...,inputN)
```

Description

`hdfhx` is the MATLAB gateway to the HDF interface for manipulating linked and external data elements.

`[out1,...,outN] = hdfhx(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the HX function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between HX functions in the HDF library and valid values for `funcstr`. For example, `hdfhx('setdir',pathname);` corresponds to the C library call `HXsetdir(pathname)`.

Functions

Value of `funcstr`	Function Syntax	Description
`'create'`	`access_id = hdfhx('create',file_id,ta`	Creates a new external file special data element. `ength)`
`'setcreatedir'`	`status = hdfhx('setcreatedir',path`	Sets the directory location for writing external file.
`'setdir'`	`status = hdfhx('setdir',pathname);`	Sets the directory for locating external files. `pathname` can contain multiple directories separated by vertical bars.

Input/Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

In cases where the HDF C library accepts `NULL` for certain inputs, an empty matrix (`[ ]` or `' '`) can be used.

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfinfo

Information about HDF4 or HDF-EOS file

Syntax

```
S = hdfinfo(filename)
S = hdfinfo(filename,mode)
```

Description

`S = hdfinfo(filename)` returns a structure `S` whose fields contain information about the contents of an HDF4 or HDF-EOS file. `filename` is a string that specifies the name of the HDF4 file.

`S = hdfinfo(filename,mode)` reads the file as an HDF4 file, if `mode` is `'hdf'`, or as an HDF-EOS file, if `mode` is `'eos'`. If `mode` is `'eos'`, only HDF-EOS data objects are queried. To retrieve information on the entire contents of a file containing both HDF4 and HDF-EOS objects, `mode` must be `'hdf'`.

Note `hdfinfo` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To get information about an HDF5 file, use `hdf5info`.

The set of fields in the returned structure `S` depends on the individual file. Fields that can be present in the `S` structure are shown in the following table.

Mode	Field Name	Description	Return Type
HDF	`Attributes`	Attributes of the data set	Structure array
	`Description`	Annotation description	Cell array
	`Filename`	Name of the file	String
	`Label`	Annotation label	Cell array
	`Raster8`	Description of 8-bit raster images	Structure array

Mode	Field Name	Description	Return Type
	`Raster24`	Description of 24-bit raster images	Structure array
	`SDS`	Description of scientific data sets	Structure array
	`Vdata`	Description of Vdata sets	Structure array
	`Vgroup`	Description of Vgroups	Structure array
`EOS`	`Filename`	Name of the file	String
	`Grid`	Grid data	Structure array
	`Point`	Point data	Structure array
	`Swath`	Swath data	Structure array

Those fields in the table above that contain structure arrays are further described in the tables shown below.

Fields Common to Returned Structure Arrays

Structure arrays returned by `hdfinfo` contain some common fields. These are shown in the table below. Not all structure arrays will contain all of these fields.

Field Name	Description	Data Type
`Attributes`	Data set attributes. Contains fields `Name` and `Value`.	Structure array
`Description`	Annotation description	Cell array
`Filename`	Name of the file	String
`Label`	Annotation label	Cell array
`Name`	Name of the data set	String
`Rank`	Number of dimensions of the data set	Double
`Ref`	Data set reference number	Double
`Type`	Type of HDF or HDF-EOS object	String

Fields Specific to Certain Structures

Structure arrays returned by `hdfinfo` also contain fields that are unique to each structure. These are shown in the tables below.

Fields of the Attribute Structure

Field Name	Description	Data Type
`Name`	Attribute name	String
`Value`	Attribute value or description	Numeric or string

Fields of the Raster8 and Raster24 Structures

Field Name	Description	Data Type
`HasPalette`	`1` (`true`) if the image has an associated palette, otherwise `0` (`false`) (8-bit only)	Logical
`Height`	Height of the image, in pixels	Number
`Interlace`	Interlace mode of the image (24-bit only)	String
`Name`	Name of the image	String
`Width`	Width of the image, in pixels	Number

Fields of the SDS Structure

Field Name	Description	Data Type
`DataType`	Data precision	String
`Dims`	Dimensions of the data set. Contains fields `Name`, `DataType`, `Size`, `Scale`, and `Attributes`. `Scale` is an array of numbers to place along the dimension and demarcate intervals in the data set.	Structure array
`Index`	Index of the SDS	Number

Fields of the Vdata Structure

Field Name	Description	Data Type
`DataAttributes`	Attributes of the entire data set. Contains fields `Name` and `Value`.	Structure array
`Class`	Class name of the data set	String

Field Name	Description	Data Type
`Fields`	Fields of the Vdata. Contains fields `Name` and `Attributes`.	Structure array
`NumRecords`	Number of data set records	Double
`IsAttribute`	`1` (`true`) if Vdata is an attribute, otherwise `0` (`false`)	Logical

Fields of the Vgroup Structure

Field Name	Description	Data Type
`Class`	Class name of the data set	String
`Raster8`	Description of the 8-bit raster image	Structure array
`Raster24`	Description of the 24-bit raster image	Structure array
`SDS`	Description of the Scientific Data sets	Structure array
`Tag`	Tag of this Vgroup	Number
`Vdata`	Description of the Vdata sets	Structure array
`Vgroup`	Description of the Vgroups	Structure array

Fields of the Grid Structure

Field Name	Description	Data Type
`Columns`	Number of columns in the grid	Number
`DataFields`	Description of the data fields in each `Grid` field of the grid. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, `FillValue`, and `TileDims`.	Structure array
`LowerRight`	Lower right corner location, in meters	Number
`Origin Code`	Origin code for the grid	Number
`PixRegCode`	Pixel registration code	Number
`Projection`	Projection code, zone code, sphere code, and projection parameters of	Structure

Field Name	Description	Data Type
	the grid. Contains fields `ProjCode`, `ZoneCode`, `SphereCode`, and `ProjParam`.	
`Rows`	Number of rows in the grid	Number
`UpperLeft`	Upper left corner location, in meters	Number

Fields of the Point Structure

Field Name	Description	Data Type
`Level`	Description of each level of the point. Contains fields `Name`, `NumRecords`, `FieldNames`, `DataType`, and `Index`.	Structure

Fields of the Swath Structure

Field Name	Description	Data Type
`DataFields`	Data fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`GeolocationField`	Geolocation fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`IdxMapInfo`	Relationship between indexed elements of the geolocation mapping. Contains fields `Map` and `Size`.	Structure
`MapInfo`	Relationship between data and geolocation fields. Contains fields `Map`, `Offset`, and `Increment`.	Structure

Examples

To retrieve information about the file `example.hdf`,

```
fileinfo = hdfinfo('example.hdf')
```

```
fileinfo =
    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

And to retrieve information from this about the scientific data set in `example.hdf`,

```
sds_info = fileinfo.SDS

sds_info =
       Filename: 'example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
           Dims: [2x1 struct]
          Label: {}
    Description: {}
          Index: 0
```

See Also

`hdfread`

Introduced before R2006a

hdfml

Utilities for working with MATLAB HDF gateway functions

Syntax

```
hdfml('closeall')
hdfml('listinfo')
tag = hdfml('tagnum',tagname)
nbytes = hdfml('sizeof',data_type)
hdfml('defaultchartype',char_type)
```

Description

`hdfml('closeall')` closes all open registered HDF file and data object identifiers.

`hdfml('listinfo')` prints information about all open registered HDF file and data object identifiers.

`tag = hdfml('tagnum',tagname)` returns the tag number corresponding to the tag name specified by `tagname`.

`nbytes = hdfml('sizeof',data_type)` returns size in bytes of specified data type.

`hdfml('defaultchartype',char_type)` defines the HDF data type for MATLAB strings. Valid values for `char_type` are `'char8'` or `'uchar8'`. The change persists until the MATLAB HDF gateway function is cleared from memory. MATLAB strings are mapped to `char8` by default.

The MATLAB HDF gateway functions maintain lists of certain HDF file and data object identifiers so that, for example, HDF objects and files can be properly closed when a user issues the command:

```
clear mex
```

These lists are updated whenever these identifiers are created or closed.

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfpt

Interface to HDF-EOS Point object

Syntax

```
[out1,...,outN] = hdfpt(funcstr,input1,...,inputN)
```

Description

`hdfpt` is the MATLAB gateway to the Point functions in the HDF-EOS C library, which is developed and maintained by EOSDIS (Earth Observing System Data and Information System). A Point data set comprises a series of data records taken at (possibly) irregular time intervals and at scattered geographic locations. Each data record consists of a set of one or more data values representing the state of a point in time and/or space.

`[out1,...,outN] = hdfpt(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the Point function in the HDF-EOS library specified by `functstr`.

There is a one-to-one correspondence between PT functions in the HDF-EOS C library and valid values for `funcstr`. For example, `hdfpt('detach',point_id)` corresponds to the C library call `PTdetach(point_id)`.

Programming Model

The programming model for accessing a point data set through `hdfpt` is as follows:

1. Open the file and initialize the PT interface by obtaining a file id from a file name.
2. Open or create a point data set by obtaining a point id from a point name.
3. Perform desired operations on the data set.
4. Close the point data set by disposing of the point id.
5. Terminate point access to the file by disposing of the file id.

To access a single point data set that already exists in an HDF-EOS file, use the following MATLAB commands:

```
fileid = hdfpt('open',filename,access);
pointid = hdfpt('attach',fileid,pointname);

% Optional operations on the data set...

status = hdfpt('detach',pointid);
status = hdfpt('close',fileid);
```

To access several files at the same time, obtain a separate file identifier for each file to be opened. To access more than one point data set, obtain a separate point id for each data set.

It is important to properly dispose of point id's and file id's so that buffered operations are written completely to disk. If you quit MATLAB or clear all MEX-files with PT identifiers still open, MATLAB issues a warning and automatically disposes of them.

Note that file identifiers returned by `hdfpt` are not interchangeable with file identifiers returned by any other HDF-EOS or HDF function.

Access Routines

Access routines initialize and terminate access to the PT interface and point data sets (including opening and closing files).

Value of `funcstr`	Function Syntax	Description
`'open'`	`file_id = hdfpt('open',filename,acce`	Given the filename and desired access mode, opens or creates an HDF file in order to create, read, or write a point. `access` can be `'read'`, `'readwrite'`, or `'create'`. `file_id` is -1 if the operation fails.
`'create'`	`point_id = hdfpt('create',file_id,poi`	Creates a point data set with the specified name. `pointname` is a string contain the name of the point data set. `point_id` is -1 if the operation fails.
`'attach'`	`point_id = hdfpt('attach',file_id,poi`	Attaches to an existing point data set within the file.

Value of `funcstr`	Function Syntax	Description
		`point_id` is -1 if the operation fails.
`'detach'`	`status = hdfpt('detach',point_id)`	Detaches from point data set.
`'close'`	`status = hdfpt('close',file_id)`	Closes file.

Definition Routines

Definition routines allow the user to set key features of a point data set.

Value of `funcstr`	Function Syntax	Description
`'deflevel'`	`status = hdfpt('deflevel',point_id, fieldList,fieldTypes,field`	Defines a new level within a point data set. `levelname` is the name of the level to be defined. `fieldList` is a string containing a comma-separated list of field names in the new level. `fieldTypes` is a cell array containing the number type string for each field. Valid number type strings include `'uchar8'`, `'uchar'`, `'char8'`, `'char'`, `'double'`, `'uint8'`, `'uint16'`, `'uint32'`, `'float'`, `'int8'`, `'int16'`, and `'int32'`. `fieldOrders` is a vector containing the order for each field.
`'deflinkage'`	`status = hdfpt('deflinkage',point_i`	Defines a linkfield between two adjacent levels. `parent` is the name of the parent level. `child` is the name of the child level. `linkfield` is the name of a field that is defined at both levels.

Basic I/O Routines

Basic I/O routines read and write data and metadata to a point data set.

Value of `funcstr`	Function Syntax	Description
`'writelevel'`	`status = hdfpt('writelevel',point_i`	Appends new records to the specified level in a point data set. `level` is the desired level index (zero-based). `data` must be a `P`-by-1 cell array where `P` is the number of fields defined for the specified level. Each cell of `data` must contain an `M(k)`-by-`N` matrix of data, where `M(k)` is the order of the `k`-th field (the number of scalar values in the field) and `N` is the number of records. The MATLAB class of the cells must match the HDF data type defined for the corresponding fields. A MATLAB string is automatically converted to match any of the HDF char types. Other data types must match exactly.
`'readlevel'`	`[data,status] = hdfpt('readlevel',point_id level,fieldList,records)`	Reads data from a given level in a point data set. `level` is the index (zero-based) of the desired level. `fieldList` is a string containing a comma-separated list of the fields to read. `records` is a vector containing the indices (zero-based) of the records to read. `data` is a `P`-by-1 cell array where `P` is the number of requested fields. Each cell of `data` contains an `M(k)`-by-`N` matrix of data where `M(k)` is the order of the `k`-

Value of `funcstr`	Function Syntax	Description
		th field and `N` is the number of records, or `length(records)`.
`'updatelevel'`	`status = hdfpt('updatelevel',point_ level,fieldList,records,da`	Updates (corrects) data in a particular level of a point data set. `level` is the index (zero-based) of the desired level. `fieldList` is a string containing a comma-separated list of field names to update. `records` is a vector containing the indices (zero-based) of the records to update. `data` is a `P`-by-1 cell array where `P` is the number of specified fields. Each cell of `data` must contain an `M(k)`-by-`N` matrix of data, where `M(k)` is the order of the `k`-th field (the number of scalar values in the field) and `N` is the number of records, or `length(records)`. The MATLAB class of the cells must match the HDF data type defined for the corresponding fields. A MATLAB string is automatically converted to match any of the HDF char types. Other data types must match exactly.
`'writeattr'`	`status = hdfpt('writeattr',point_id`	Writes or updates the point data set attribute with the specified name. If the attribute does not already exist, it is created.
`'readattr'`	`[data,status] = hdfpt('readattr',point_id,`	Reads the attribute data from the specified attribute.

Inquiry Routines

Inquiry routines return information about data contained in a point data set.

Value of `funcstr`	Function Syntax	Description
`'nlevels'`	`nlevels = hdfpt('nlevels',point_id)`	Returns the number of levels in a point data set. `nlevels` is -1 if the operation fails.
`'nrecs'`	`nrecs = hdfpt('nrecs',point_id,lev`	Returns the number of records in the specified level. `nrecs` is -1 if the operation fails.
`'nfields'`	`[numfields,strbufsize] = hdfpt('nfields',point_id,l`	Returns the number of fields in the specified level. `strbufsize` is the length of the comma-separated field name string. `numfields` is -1 and `strbufsize` is `[]` if the operation fails.
`'levelinfo'`	`[numfields,fieldList,field Type,fieldOrder] = ... hdfpt('levelinfo',point_id`	Returns information on fields for a specified level. `fieldList` is a string containing a comma-separated list of field names. `fieldType` is a cell array of strings that defined the data type for each field. `fieldOrder` is a vector containing the order (number of scalar values) associated with each field. If the operation fails, `numfields` is -1 and the other outputs are empty.
`'levelindx'`	`level = hdfpt('levelindx',point_id`	Returns the level index (zero-based) of the level with the specified name. `level` is -1 if the operation fails.
`'bcklinkinfo'`	`[linkfield,status] = hdfpt('bcklinkinfo',point_`	Returns the linkfield to the previous level. `status` is -1 and `linkfield` is `[]` if the operation fails.

Value of `funcstr`	Function Syntax	Description
`'fwdlinkinfo'`	`[linkfield,status] = hdfpt('fwdlinkinfo',point_`	Returns the linkfield to the following level. `status` is -1 and `linkfield` is `[]` if the operation fails.
`'getlevelname'`	`[levelname,status] = hdfpt('getlevelname',point`	Returns the name of a level given the level index. `status` is -1 and `levelname` is `[]` if the operation fails.
`'sizeof'`	`[byteSize,fieldLevels] = hdfpt('sizeof',point_id,fi`	Returns the size in bytes and field levels of the specified fields. `fieldList` is a string containing a comma-separated list of field names. `byteSize` is the total size of bytes of the specified fields, and `fieldLevels` is a vector containing the level index corresponding to each field. `byteSize` is -1 and `fieldLevels` is `[]` if the operation fails.
`'attrinfo'`	`[numberType,count,status] = ... hdfpt('attrinfo',point_id,`	Returns the number type and size in bytes of the specified attribute. `attrname` is the name of the attribute. `numberType` is a string corresponding to the HDF data type of the attribute. `count` is the number of bytes used by the attribute data. `status` is -1 and `numberType` and `count` are `[]` if the operation fails.

Value of `funcstr`	Function Syntax	Description
`'inqattrs'`	`[nattrs,attrnames] = hdfpt('inqattrs',point_id)`	Retrieve information about attributes defined in a point data set. `nattrs` and `attrnames` are the number and names of all the defined attributes, respectively. If the operation fails, `nattrs` is -1 and `attrnames` is `[]`.
`'inqpoint'`	`[numpoints,pointnames] = hdfpt('inqpoint',filename)`	Retrieve number and names of point data sets defined in an HDF-EOS file. `pointnames` is a string containing a comma-separated list of point names. `numpoints` is -1 and `pointnames` is `[]` if the operation fails.

Utility Routines

Placeholder.

Value of `funcstr`	Function Syntax	Description
`'getrecnums'`	`[outRecords,status] = hdfpt('getrecnums',... point_id,inLevel,outLevel,`	Returns the record numbers in `outLevel` corresponding the group of records specified by `inRecords` in level `inLevel`. The `inLevel` and `outLevel` arguments are zero-based level indices. `inRecords` is a vector of zero-based record indices. `status` is -1 and `outRecords` is `[]` if the operation fails.

Subset Routines

Subset routines allow reading of data from a specified geographic region.

Value of `funcstr`	Function Syntax	Description
`'defboxregion'`	`region_id = hdfpt('defboxregion',point`	Defines a longitude-latitude box region for a point. `cornerLon` is a two-element vector containing the longitudes of opposite box corners. `cornerLat` is a two-element vector containing the latitudes of opposite box corners. `region_id`is -1 if the operation fails.
`'defvrtregion'`	`period_id = hdfpt('defvrtregion',point vert_field,range)`	Defines a vertical region for a point. `vert_field` is the name of the field to subset. `range` is a two-element vector containing the minimum and maximum vertical values. `period_id` is -1 if the operation fails.
`'regioninfo'`	`[byteSize,status] = hdfpt('regioninfo',point_i region_id,level,fieldList)`	Returns the data size in bytes of the subset period of the specified level. `fieldlist` is a string containing a comma-separated list of fields to extract. `status` and `byteSize` are -1 if the operation fails.
`'regionrecs'`	`[numRec,recNumbers,status] = hdfpt('regionrecs',... point_id,region_id,level)`	Returns the records numbers within the subsetted region of the specified level. `status` and `numrec` are -1 and `recNumbers` is `[ ]` if the operation fails.

Value of `funcstr`	Function Syntax	Description
`'extractregion'`	`[data,status] = hdfpt('extractregion',poin region_id,level,fieldList)`	Reads data from the specified subset region. `fieldList` is a string containing a comma-separated list of requested fields. `data` is a `P`-by-1 cell array where `P` is the number of requested fields. Each cell of `data` contains an `M(k)`-by-`N` matrix of data where `M(k)` is the order of the `k`-th field and `N` is the number of records. `status` is -1 and `data` is `[]` if the operation fails.
`'deftimeperiod'`	`period_id = hdfpt('deftimeperiod',poin`	Defines a time period for a point data set. `period_id` is -1 if the operation fails.
`'periodinfo'`	`[byteSize,status] = hdfpt('periodinfo',point_i period_id,level,fieldList)`	Retrieves the size in bytes of the subsetted period. `fieldList` is string containing a comma-separated list of desired field names. `byteSize` and `status` are -1 if the operation fails.
`'periodrecs'`	`[numRec,recNumbers,status] = hdfpt('periodrecs',... point_id,period_id,level)`	Returns the records numbers within the subsetted time period of the specified level. `numRec` and `status` are -1 if the operation fails.

Value of `funcstr`	Function Syntax	Description
`'extractperiod'`	`[data,status] = hdfpt('extractperiod',... point_id,period_id,level,f`	Reads data from the specified subsetted time period. `fieldList` is a string containing a comma-separated list of requested fields. `data` is a `P`-by-1 cell array where `P` is the number of requested fields. Each cell of `data` contains an `M(k)`-by-`N` matrix of data where `M(k)` is the order of the `k`-th field and `N` is the number of records. `status` is -1 and `data` is `[]` if the operation fails.

Input/Output Arguments

Most routines return the flag, `status`, which is 0 when the routine succeeds and -1 when the routine fails. Routines with syntaxes which do not contain `status` return failure information in one of its outputs as notated in the function syntaxes.

`levelName` is a string.

Some of the C library functions accept input values that are defined in terms of C macros. For example, the C `PTopen()` function requires an access mode input that can be DFACC_READ, DFACC_RDWR, or DFACC_CREATE, where these symbols are defined in the appropriate C header file. Where macro definitions are used in the C library, the equivalent MATLAB syntaxes use strings derived from the macro names. You can either use a string containing the entire macro name, or you can omit the common prefix. You can use either upper or lower case. For example, this C function call:

```
status = PTopen("PointFile.hdf",DFACC_CREATE)
```
is equivalent to these MATLAB function calls:

```
status = hdfpt('open','PointFile.hdf','DFACC_CREATE')
status = hdfpt('open','PointFile.hdf','dfacc_create')
status = hdfpt('open','PointFile.hdf','CREATE')
status = hdfpt('open','PointFile.hdf','create')
```

In cases where a C function returns a value with a macro definition, the equivalent MATLAB function returns the value as a string containing the lowercase short form of the macro.

HDF number types are specified by strings, including `'uchar8'`, `'uchar'`, `'char8'`, `'char'`, `'double'`, `'uint8'`, `'uint16'`, `'uint32'`, `'float'`, `'int8'`, `'int16'`, and `'int32'`.

In cases where the HDF-EOS library accepts `NULL`, use an empty matrix (`[ ]`).

See Also

`matlab.io.hdfeos.sw`

hdfread

Read data from HDF4 or HDF-EOS file

Syntax

```
data = hdfread(filename, datasetname)
data = hdfread(hinfo)
data = hdfread(...,param,value,...)
data = hdfread(filename,EOSname,param,value,...)
[data,map] = hdfread(...)
```

Description

`data = hdfread(filename, datasetname)` returns all the data in the data set specified by `datasetname` from the HDF4 or HDF-EOS file specified by `filename`. To determine the name of a data set in an HDF4 file, use the `hdfinfo` function.

Note `hdfread` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To read data from an HDF5 file, use `h5read`.

`data = hdfread(hinfo)` returns all the data in the data set specified by the structure`hinfo`, returned by the `hdfinfo` function. Specify the field in the `hinfo` structure that relates to a particular type of data set, and use indexing to specify which data set, when there are more than one. See “Specify data set to read” on page 1-3543 for more information.

`data = hdfread(...,param,value,...)` returns subsets of the data according to the specified parameter and value pairs. See the tables below to find the valid parameters and values for different types of data sets.

`data = hdfread(filename,EOSname,param,value,...)` subsets the data field from the HDF-EOS point, grid, or swath specified by `EOSname`.

`[data,map] = hdfread(...)` returns the image `data` and the colormap `map` for an 8-bit raster image.

Subsetting Parameters

The following tables show the subsetting parameters that can be used with the `hdfread` function for certain types of HDF4 data. These data types are

- HDF Scientific Data (SD)
- HDF Vdata (V)
- HDF-EOS Grid Data
- HDF-EOS Point Data
- HDF-EOS Swath Data

Note the following:

- If a parameter requires multiple values, use a cell array to store the values. For example, the `'Index'` parameter requires three values: `start`, `stride`, and `edge`. Enclose these values in curly braces as a cell array.

  ```
  hdfread(..., 'Index', {start,stride,edge})
  ```
- All values that are indices are 1-based.

Subsetting Parameters for HDF Scientific Data (SD) Data Sets

When you are working with HDF SD files, `hdfread` supports the parameters listed in this table.

Parameter	Description
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — A 1-based array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set. • `stride` — A 1-based array specifying the interval between the values to read Default: `1`, read every element of the data set.

Parameter	Description
	• `edge` — A 1-based array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions

For example, this code reads the data set `Example SDS` from the HDF file `example.hdf`. The `'Index'` parameter specifies that `hdfread` start reading data at the beginning of each dimension, read until the end of each dimension, but only read every other data value in the first dimension.

```
data = hdfread('example.hdf','Example SDS','Index',{[],[2 1],[]})
```

Subsetting Parameters for HDF Vdata Sets

When you are working with HDF Vdata files, `hdfread` supports these parameters.

Parameter	Description
`'Fields'`	Text string specifying the name of the field to be read. When specifying multiple field names, use a cell array.
`'FirstRecord'`	1-based number specifying the record from which to begin reading
`'NumRecords'`	Number specifying the total number of records to read

For example, this code reads the Vdata set `Example Vdata` from the HDF file `example.hdf`.

```
data = hdfread('example.hdf','Example Vdata','FirstRecord', 2,'NumRecords', 5)
```

Subsetting Parameters for HDF-EOS Grid Data

When you are working with HDF-EOS grid data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive parameters — You can only specify one of these parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	

Parameter	Description
`'Fields'`	String specifying the field to be read. You can specify only one field name for a Grid data set.
Mutually Exclusive Optional Parameters	
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Interpolate'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude points that define a region for bilinear interpolation. Each element is an N-length vector specifying longitude and latitude coordinates.
`'Pixels'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. Each element is an N-length vector specifying longitude and latitude coordinates. This region is converted into pixel rows and columns with the origin in the upper left corner of the grid. Note: This is the pixel equivalent of reading a `'Box'` region.
`'Tile'`	Vector specifying the coordinates of the tile to read, for HDF-EOS Grid files that support tiles
Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.

Parameter	Description
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and end-point for a period of time
`'Vertical'`	Two-element cell array, `{dimension, range}` `dimension` — String specifying the name of the data set field to be read from. You can specify only one field name for a Grid data set. `range` — Two-element array specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
data = hdfread('grid.hdf','PolarGrid','Fields','ice_temp','Index', {[5 10],[],[15 20]})
```

Subsetting Parameters for HDF-EOS Point Data

When you are working with HDF-EOS Point data, `hdfread` has two required parameters and three optional parameters.

Parameter	Description
Required Parameters	
`'Fields'`	String naming the data set field to be read. For multiple field names, use a comma-separated list.
`'Level'`	1-based number specifying which level to read from in an HDF-EOS Point data set
Mutually Exclusive Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.

Parameter	Description
`'RecordNumbers'`	Vector specifying the record numbers to read
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and endpoint for a period of time

For example,

```
hdfread(...,'Fields',{field1, field2},...
            'Level',level,'RecordNumbers',[1:50, 200:250])
```

Subsetting Parameters for HDF-EOS Swath Data

When you are working with HDF-EOS Swath data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive

You can only use one of the mutually exclusive parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	String naming the data set field to be read. You can specify only one field name for a Swath data set.
Mutually Exclusive Optional Parameters	
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. • `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — An array specifying the length of each dimension to read

Parameter	Description
	Default: An array containing the lengths of the corresponding dimensions
`'Time'`	Three-element cell array, `{start, stop, mode}`, where `start` and `stop` specify the beginning and the endpoint for a period of time, and `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`).
Optional Parameters	
`'Box'`	Three-element cell array, `{longitude, latitude, mode}` specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are two-element vectors that specify longitude and latitude coordinates. `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`). • Any point is within the box (`mode='anypoint'`).
`'Vertical'`	Two-element cell array, `{dimension, range}` • `dimension` is a string specifying either a dimension name or field name to subset the data by. • `range` is a two-element vector specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
hdfread('swath.hdf', 'Example Swath', 'Fields', 'Temperature', ...
         'Time', {5000, 6000, 'midpoint'})
```

Examples

Read Data Set in HDF File

Specify the name of the HDF file and the name of the data set. This example reads a data set named `temperature` from a sample HDF file.

```
data = hdfread('sd.hdf','temperature');
```

Specify data set to read

Call `hdfinfo` to retrieve information about the contents of the HDF file.

```
fileinfo = hdfinfo('sd.hdf')
```

```
fileinfo =

      Filename: 'B:\matlab\toolbox\matlab\imagesci\sd.hdf'
    Attributes: [1x1 struct]
           SDS: [1x2 struct]
         Vdata: [1x1 struct]
```

Extract the structure containing information about the particular data set you want to import from the data returned by `hdfinfo`. This example uses the structure in the SDS field to retrieve a scientific data set.

```
sds_info = fileinfo.SDS(2)
```

```
sds_info =

       Filename: 'B:\matlab\toolbox\matlab\imagesci\sd.hdf'
           Type: 'Scientific Data Set'
           Name: 'temperature'
           Rank: 2
       DataType: 'double'
     Attributes: [1x11 struct]
           Dims: [2x1 struct]
          Label: {}
```

```
    Description: {}
          Index: 1
```

Pass this structure to `hdfread` to import the data in the data set.

```
data = hdfread(sds_info);
```

Read Data from HDF-EOS Grid Field

Read data from the HDF-EOS global grid field, `TbOceanRain`, in the example file, `example.hdf`.

```
data1 = hdfread('example.hdf','MonthlyRain','Fields','TbOceanRain');
```

Read data for the northern hemisphere for the same field. Use the `Box` parameter to specify the longitude and latitude coordinates for that region.

```
data2 = hdfread('example.hdf','MonthlyRain', ...
'Fields','TbOceanRain', ...
'Box', {[0 360],[0 90]});
```

Read Subset of Data in Data Set

Retrieve info about the example file, `example.hdf`.

```
fileinfo = hdfinfo('example.hdf');
```

Retrieve information about Scientific Data Set in `example.hdf`.

```
data_set_info = fileinfo.SDS;
```

Check the size of the data set.

```
data_set_info.Dims.Size
```

```
ans =

    16


ans =

     5
```

Read a subset of the data in the data set using the `'index'` parameter with `hdfread`. This example specifies a starting index of `[3 3]`, an interval of 1 between values (`[]` meaning the default value of 1), and a length of 10 rows and 2 columns.

```
data = hdfread(data_set_info,'Index',{[3 3],[],[10 2]});
data(:,1)
```

```
ans =

     7
     8
     9
    10
    11
    12
    13
    14
    15
    16
```

```
data(:,2)
```

```
ans =

     8
     9
    10
    11
    12
    13
    14
    15
    16
    17
```

Access Data in Fields of Vdata

Use the `Vdata` field from the information returned by `hdfinfo` to read three fields of the data, `Idx`, `Temp`, and `Dewpt`.

```
s = hdfinfo('example.hdf');
data = hdfread(s.Vdata(1),'Fields',{'Idx','Temp','Dewpt'})


data =

    [1x10 int16]
    [1x10 int16]
    [1x10 int16]
```

See Also

`hdfinfo`

Introduced before R2006a

hdftool

Browse and import data from HDF4 or HDF-EOS files

Syntax

```
hdftool
hdftool(filename)
h = hdftool( ___ )
```

Description

`hdftool` starts the HDF Import Tool, a graphical user interface used to browse the contents of HDF4 and HDF-EOS files and import data and subsets of data from these files. To open an HDF4 or HDF-EOS file, select **Open** from the **Home** tab. You can open multiple files in the HDF Import Tool by selecting **Open** from the **Home** tab.

`hdftool(filename)` opens the HDF4 or HDF-EOS file specified by `filename` in the HDF Import Tool.

`h = hdftool( ___ )` returns a handle `h` to the HDF Import Tool. To close the tool from the command line, use `close(h)`.

Examples

```
hdftool('example.hdf');
```

See Also

`hdfinfo` | `hdfread` | `uiimport`

Introduced before R2006a

hdfv

Gateway to HDF Vgroup (V) interface

Syntax

```
[out1,...,outN] = hdfv(funcstr,input1,...,inputN)
```

Description

`hdfv` is the MATLAB gateway to the HDF Vgroup (V) interface.

`[out1,...,outN] = hdfv(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the V function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between V functions in the HDF library and valid values for `funcstr`. For example, `hdfv('nattrs',vgroup_id)` corresponds to the C library call `Vnattrs(vgroup_id)`.

Access Functions

Access functions open files, initialize the Vgroup interface, and access individual groups. They also terminate access to vgroups and the Vgroup interface and close HDF files.

Value of `funcstr`	Function Syntax	Description
`'start'`	`status = hdfv('start',file_id)`	Initializes the V interface.
`'attach'`	`vgroup_id = hdfv('attach',file_id,vgr`	Establishes access to a vgroup. `access` can be `'r'` or `'w'`.
`'detach'`	`status = hdfv('detach',vgroup_id)`	Terminates access to a vgroup.
`'end'`	`status = hdfv('end',file_id)`	Terminates access to the V interface.

Create Functions

Create functions organize, label, and add data objects to vgroups.

Value of `funcstr`	Function Syntax	Description
`'setclass'`	`status = hdfv('setclass',vgroup_id`	Assigns a class to a vgroup.
`'setname'`	`status = hdfv('setname',vgroup_id,`	Assigns a name to a vgroup.
`'insert'`	`ref = hdfv('insert',vgroup_id, id)`	Adds a vgroup or vdata to an existing group. `id` can be a vdata id or a vgroup id.
`'addtagref'`	`status = hdfv('addtagref',vgroup_i`	Adds any HDF data object to an existing vgroup.
`'setattr'`	`status = hdfv('setattr',vgroup_id,`	Sets the attribute of a vgroup.

File Inquiry Functions

File inquiry functions return information about how vgroups are stored in a file. They are useful for locating vgroups in a file.

Value of `funcstr`	Function Syntax	Description
`'lone'`	`[refs,count] = hdfv('lone',file_id,maxsi`	Returns the reference numbers of vgroups not included in other vgroups.
`'getid'`	`next_ref = hdfv('getid',file_id,vgro`	Returns the reference number for the next vgroup in the HDF file.
`'find'`	`vgroup_ref = hdfv('find',file_id,vgrou`	Returns the reference number of the vgroup with the specified name if successful and zero otherwise.
`'findclass'`	`vgroup_ref = hdfv('findclass',file_id,`	Returns the reference number of the vgroup with the specified class.

Vgroup Inquiry Functions

Vgroup inquiry functions provide specific information about a specific vgroup. This information includes the class, name, member count, and additional member information.

Value of `funcstr`	Function Syntax	Description
`'getclass'`	`[class_name,status] = hdfv('getclass',vgroup_id`	Returns the name of the class of the specified group.
`'getname'`	`[vgroup_name,status] = hdfv('getname',vgroup_id)`	Returns the name of the specified group.
`'inquire'`	`[num_entries,name,status] = hdfv('inquire',vgroup_id)`	Returns the number of entries and the name of a vgroup.
`'isvg'`	`status = hdfv('isvg',vgroup_id,ref`	Checks if the object specified by ref refers to a child vgroup of the vgroup specified by `vgroup_id`.
`'isvs'`	`status = hdfv('isvs',vgroup_id,vda`	Checks if the object specified by `vdata_ref` refers to a child vdata of the vgroup specified by `vgroup_id`.
`'gettagref'`	`[tag,ref,status] = hdfv('gettagref',vgroup_i`	Retrieves a tag/reference number pair for a data object in the specified vgroup.
`'ntagrefs'`	`count = hdfv('ntagrefs',vgroup_id`	Returns the number of tag/reference number pairs contained in the specified vgroup.
`'gettagrefs'`	`[tag,refs,count] = hdfv('gettagrefs',vgroup_`	Retrieves the tag/reference pairs of all the data objects within a vgroup.
`'inqtagref'`	`tf = hdfv('inqtagref',vgroup_i`	Checks if an object belongs to a vgroup.
`'getversion'`	`version = hdfv('getversion',vgroup_`	Queries the vgroup version of a given vgroup.

Value of `funcstr`	Function Syntax	Description
`'nattrs'`	`count = hdfv('nattrs',vgroup_id)`	Queries the total number of vgroup attributes.
`'attrinfo'`	`[name,data_type,count,nby = hdfv('attrinfo',vgroup_id attr_index)`	Queries information on a given vgroup attribute.
`'getattr'`	`[values,status] = hdfv('getattr',vgroup_id,`	Queries the values of a given attribute.
`'Queryref'`	`ref = hdfv('Queryref',vgroup_id`	Returns the reference number of the specified vgroup.
`'Querytag'`	`tag = hdfv('Querytag',vgroup_id`	Returns the tag of the specified vgroup.
`'flocate'`	`vdata_ref = hdfv('flocate',vgroup_id,`	Returns the reference number of the vdata containing the specified field name in the specified vgroup.
`'nrefs'`	`count = hdfv('nrefs',vgroup_id,ta`	Returns the number of data objects with the specified tag in the specified vgroup.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfvf

Gateway to VF functions in HDF Vdata interface

Syntax

```
[out1,...,outN] = hdfvf(funcstr,input1,...,inputN)
```

Description

`hdfvf` is the MATLAB gateway to the VF unctions in the HDF Vdata interface.

`[out1,...,outN] = hdfvf(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the VF function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between VF functions in the HDF library and valid values for `funcstr`. For example, `hdfvf('nfields',vdata_id)` corresponds to the C library call `VFnfields(vdata_id)`.

Field Inquiry Functions

Field inquiry functions provide specific information about the fields in a given vdata, including the field's size, name, order, type, and number of fields in the vdata.

Value of `funcstr`	Function Syntax	Description
`'fieldesize'`	`fsize = hdfvf('fieldesize',vdata_id`	Retrieves the field size (as stored in a file) of a specified field.
`'fieldisize'`	`fsize = hdfvf('fieldisize',vdata_id`	Retrieves the field size (as stored in memory) of a specified field.
`'fieldname'`	`name = hdfvf('fieldname',vdata_id,`	Retrieves the name of the specified field in the given vdata.
`'fieldorder'`	`order = hdfvf('fieldorder',vdata_id`	Retrieves the order of the specified field in the given vdata.

Value of `funcstr`	Function Syntax	Description
`'fieldtype'`	`data_type = hdfvf('fieldtype',vdata_id,`	Retrieves the data type for the specified field in the given vdata.
`'nfields'`	`count = hdfvf('nfields',vdata_id)`	Retrieves the total number of fields in the specified vdata.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfvh

Gateway to VH functions in HDF Vdata interface

Syntax

```
[out1,...,outN] = hdfvh(funcstr,input1,...,inputN)
```

Description

`hdfvh` is the MATLAB gateway to VH functions in the HDF Vdata interface.

`[out1,...,outN] = hdfvh(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the VH function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between VH functions in the HDF library and valid values for `funcstr`.

High-level Vdata Functions

High-level Vdata functions write data to single-field vdatas.

Value of `funcstr`	Function Syntax	Description
`'makegroup'`	`vgroup_ref = hdfvh('makegroup',file_id, vgroup_name,vgroup_class)`	Groups a collection of data objects within a vgroup.
`'storedata'`	`count = hdfvh('storedata',file_id, vdata_name,vdata_class)`	Creates vdatas containing records limited to one field with one component per field.
`'storedatam'`	`count = hdfvh('storedatam',file_id vdata_name,vdata_class)`	Creates vdatas containing records with one field containing one or more components.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

hdfdf24 | hdfdfr8 | hdfh | hdfhd | hdfhe | hdfhx | hdfml | hdfv | hdfvf | hdfvh | hdfvs | matlab.io.hdf4.sd

hdfvs

Gateway to VS functions in HDF Vdata interface

Syntax

```
[out1,...,outN] = hdfvs(funcstr,input1,...,inputN)
```

Description

`hdfvs` is the MATLAB gateway to the VS functions in the HDF Vdata interface.

`[out1,...,outN] = hdfvs(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the VS function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between VS functions in the HDF library and valid values for `funcstr`. For example, `hdfvs('detach',vdata_id)` corresponds to the C library call `VSdetach(vdata_id)`.

Access Functions

Access functions attach, or allow access, to vdatas. Data transfer can only occur after a vdata has been accessed. These routines also detach from, or properly terminate access to, vdatas when data transfer has been completed.

Value of `funcstr`	Function Syntax	Description
`'attach'`	`vdata_id = hdfvs('attach',file_id,vd`	Establishes access to a specified vdata. `access` can be `'r'` or `'w'`.
`'detach'`	`status = hdfvs('detach',vdata_id)`	Terminates access to a specified vdata.

Read and Write Functions

Read and write functions read and write the contents of a vdata.

Value of `funcstr`	Function Syntax	Description
`'fdefine'`	`status = hdfvs('fdefine',vdata_id,`	Defines a new vdata field. `data_type` is a string specifying the HDF number type. It can be any hese strings: `'uchar8'`, `'uchar'`, `'char8'`, `'char'`, `'double'`, `'uint8'`, `'uint16'`, `'uint32'`, `'float'`, `'int8'`, `'int16'`, or `'int32'`.
`'setclass'`	`status = hdfvs('setclass',vdata_id`	Assigns a class to a vdata.
`'setfields'`	`status = hdfvs('setfields',vdata_i`	Specifies the vdata fields to be written.
`'setinterlace'`	`status = hdfvs('setinterlace',vdat`	Sets the interlace mode for a vdata. `interlace` can be `'full'` or `'no'`.
`'setname'`	`status = hdfvs('setname',vdata_id,`	Assigns a name to a vdata.
`'write'`	`count = hdfvs('write', vdata_id, data)`	Writes to a vdata. data must be an `nfields`-by-1 cell array. Each cell must contain an `order(i)`-by-`n` vector of data where `order(i)` is the number of scalar values in each field. The types of the data must match the field types set via `hdfvs('setfields')` or the fields in an already existing vdata.

Value of `funcstr`	Function Syntax	Description
`'read'`	`[data,count] = hdfvs('read',vdata_id,n)`	Reads from a vdata. Data is returned in a `nfields`-by-1 cell array. Each cell contains a `order(i)`-by-`n` vector of data where `order` is the number of scalar values in each field. The fields are returned in the same order as specified in `hdfvs('setfields',...)`.
`'seek'`	`pos = hdfvs('seek',vdata_id,rec`	Seeks to a specified record in a vdata.
`'setattr'`	`status = hdfvs('setattr',vdata_id,`	Sets the attribute of a vdata field or vdata.
`'setexternalfile'`	`status = hdfvs('setexternalfile',v`	Stores vdata information in an external file.
`'getattr'`	`[value,status] = hdfvs('getattr',vdata_id,`	Reads the value of an attribute attached to a vdata or a vdata field. Set `field_index` to `'vdata'` to retrieve an attribute attached to the field itself. Set `field_index` to the numerical index of the field to retrieve an attribute attached to a vdata field.
`'setattr'`	`status = hdfvs('setattr',vdata_id,`	Sets the attribute of a vdata field or vdata.`field_index` can be an index number or `'vdata'`.

File Inquiry Functions

File inquiry functions provide information about how vdatas are stored in a file. They are useful for locating vdatas in a file.

Value of `funcstr`	Function Syntax	Description
`'find'`	`vdata_ref = hdfvs('find',file_id,vdat`	Searches for a given vdata name in the specified HDF file.

Value of `funcstr`	Function Syntax	Description
`'findclass'`	`vdata_ref = hdfvs('findclass',file_id`	Returns the reference number of the first vdata corresponding to the specified vdata class.
`'getid'`	`next_ref = hdfvs('getid',file_id,vda`	Returns the identifier of the next vdata in the file.
`'lone'`	`[refs,count] = hdfvs('lone',file_id,maxs`	Returns the reference numbers of the vdatas that are not linked into vgroups.

Vdata Inquiry Functions

Vdata inquiry functions provide specific information about a given vdata, including the vdata's name, class, number of fields, number of records, tag and reference pairs, interlace mode, and size.

Value of `funcstr`	Function Syntax	Description
`'fexist'`	`status = hdfvs('fexist',vdata_id,f`	Tests for the existence of fields in the specified vdata.
`'inquire'`	`[n,interlace,fields,nbyte = ... hdfvs('inquire',vdata_id)`	Returns information about the specified vdata.
`'elts'`	`count = hdfvs('elts',vdata_id)`	Returns the number of records in the specified vdata.
`'getclass'`	`[class_name,status] = hdfvs('getclass',vdata_id`	Returns the HDF class of the specified vdata.
`'getfields'`	`[field_names,count] = hdfvs('getfields',vdata_i`	Returns all field names within the specified vdata.
`'getinterlace'`	`[interlace,status] = hdfvs('getinterlace',vdat`	Retrieves the interlace mode of the specified vdata.
`'getname'`	`[vdata_name,status] = hdfvs('getname',vdata_id)`	Retrieves the name of the specified vdata.
`'getversion'`	`version = hdfvs('getversion',vdata_`	Returns the version number of a vdata.

Value of `funcstr`	Function Syntax	Description
`'sizeof'`	`nbytes = hdfvs('sizeof',vdata_id,f`	Returns the fields sizes of the specified vdata.
`'Queryfields'`	`[fields,status] = hdfvs('Queryfields',vdata`	Returns the field names of the specified vdata.
`'Queryname'`	`[name,status] = hdfvs('Queryname',vdata_i`	Returns the name of the specified vdata.
`'Queryref'`	`ref = hdfvs('Queryref',vdata_id`	Retrieves the reference number of the specified vdata.
`'Querytag'`	`tag = hdfvs('Querytag',vdata_id`	Retrieves the tag of the specified vdata.
`'Querycount'`	`[count,status] = hdfvs('Querycount',vdata_`	Returns the number of records in the specified vdata.
`'Queryinterlace'`	`[interlace,status] = hdfvs('Queryinterlace',vd`	Returns the interlace mode of the specified vdata.
`'Queryvsize'`	`vsize = hdfvs('Queryvsize',vdata_`	Retrieves the local size in bytes of the specified vdata record.
`'findex'`	`[field_index,status] = hdfvs('findex',vdata_id,f`	Queries the index of a vdata field given the field name.
`'nattrs'`	`count = hdfvs('nattrs',vdata_id)`	Returns the number of attributes of the specified vdata and the vdata fields contained in it.
`'fnattrs'`	`count = hdfvs('fnattrs',vdata_id,`	Queries the total number of vdata attributes.
`'findattr'`	`attr_index = hdfvs('findattr',vdata_id`	Retrieves the index of an attribute given the attribute name.
`'isattr'`	`tf = hdfvs('isattr',vdata_id)`	Determines if the given vdata is an attribute.

Value of `funcstr`	Function Syntax	Description
`'attrinfo'`	`[name,data_type,count,nby = hdfvs('attrinfo',... vdata_id,field_index,attr`	Returns the name, data type, number of values, and the size of the values of the specified attributes of the specified vdata field or vdata.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

height

Number of table rows

Syntax

```
H = height(T)
```

Description

`H = height(T)` returns the number of rows in the table, `T`.

`height(T)` is equivalent to `size(T,1)`.

Examples

Number of Table Rows

Create a table, `T`.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
T =

               Age    Height    Weight    BloodPressure
               ___    ______    ______    _____________

    Smith      38     71        176       124          93
    Johnson    43     69        163       109          77
    Williams   38     64        131       125          83
    Jones      40     67        133       117          75
    Brown      49     64        119       122          80
```

Find the number of rows in table `T`.

```
H = height(T)
```

```
H =

     5
```

`T` contains five rows; `height` does not count the variable names.

Input Arguments

`T` — Input table

table

Input table, specified as a table.

See Also

`numel` | `size` | `width`

help

Help for functions in Command Window

Syntax

```
help
help name
```

Description

`help` lists all primary help topics in the Command Window. Each main help topic corresponds to a folder name on the MATLAB search path.

`help name` displays the help text for the functionality specified by `name`, such as a function, method, class, or toolbox.

Input Arguments

`name`

String that specifies an operator symbol (such as +) or the name of a function, class, method, package, toolbox folder, or other functionality.

Some classes and other packaged items require that you specify the package name. Events, properties, and some methods require that you specify the class name. Separate the components of the name with periods, such as:

```
help className.name
help packageName.name
help packageName.className.name
```

If `name` is overloaded, that is, appears in multiple folders on the search path, `help` displays the help text for the first instance of `name` found on the search path, and displays a hyperlinked list of the overloaded functions and their folders.

When `name` specifies the name or partial path of a toolbox folder:

- If the folder contains a nonempty `Contents.m` file, the `help` function displays the file. `Contents.m` contains a list of MATLAB program files in the folder and their descriptions. If `Contents.m` exists, but is empty, MATLAB responds with `No help found for` *`name`*.
- If the folder does not contain a `Contents.m` file, the `help` function lists the first line of help text for each program file in the folder.
- If `name` is the name of both a function and a toolbox, `help` displays the associated text for both the toolbox and the function.

Examples

Functions and Overloaded Methods

Display help for the MATLAB `close` function.

```
help close
```

Because `close` refers to the name of a function and to the name of several methods, the help text includes hyperlinks to the overloaded methods.

Request help for the Database Toolbox™ `close` method.

```
help database.close
```

Package, Class, and Method Help

Display help for the `containers` package, `Map` class, and the `isKey` method.

```
help containers
help containers.Map
help containers.Map.isKey
```

Not all packages, classes, and associated methods or events require complete specification. For example, display the help for the `throwAsCaller` method of the `MException` class.

```
help throwAsCaller
```

Functions in Folder

List all of the functions in the folder *`matlabroot`*`/toolbox/matlab/general` by specifying a partial path.

```
help general
```

Alternatives

View more extensive help using the `doc` command or the Function Browser. To open the Function Browser, click its icon,  .

More About

Tips

- Some help text displays the names of functions in uppercase characters to make them stand out from the rest of the text. When typing these function names, use lowercase. For function names that appear in mixed case (such as `javaObject`), type the names as shown.
- To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`, and then enter the `help` statement.
- Some classes require that you specify the package name to display the help text. To identify the package name, create an instance of the class, and then call `class(obj)`.

- "Ways to Get Function Help"
- "Add Help for Your Program"

See Also

`class` | `dbtype` | `doc` | `lookfor` | `more` | `path` | `what` | `which` | `whos`

Introduced before R2006a

helpbrowser

Open Help browser to access online documentation

Note: `helpbrowser` will be removed in a future release. Use `doc` instead.

Syntax

```
helpbrowser
```

Description

`helpbrowser` displays the Help browser, open to its default startup page.

More About

- "Ways to Get Function Help"

See Also

`doc` | `help`

Introduced before R2006a

helpdesk

Open Help browser

Note: `helpdesk` will be removed in a future release. Use `doc` instead.

Syntax

```
helpdesk
```

Description

`helpdesk` opens the Help browser to the default startup page. In previous releases, `helpdesk` displayed the Help Desk, which was the precursor to the Help browser.

See Also

`doc`

Introduced before R2006a

helpdlg

Create help dialog box

Syntax

```
helpdlg
helpdlg('helpstring')
helpdlg('helpstring','dlgname')
h = helpdlg(...)
```

Description

`helpdlg` creates a nonmodal help dialog box or brings the named help dialog box to the front.

Note A nonmodal dialog box enables the user to interact with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`helpdlg` displays a dialog box named `'Help Dialog'` containing the string `'This is the default help string.'`

`helpdlg('helpstring')` displays a dialog box named `'Help Dialog'` containing the string specified by `'helpstring'`.

`helpdlg('helpstring','dlgname')` displays a dialog box named `'dlgname'` containing the string `'helpstring'`.

`h = helpdlg(...)` returns the handle of the dialog box.

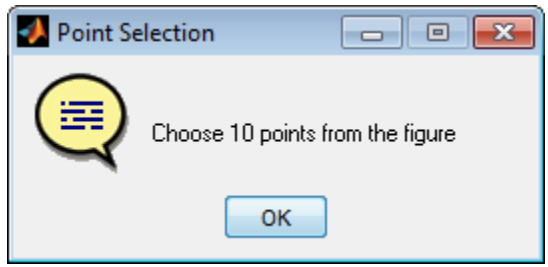
Examples

The statement

```
helpdlg('Choose 10 points from the figure',...
```

```
        'Point Selection');
```

displays this dialog box:



More About

Tips

MATLAB wraps the text in `'helpstring'` to fit the width of the dialog box. The dialog box remains on your screen until you press the **OK** button or the **Enter** key. After either of these actions, the help dialog box disappears.

See Also

`dialog` | `errordlg` | `inputdlg` | `listdlg` | `msgbox` | `questdlg` | `warndlg` | `figure` | `uiwait` | `uiresume`

Introduced before R2006a

helpwin

Provide access to help comments for all functions

Note: `helpwin` will be removed in a future release. Use `doc` instead.

Syntax

```
helpwin
helpwin topic
```

Description

`helpwin` lists topics for groups of functions in the MATLAB Help browser. It shows brief descriptions of the topics and provides links to display help comments for the functions. You cannot follow links in the `helpwin` list of functions if the MATLAB software is busy (for example, running a program).

`helpwin topic` displays help information for the topic. If `topic` is a folder, it displays all functions in the folder. If `topic` is a function, `helpwin` displays help for that function. From the page, you can access a list of folders (**Default Topics** link) as well as the reference page help for the function (**Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

Examples

Typing

```
helpwin datafun
```

displays the functions in the `datafun` folder and a brief description of each.

Typing

```
helpwin fft
```

displays the help for the `fft` function.

See Also

`doc` | `help`

Introduced before R2006a

hess

Hessenberg form of matrix

Syntax

```
H = hess(A)
[P,H] = hess(A)
[AA,BB,Q,Z] = hess(A,B)
```

Description

`H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.

`[P,H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P*H*P'` and `P'*P = eye(size(A))` .

`[AA,BB,Q,Z] = hess(A,B)` for square matrices `A` and `B`, produces an upper Hessenberg matrix `AA`, an upper triangular matrix `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA` and `Q*B*Z = BB`.

Definitions

A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H =
   -149    -50   -154
    537    180    546
    -27     -9    -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =
   -149.0000    42.2037   -156.3165
   -537.6783   152.5511   -554.9272
           0     0.0728      2.4489
```

See Also

`eig` | `qz` | `schur`

Introduced before R2006a

matlab.mixin.Heterogeneous class

Package: matlab.mixin

Superclass for heterogeneous array formation

Description

`matlab.mixin.Heterogeneous` is an abstract class that provides support for the formation of heterogeneous arrays. A heterogeneous array is an array of objects that differ in their specific class, but are all derived from or are instances of a root class. The root class derives directly from `matlab.mixin.Heterogeneous`.

Heterogeneous Hierarchy

Use `matlab.mixin.Heterogeneous` to define hierarchies of classes whose instances you can combine into heterogeneous arrays.

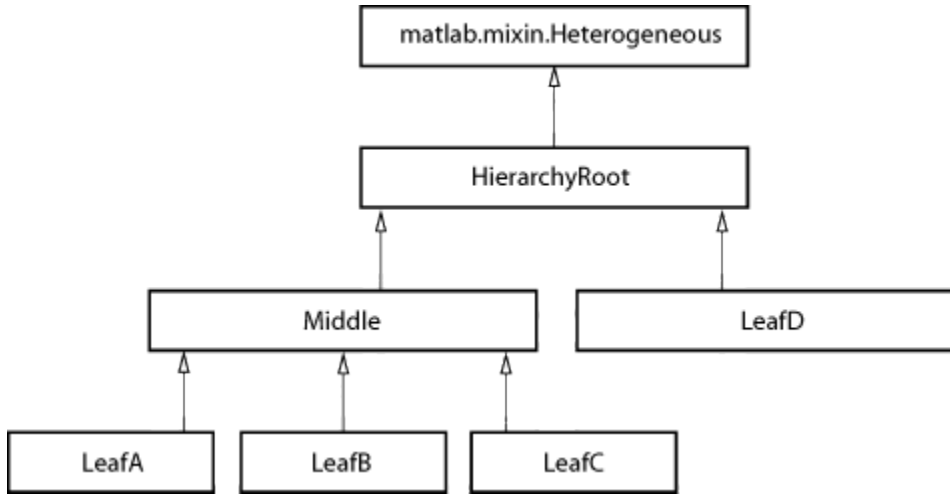
The following class definition enables the formation of heterogeneous arrays that combine instances of any classes derived from `HierarchyRoot`.

```
classdef HierarchyRoot < matlab.mixin.Heterogeneous
% HierarchyRoot is a direct subclass of
% matlab.mixin.Heterogeneous
% HierarchyRoot is the root of this heterogeneous hierarchy
```

Deriving the `HierarchyRoot` class directly from `matlab.mixin.Heterogeneous` enables the `HierarchyRoot` class to become the root of a hierarchy of classes. You can combine instances of the members of this hierarchy into a heterogeneous array. Only instances of classes derived from the same root class can combine to form a valid heterogeneous array.

Class of a Heterogeneous Array

The class of a heterogeneous array is always the class of the most specific superclass common to all objects in the array. For example, suppose you define the following class hierarchy:



Forming an array (`harray`) of an instance of `LeafA` with an instance of `LeafB` creates an array of class `Middle`:

```
harray = [LeafA,LeafB];
class(harray)
ans =
Middle
```

Forming an array (`harray`) of an instance of `LeafC` with an instance of `LeafD` creates an array of class `HierarchyRoot`:

```
harray = [LeafC,LeafD];
class(harray)
ans =
HierarchyRoot
```

Forming an array (`harray`) of an instance of `LeafA` with an other instance of `LeafA` creates a homogeneous array of class `LeafA`:

```
harray = [LeafA,LeafA];
class(harray)
ans =
LeafA
```

Note: You cannot form heterogeneous arrays that include instances of classes that are not derived from the same hierarchy root (that is, the `HierarchyRoot` class in the hierarchy shown previously).

Forming a Heterogeneous Array

Heterogeneous arrays are the result of operations that produces arrays containing instances of two or more classes from the heterogeneous hierarchy. Usually, concatenation or indexed assignment form these arrays. For example, these statements form `harray` using indexed assignment:

```
harray(1) = LeafA;
harray(2) = LeafC;
class(harray)
ans =
Middle
```

Growing the Array Can Change Its Class

Assigning new objects into an array containing objects derived from `matlab.mixin.Heterogeneous` can change the class of the array. For example, given a homogeneous array containing objects only of the `LeafA` class:

```
harray = [LeafA,LeafA,LeafA];
class(harray)
ans =
LeafA
```

Adding an object of another class derived from the same root to `harray` converts the array's class to the most specific superclass:

```
harray(4) = LeafB;
class(harray)
ans =
Middle
```

Method Dispatching

When MATLAB invokes a method for which the dominant argument is a heterogeneous array, the method:

- Must be defined for the class of the heterogeneous array, either directly by the class of the array or inherited from a superclass.

- Must be `Sealed = true` (cannot be overridden by a subclass).

The class of the heterogeneous array determines which class method executes for any given method invocation, as is the case with a homogeneous array. MATLAB does not consider the class of individual elements in the array when dispatching to methods.

Sealing Inherited Methods

The requirement that methods called on a heterogeneous array be `Sealed = true` ensures correct and predictable behavior with all array elements.

You must override methods that are inherited from outside the heterogeneous hierarchy if these methods are not `Sealed = true` and you want to call these methods on heterogeneous arrays.

For example, suppose you define a heterogeneous array by subclassing `matlab.mixin.SetGet`, in addition to `matlab.mixin.Heterogeneous`. Override the `set` method to call the `matlab.mixin.SetGet` superclass method as required by your class design:

```
methods(Sealed)
   function varargout = set(obj,varargin)
      if nargout == 0
         set@matlab.mixin.SetGet(obj, varargin{:});
      else
         varargout{:} = set@matlab.mixin.SetGet(obj,varargin{:});
      end
   end
end
```

Method implementations can take advantage of the fact that, given a heterogeneous array `harray`, and a scalar index `n`, the expression

```
harray(n)
```

is not a heterogeneous array. Therefore, when invoking a method on a single element of a heterogeneous array, special requirements for heterogeneous arrays do not apply.

Defining the Default Object

When working with object arrays (both heterogeneous and homogeneous), MATLAB creates default objects to fill in missing array elements by calling the class constructor

with no arguments. Filling in missing array elements becomes necessary in cases such as:

- Indexed assignment creates an array with gaps. For example, if `harray` is not previously defined:

  ```
  harray(5) = LeafA;
  ```

- Loading a heterogeneous array from a MAT-file when MATLAB cannot find the class definition of a specific object.

The `matlab.mixin.Heterogeneous` class provides a default implementation of a method called `getDefaultScalarElement`. This method returns an instance of the root class of the heterogeneous hierarchy, unless the root class is abstract.

If the root class is abstract or is not an appropriate default object for the classes in the heterogeneous hierarchy, you can override the `getDefaultScalarElement` method to return an instance of a class derived from the root class.

Defining the getDefaultScalarElement Method

Specify the class of the default object by overriding the `matlab.mixin.Heterogeneous` method called `getDefaultScalarElement` in the root class of the heterogeneous hierarchy. You can override `getDefaultScalarElement` only in the root class (direct subclasses of `matlab.mixin.Heterogeneous`).

`getDefaultScalarElement` has the following signature:

```
methods (Static, Sealed, Access = protected)
   function default_object = getDefaultScalarElement
   ...
   end
end
```

The `getDefaultScalarElement` method must satisfy these criteria:

- Static — MATLAB calls this method without an object.
- Protected — MATLAB calls this method, object users do not.
- Sealed (not required) — Seal this method to ensure users of the heterogeneous hierarchy do not change the intended behavior of the class.
- It must return a scalar object
- It must pass the `isa` test for the root class, that is:

```
(isa(getDefaultScalarElement,'HierarchyRoot')
```

where *`HierarchyRoot`* is the name of the heterogeneous hierarchy root class. This means the default object can be an instance of any class derived from the root class.

Cannot Redefine Indexing or Concatenation

The use of heterogeneous arrays requires consistent indexing and concatenation behaviors. Therefore, subclasses of `matlab.mixin.Heterogeneous` cannot change their default indexed reference, indexed assignment, or concatenation behavior.

You cannot override the following methods in your subclasses:

- `cat`
- `horzcat`
- `vertcat`
- `subref`
- `subsasign`

In cases involving multiple inheritance in which your subclass inherits from superclasses in addition to `matlab.mixin.Heterogeneous`, the superclasses cannot define any of these methods.

Default Concatenation Behavior

Statements of the form:

```
a = [obj1,obj2,...];
```

create an array, `a`, containing the objects listed in brackets.

Concatenating `Heterogeneous` objects of the same specific class retains the class of the objects and does not form a heterogeneous array.

Concatenating `Heterogeneous` objects derived from the same root superclass, but that are of different specific classes, yields a heterogeneous array. MATLAB does not attempt to convert the class of any array members if all are part of the same root hierarchy.

Indexed Assignment Behavior

Statements of the form:

```
a(m:n) = [objm,...objn];
```

assign the right-hand side objects to the array elements (`m:n`), specified on the left side of the assignment.

Indexed assignment to a heterogeneous array can:

- Increase or decrease the size of the array
- Overwrite existing array elements
- Change property values of objects within the array
- Change the class of the array
- Change whether the array is heterogeneous

Indexed Reference Behavior

Statements of the form:

```
a = harray(m:n);
```

assign the elements of `harray` referenced by indices `m:n`, to array `a`.

Indexed reference on a heterogeneous array returns a sub-range of the original array. Depending on the specific elements within that sub-range (`m:n`), the result might have a different class than the original array, and might not be heterogeneous.

Converting Nonmember Objects

If you attempt to form a heterogeneous array with objects that are not derived from the same root class, MATLAB calls the `convertObject` method, if it exists, to convert objects to the dominant class. Implementing a `convertObject` method enables the formation of heterogeneous arrays containing objects that are not part of the heterogeneous hierarchy.

When Is Conversion Necessary

Suppose there are two classes `A` and `B`, where `B` is not derived from `matlab.mixin.Heterogeneous`, or where `A` and `B` are derived from different root classes that are derived from `matlab.mixin.Heterogeneous`.

MATLAB attempts to call the `convertObject` method implemented by the root class of `A` in the following cases:

- The indexed assignment:

  ```
  A(k) = B
  ```

- Horizontal and vertical concatenations:

  ```
  [A,B] and [A;B]
  ```

Implement a `convertObject` method if you want to support conversion of objects whose class is not defined in your heterogeneous hierarchy. You do not need to implement this method if your class design does not require this conversion.

Implementing convertObject

Only the root class of the heterogeneous hierarchy can implement a `convertObject` method.

The `convertObject` method must have the following signature:

```
Method (Static, Sealed, Access = protected)
   function cobj = convertObject('DomClass',objToConvert)
   ...
   end
end
```

For indexed assignment `A(k) = B` and concatenation `[A,B]`:

- *`DomClass`* — Name of the class of the array `A`
- `objToConvert` — Object to be converted, `B` in this case
- `cobj` — Legal member of the heterogeneous hierarchy to which `A` belongs

You must implement `convertObject` to return a valid object of class `A` or MATLAB issues an error.

Handle Compatibility

The `matlab.mixin.Heterogeneous` class is handle compatible. It can be combined with either handle or value classes when defining a subclass using multiple superclasses. See “Supporting Both Handle and Value Subclasses” for information on handle compatibility.

The `matlab.mixin.Heterogeneous` class is a value class. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Methods

cat	Concatenation for heterogeneous arrays
getDefaultScalarElement	Return default object for heterogeneous array operations
horzcat	Horizontal concatenation for heterogeneous arrays
vertcat	Vertical concatenation for heterogeneous arrays

See Also

`handle`

How To

- Class Attributes
- Property Attributes
- “Creating Subclasses — Syntax and Techniques”

cat

Class: matlab.mixin.Heterogeneous
Package: matlab.mixin

Concatenation for heterogeneous arrays

Syntax

```
C = cat(dim,A,B)
```

Description

`C = cat(dim,A,B)` concatenates objects `A` and `B` along the dimension `dim`.
The class of object arrays `A` and `B` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

- If `A` and `B` are of the same class, the class of the resulting array is unchanged.
- If `A` and `B` are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array and the array's class is that of the most specific superclass shared by `A` and `B`.

The `cat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

`dim`

Scalar dimension along which to concatenate arrays

Default:

`A`

Object array derived from the same root subclass of `matlab.mixin.Heterogeneous` as `B`

B

Object array derived from the same root subclass of `matlab.mixin.Heterogeneous` as A

Output Arguments

C

Array resulting from the specified concatenation. The class of this array is that of the most specific superclass shared by A and B.

Attributes

Sealed	true

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.Heterogeneous` | `cat`

matlab.mixin.Heterogeneous.getDefaultScalarElement

Class: matlab.mixin.Heterogeneous
Package: matlab.mixin

Return default object for heterogeneous array operations

Syntax

```
defaultObject = getDefaultScalarElement
```

Description

`defaultObject = getDefaultScalarElement` returns the default object for a heterogeneous hierarchy. Override this method if the "Root Class" on page 1-3587 is abstract or is not an appropriate default object for the classes in the heterogeneous hierarchy. `getDefaultScalarElement` must return an instance of another member of the heterogeneous hierarchy.

The `matlab.mixin.Heterogeneous` class provided a default implementation of this method that returns an instance of the "Root Class" on page 1-3587.

MATLAB calls the `getDefaultScalarElement` method when requiring a default object. See `matlab.mixin.Heterogeneous` for more information on heterogeneous arrays and default objects.

Tips

- Override `getDefaultScalarElement` only if the "Root Class" on page 1-3587 is not suitable as a default object.
- Override `getDefaultScalarElement` only in the "Root Class" on page 1-3587 of the heterogeneous hierarchy.
- `getDefaultScalarElement` must return a scalar object.
- `getDefaultScalarElement` must be a static method with protected access. While not required by MATLAB, you can seal this method to prevent overriding by other classes.

- MATLAB returns an error if the value returned by `getDefaultScalarElement` is not scalar or is not an instance of a class that is a valid member of the hierarchy.

Output Arguments

`defaultObject`

The default object for heterogeneous array operations.

Definitions

Root Class

Root class – The direct subclass of `matlab.mixin.Heterogeneous` that forms the root of a heterogeneous hierarchy. Classes of objects that you can combine into heterogeneous arrays must derive from this root class.

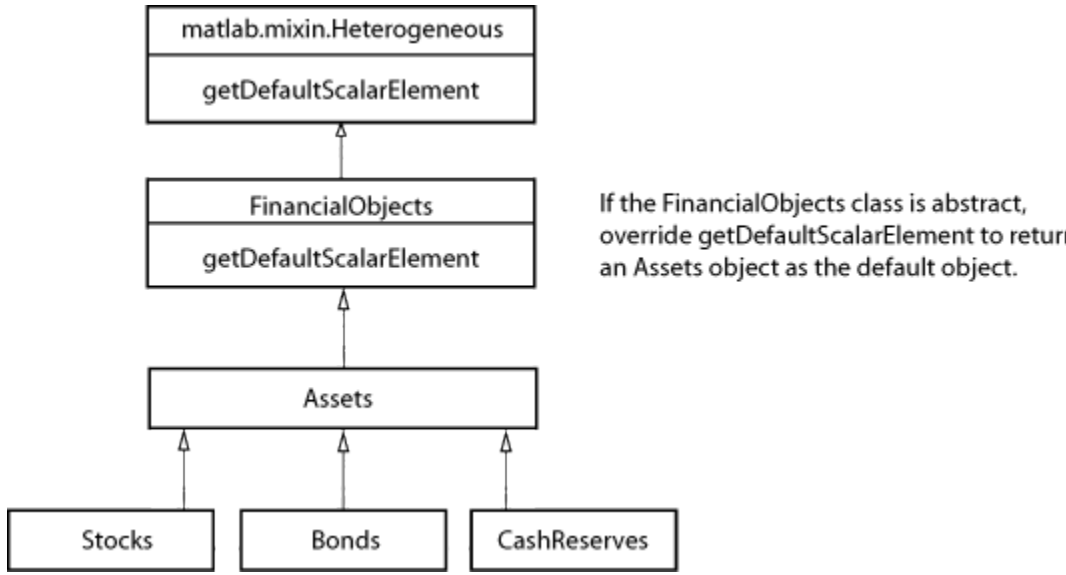
Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true` not required

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

This example describes a heterogeneous hierarchy with a root class (`FinancialObjects`) that is an abstract class and cannot, therefore, be used for the default object. The `FinancialObjects` class definition includes an override of the `getDefaultScalarElement` method which returns an instance of the `Assets` class as the default object.



The root class can override the `getDefaultScalarElement` method that is defined in `matlab.mixin.Heterogeneous` class and return an `Assets` object as the default object.

```
classdef FinancialObjects < matlab.mixin.Heterogeneous
   methods (Abstract)
      val = determineCurrentValue(obj)
   end
   methods (Static, Sealed, Access = protected)
      function default_object = getDefaultScalarElement
         default_object = Assets;
      end
   end
end
```

See Also

`matlab.mixin.Heterogeneous`

horzcat

Class: matlab.mixin.Heterogeneous
Package: matlab.mixin

Horizontal concatenation for heterogeneous arrays

Syntax

```
C = horzcat(A1,A2,...)
```

Description

`C = horzcat(A1,A2,...)` concatenates the `matlab.mixin.Heterogeneous` objects `A1`, `A2`, and so on, to form the array `C`. All input arrays must have the same number of rows.

The class of object arrays `A1,A2,...` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

MATLAB calls:

```
C = horzcat(A1,A2,...)
```

for the expressions:

```
C = [A1,A2,...]
C = [A1 A2 ...]
```

when `A1` is an array of `matlab.mixin.Heterogeneous` objects.

If all input arguments are of the same specific class, the class of the resulting array is unchanged. If all input arguments are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array. The array's class is that of the most specific superclass shared by all input arguments.

If all input arguments are not members of the same heterogeneous hierarchy, MATLAB calls the `convertObjects` method, if defined by the dominant root class (the first argument or the left-most element in the concatenation if no other class is dominant).

The `horzcat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

A1

Object array of class `matlab.mixin.Heterogeneous`

A2

Object array of class `matlab.mixin.Heterogeneous`

Output Arguments

C

Array resulting from the specified concatenation. The class of this array is that of the most specific superclass shared by the input arguments.

Attributes

`Sealed`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.Heterogeneous` | `horzcat`

vertcat

Class: matlab.mixin.Heterogeneous
Package: matlab.mixin

Vertical concatenation for heterogeneous arrays

Syntax

```
C = vertcat(A1,A2,...)
```

Description

`C = vertcat(A1,A2,...)` concatenates the `matlab.mixin.Heterogeneous` objects `A1`, `A2`, and so on, to form the array `C`. All input arrays must have the same number of columns.

The class of object arrays `A1,A2,...` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

MATLAB calls:

```
C = vertcat(A1,A2,...)
```

for the expression:

```
C = [A1;A2;...]
```

when `A1` and `A2`, and so on are arrays of `matlab.mixin.Heterogeneous` objects.

If all input arguments are of the same specific class, the class of the resulting array is unchanged. If all input arguments are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array. The array's class is that of the most specific superclass shared by all input arguments.

If all input arguments are not members of the same heterogeneous hierarchy, MATLAB calls the `convertObjects` method, if defined by the dominant root class (the first argument or the left-most element in the concatenation if no other class is dominant).

The `horzcat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

`A1`

Object array of class `matlab.mixin.Heterogeneous`

`A2`

Object array of class `matlab.mixin.Heterogeneous`

Output Arguments

`C`

Array resulting from the specified vertical concatenation. The class of this array is that of the most specific superclass shared by the input arguments.

Attributes

`Sealed`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.Heterogeneous` | `vertcat`

hex2dec

Convert hexadecimal number string to decimal number

Syntax

```
d = hex2dec('hex_value')
```

Description

`d = hex2dec('hex_value')` converts *`hex_value`* to its floating-point integer representation. The argument *`hex_value`* is a hexadecimal integer stored in a MATLAB string or cell array of strings. If the value of *`hex_value`* is greater than hexadecimal `10,000,000,000,000`, `hex2dec` might not return an exact conversion.

If *`hex_value`* is a cell array of strings, each row is interpreted as a hexadecimal string.

Examples

```
hex2dec('3ff')

ans =

        1023
```

For a character array `S`,

```
S =
0FF
2DE
123
hex2dec(S)

ans =

255
734
291
```

See Also

dec2hex | format | hex2num | sprintf

hex2num

Convert hexadecimal number string to double-precision number

Syntax

```
n = hex2num(S)
```

Description

`n = hex2num(S)`, where `S` is a 16 character string or cell array of strings representing a hexadecimal number, returns the IEEE double-precision floating-point number `n` that it represents. Fewer than 16 characters are padded on the right with zeros. If `S` is a cell array of strings, each row is interpreted as a double-precision number.

`NaN`s, infinities and denorms are handled correctly.

Examples

```
hex2num('400921fb54442d18')
```

returns `Pi`.

```
hex2num('bff')
```

returns

```
ans =

    -1
```

See Also

`num2hex` | `hex2dec` | `sprintf` | `format`

Introduced before R2006a

hgexport

Export figure

Syntax

```
hgexport(h,filename)
hgexport(h,'-clipboard')
```

Description

`hgexport(h,filename)` writes figure `h` to the file `filename`.

`hgexport(h,'-clipboard')` writes figure `h` to the Microsoft Windows clipboard.

The format in which the figure is exported is determined by which renderer you use. The Painters renderer generates a metafile. The OpenGL renderer generate a bitmap.

Alternatives

Use the **File > Export Setup** dialog. Use **Edit > Copy Figure** to copy the figure's content to the system clipboard. For details, see "Customize Figure Interactively Before Saving" and "Copy Figure to Clipboard From Edit Menu".

See Also

`print`

Introduced before R2006a

hggroup

Create group object

Syntax

```
h = hggroup
h = hggroup(...,'PropertyName',propertyvalue,...)
```

Properties

For a list of properties, see Group Properties.

Description

`h = hggroup` creates a group object as a child of the current axes and returns its handle, `h`.

`h = hggroup(...,'PropertyName',propertyvalue,...)` creates a group object with the property values specified in the argument list.

A group object can be the parent of any axes, as well as other group objects. Use group objects to form a group of child objects that can be treated as a single object.

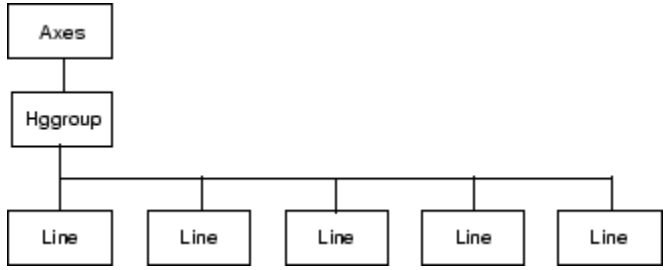
Examples

Plot random data and parented the lines to the group object.

```
hg = hggroup;
plot(randn(5),randn(5),'Parent',hg)
```

Instance Diagram for This Example

The following diagram shows the object hierarchy created by this example.



More About

- “Create Object Groups”

See Also

`hgtransform`

Introduced before R2006a

Group Properties

Control group appearance and behavior

Group object properties control the behavior of group objects. By changing property values, you can modify certain aspects of the group object.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = hggroup;
c = h.Children;
h.Visible = 'off';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Visibility

`Visible` — Visibility of group

`'on'` (default) | `'off'`

Visibility of group, specified as one of these values:

- `'on'` — Display all objects in the group.
- `'off'` — Hide all objects in the group. You still can access the properties of invisible group objects. Setting the `Visible` property for the group object does not change the `Visible` property for objects in the group.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces

the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.

- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'hggroup'`

Type of graphics object, returned as `'hggroup'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Tag to associate with group

`''` (default) | string

Tag to associate with the group, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with group

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the group object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the group.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the group object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the group from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the group object in the legend as one entry (default).
   - `'off'` — Do not include the group object in the legend.
   - `'children'` — Include only children of the group object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of group

axes object | group object | transform object

Parent of group, specified as an axes, group, or transform object.

`Children` — Children of group

empty `GraphicsPlaceholder` array | array of graphics objects

Children of group, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property of the group. To add a child to this list, set the `Parent` property of the child graphics object to the group object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of group object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The group object handle is always visible.
- `'off'` — The group object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the

HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

- `'callback'` — The group object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the group at the command-line, but allows callback functions to access it.

If the group object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the group. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The group object — You can access properties of the group object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the group. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the group when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the group.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Children that can capture mouse clicks

`'visible'` (default) | `'none'`

Children that can capture mouse clicks, specified as one of these values:

- `'visible'` — Any child object can capture a mouse click, depending on the `PickableParts` property value of the child.
- `'none'` — No child objects can capture mouse clicks, regardless of the `PickableParts` property value of the child.

`HitTest` — Response to mouse clicks captured by children

`'on'` (default) | `'off'`

Response to mouse clicks captured by children, specified one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of group. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Do not trigger the callbacks of the group. Instead, trigger the callbacks for the nearest ancestor of the group that has a `HitTest` property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

A group cannot capture mouse clicks. However, if you click a child of the group and if the child has a `HitTest` property set to `'off'`, then the child passes the click to the group.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running

callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the group is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the group tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the group. Setting the `CreateFcn` property on an existing group has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during group creation. MATLAB executes the callback after creating the group and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The group object — You can access properties of the group object from within the callback function. You also can access the group object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the group. MATLAB executes the callback before destroying the group so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The group object — You can access properties of the group object from within the callback function. You also can access the group object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of group

`'off'` (default) | `'on'`

Deletion status of group, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the group begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the group no longer exists.

Check the value of the BeingDeleted property to verify that the group is not about to be deleted before querying or modifying it.

See Also

`hggroup`

More About

- “Access Property Values”
- “Graphics Object Properties”

hgload

Load graphics object hierarchy from file

Syntax

```
h = hgload('filename')
[h,old_prop_values] = hgload(...,property_structure)
```

Description

Note: `hgload` is not recommended. Use `openfig` instead.

`h = hgload('filename')` loads graphics object hierarchy from the FIG-file specified by `filename` and returns handles to the top-level objects. If `filename` contains no extension, then MATLAB adds the `.fig` extension.

`[h,old_prop_values] = hgload(...,property_structure)` overrides the properties on the top-level objects stored in the FIG-file with the values in `property_structure`, and returns their previous values in `old_prop_values`.

`property_structure` must be a structure having field names that correspond to property names and values that are the new property values.

`old_prop_values` is a cell array equal in length to `h`, containing the old values of the overridden properties for each object. Each cell contains a structure having field names that are property names, each of which contains the original value of each property that has been changed. Any property specified in `property_structure` that is not a property of a top-level object in the FIG-file is not included in `old_prop_values`.

Nonserializable objects (such as the default toolbars and the default menus) are not saved because they are created when the figure is created. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files.

Alternatives

Use the **File > Open** on the figure window menu to access figure files with the **Open** dialog.

See Also

`hgsave | open`

Introduced before R2006a

hgsave

Save graphics object hierarchy to file

Syntax

```
hgsave('filename')
hgsave(h,'filename')
hgsave(...,'-v6')
hgsave(...,'-v7.3')
```

Description

Note: `hgsave` is not recommended. Use `savefig` instead.

`hgsave('filename')` saves the current figure to a file named `filename`.

`hgsave(h,'filename')` saves the objects identified by the array of handles `h` to a file named `filename`. If you do not specify an extension for `filename`, then the extension `.fig` is appended. If `h` is a vector, none of the handles in `h` may be ancestors or descendents of any other handles in `h`.

`hgsave(...,'-v6')` saves the FIG-file in a format that can be loaded by versions prior to MATLAB 7.

`hgsave(...,'-v7.3')` saves the FIG-file in a format that can be loaded only by MATLAB versions 7.3 and above. This format, based on HDF5 files, is intended for saving FIG-files larger than 2 GB.

Backward Compatibility

When creating a figure you want to save and use in a MATLAB version prior to MATLAB 7, use the `'v6'` option with the plotting function and the `'-v6'` option for `hgsave`. Check the reference page for the plotting function you are using for more information.

In MATLAB release R2014b or later, you cannot open a save FIG-file in earlier versions of MATLAB. Use `savefig` to save figures that are compatible with earlier versions of MATLAB.

Alternatives

Use the **File > Export Setup** dialog. Use **Edit > Copy Figure** to copy the figure's content to the system clipboard. For details, see "Customize Figure Interactively Before Saving" and "Copy Figure to Clipboard From Edit Menu".

See Also

`hgload` | `open` | `save` | `savefig`

Introduced before R2006a

hgsetget

Abstract class used to derive handle class with set and get methods

Note: `hgsetget` will be removed in a future release. Use `matlab.mixin.SetGet` instead.

Syntax

```
classdef myclass < hgsetget
```

Description

`classdef myclass < hgsetget` makes *`myclass`* a subclass of the `hgsetget` class, which is a subclass of the `handle` class.

Use the `hgsetget` class to derive classes that inherit `set` and `get` methods that behave like Handle Graphics `set` and `get` functions.

Methods

When you derive a class from the `hgsetget` class, your class inherits the following methods.

Method	Purpose
set	Assigns values to the specified properties or returns a cell array of possible values for writable properties.
get	Returns value of specified property or a `struct` with all property values.
setdisp	Called when `set` is called with no output arguments and a handle array, but no property name. Override this method to change what set displays.

Method	Purpose
getdisp	Called when `get` is called with no output arguments and handle array, but no property name. Override this method to change what get displays.

More About

- "Implementing a Set/Get Interface for Properties"

hgtransform

Create transform graphics object

Syntax

```
h = hgtransform
h = hgtransform('PropertyName',propertyvalue,...)
```

Properties

For a list of properties, see Transform Properties.

Description

`h = hgtransform` creates a transform object and returns its handle.

`h = hgtransform('PropertyName',propertyvalue,...)` creates a transform object with the property value settings specified in the argument list. For a description of the properties, see Transform Properties.

Transform objects can contain other objects, which lets you treat the transform object and its children as a single entity with respect to visibility, size, orientation, etc. You can group objects by parenting them to a single transform object (i.e., setting the object's `Parent` property to the transform object's handle):

```
h = hgtransform;
surface('Parent',h,...)
```

The primary advantage of parenting objects to an transform object is that you can perform *transforms* (for example, translation, scaling, rotation, etc.) on the child objects in unison.

The parent of a transform object is either an axes object or another transform object.

Although you cannot see a transform object, setting its `Visible` property to `off` makes all its children invisible as well.

Exceptions and Limitations

- A transform object can be the parent of any number of axes child objects belonging to the same axes, except for light objects.
- Transform objects can never be the parent of axes objects and therefore can contain objects only from a single axes.
- Transform objects can be the parent of other transform objects within the same axes.
- You cannot transform image objects because images are not true 3-D objects. Texture mapping the image data to a surface `CData` enables you to produce the effect of transforming an image in 3-D space.
- Transforms do not affect text objects unless the text object uses data units. If a text object has a position specified in data units, then the transform moves the lower left corner of the text. The transform does not affect the font size or orientation. To change the font size and orientation, use text properties.

Note Many plotting functions clear the axes (remove axes children) before drawing the graph. Clearing the axes also deletes any transform objects in the axes.

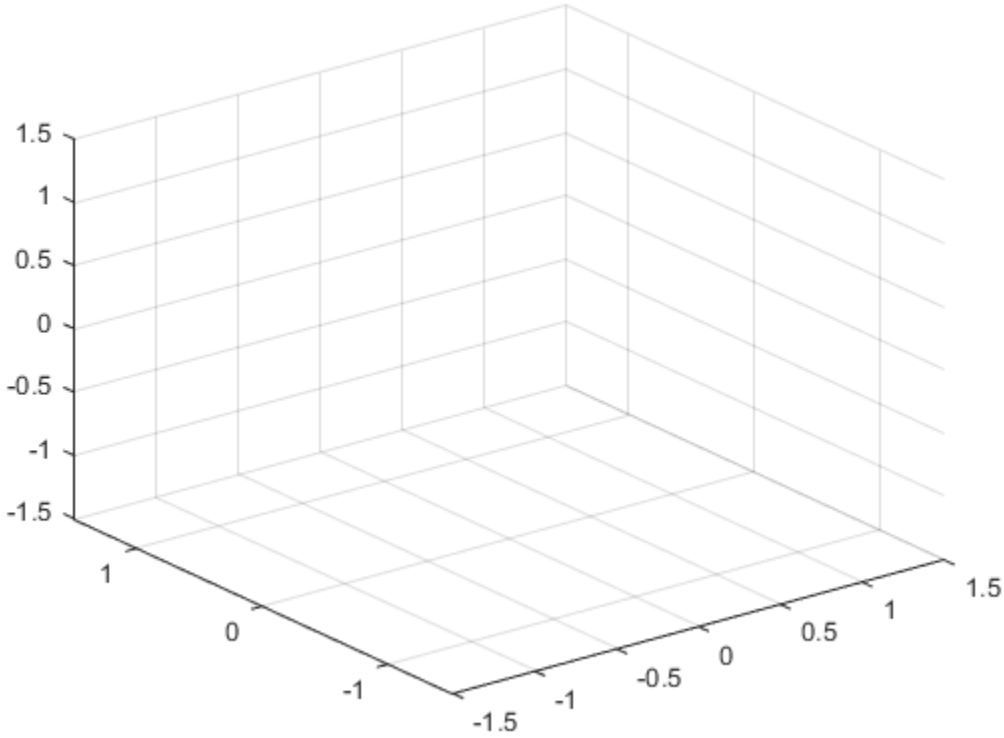
Examples

Transforming a Group of Objects

This example shows how to create a 3-D star with a group of surface objects parented to a single transform object. The transform object then rotates the object about the z-axis while scaling its size.

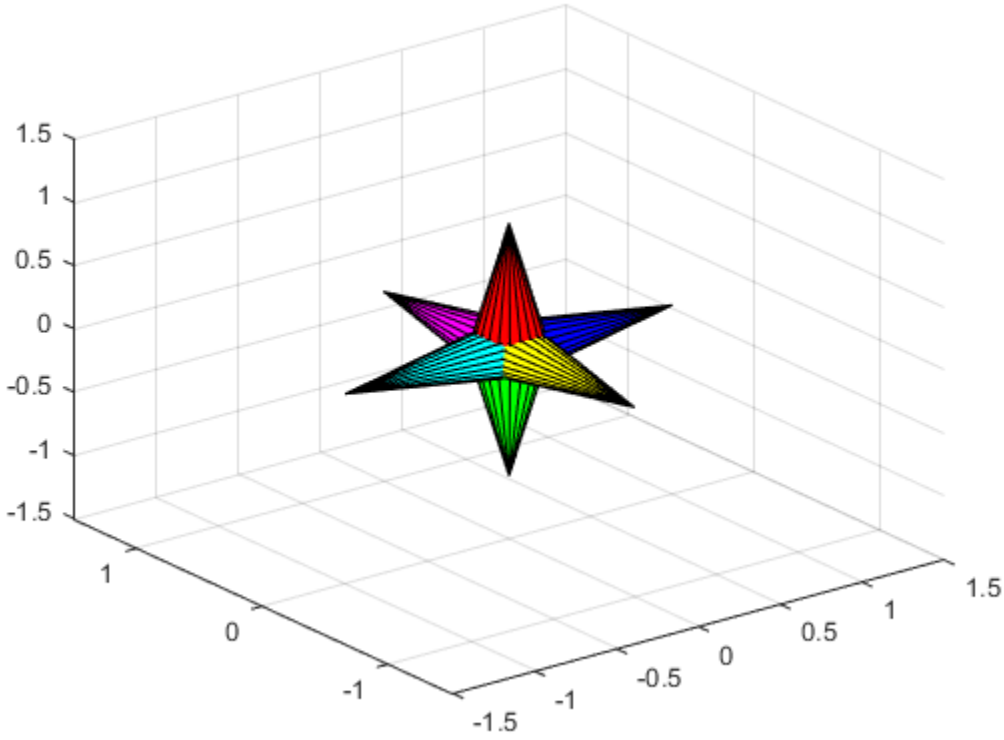
Create an axes and adjust the view. Set the axes limits to prevent auto limit selection during scaling.

```
ax = axes('XLim',[-1.5 1.5],'YLim',[-1.5 1.5],'ZLim',[-1.5 1.5]);
view(3)
grid on
```



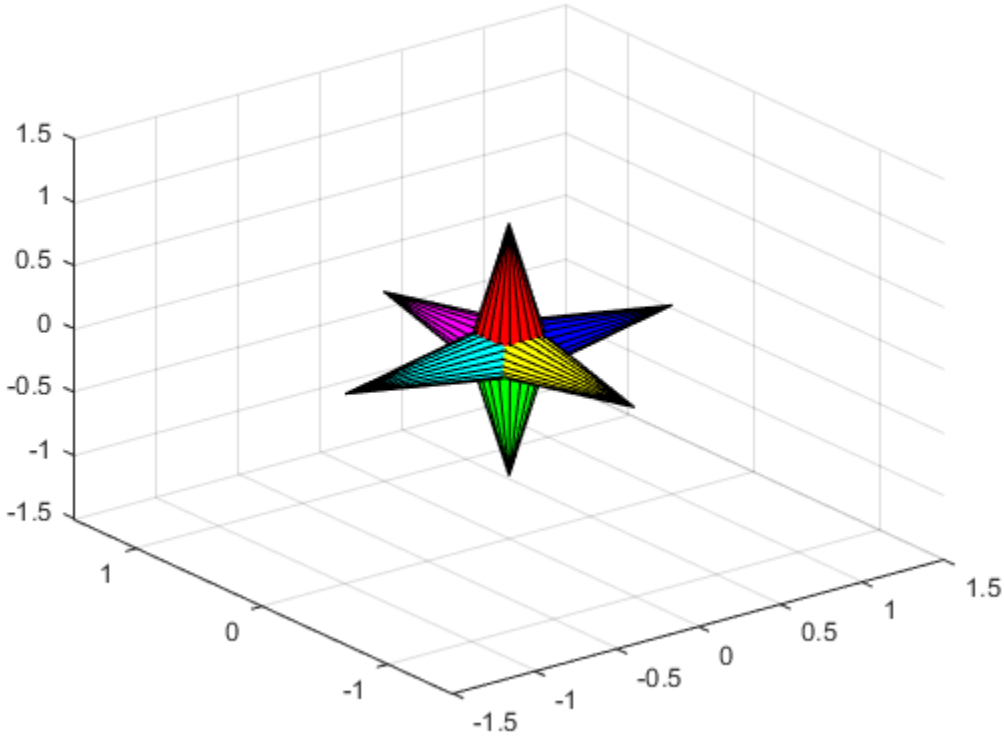
Create the objects you want to parent to the transform object.

```
[x,y,z] = cylinder([.2 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```



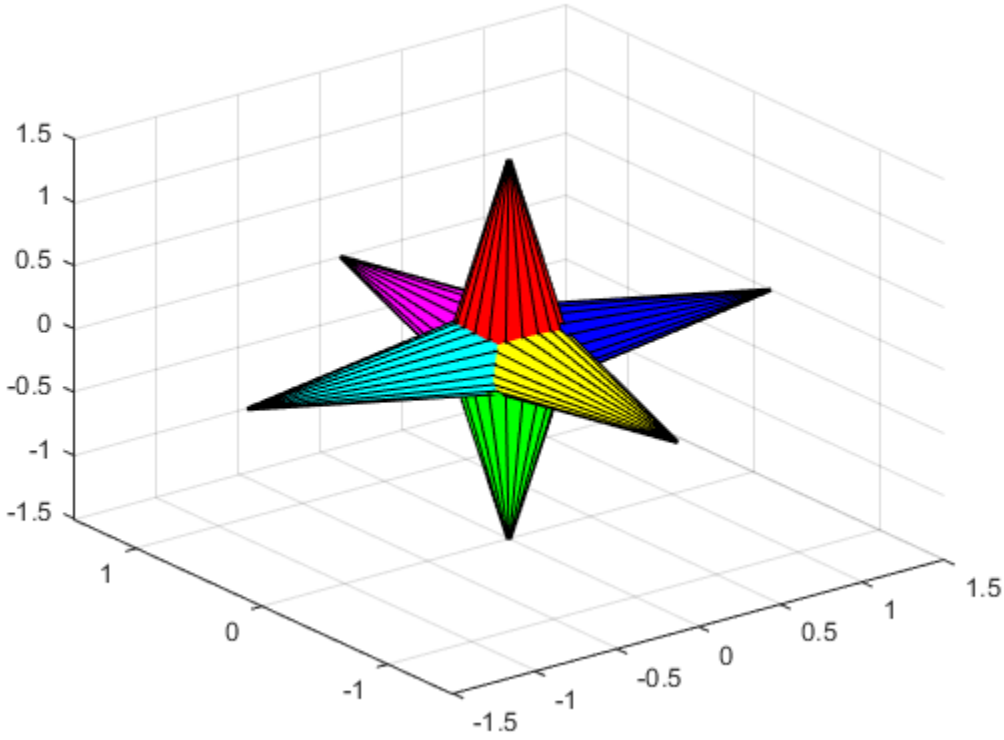
Create a transform object and parent the surface objects to it. Initialize the rotation and scaling matrix to the identity matrix (eye).

```
t = hgtransform('Parent',ax);
set(h,'Parent',t)

Rz = eye(4);
Sxy = Rz;
```



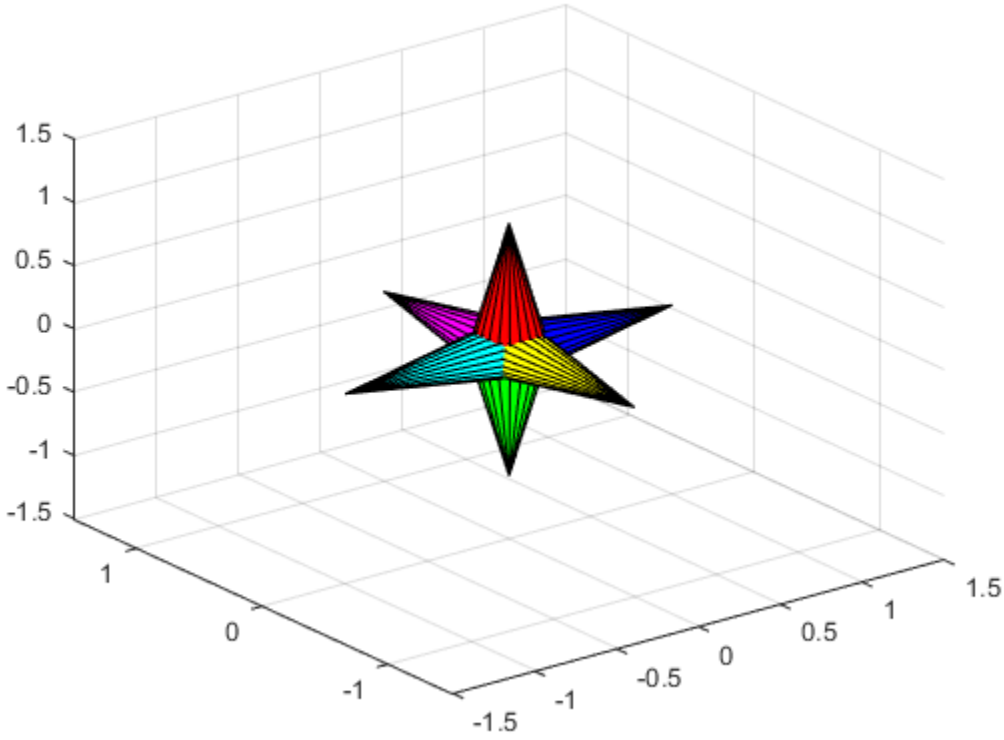
Form the *z*-axis rotation matrix and the scaling matrix. Rotate group and scale by using the increasing values of `r`.

```
for r = 1:.1:2*pi
    % Z-axis rotation matrix
    Rz = makehgtform('zrotate',r);
    % Scaling matrix
    Sxy = makehgtform('scale',r/4);
    % Concatenate the transforms and
    % set the transform Matrix property
    set(t,'Matrix',Rz*Sxy)
    drawnow
end
pause(1)
```



Reset to the original orientation and size using the identity matrix.

```
set(t,'Matrix',eye(4))
```

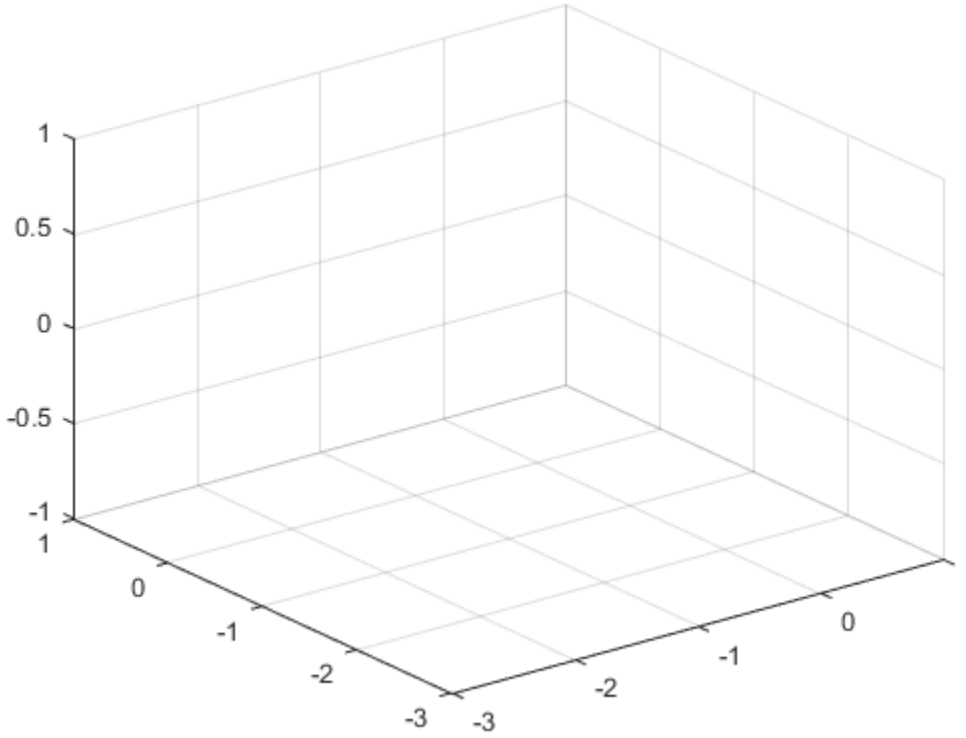


Transforming Objects Independently

This example creates two transform objects to illustrate how to transform each independently within the same axes. A translation transformation moves one transform object away from the origin.

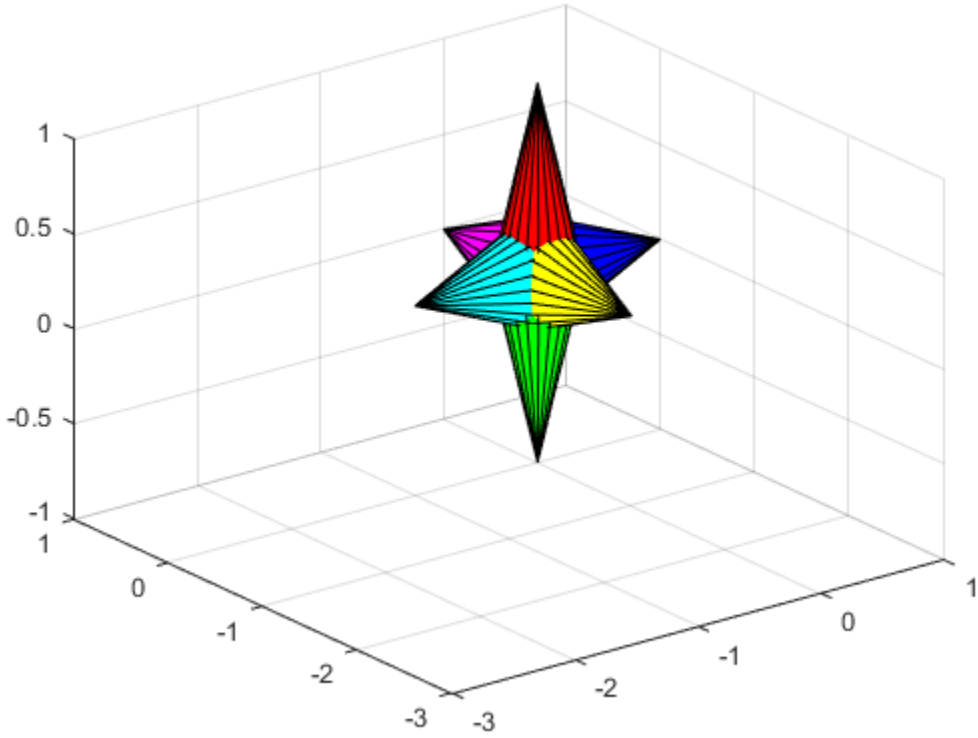
Create and set up the axes object that will be the parent of both transform objects. Set the limits to accommodate the translated object.

```
ax = axes('XLim',[-3 1],'YLim',[-3 1],'ZLim',[-1 1]);
view(3)
grid on
```



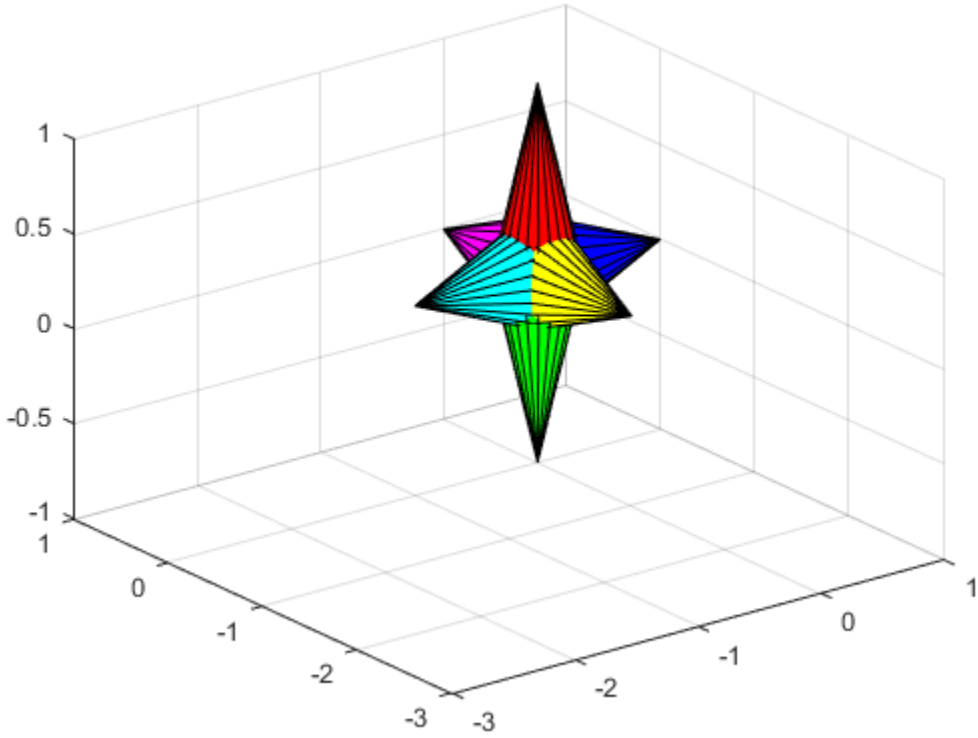
Create the surface objects to group.

```
[x,y,z] = cylinder([.3 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```



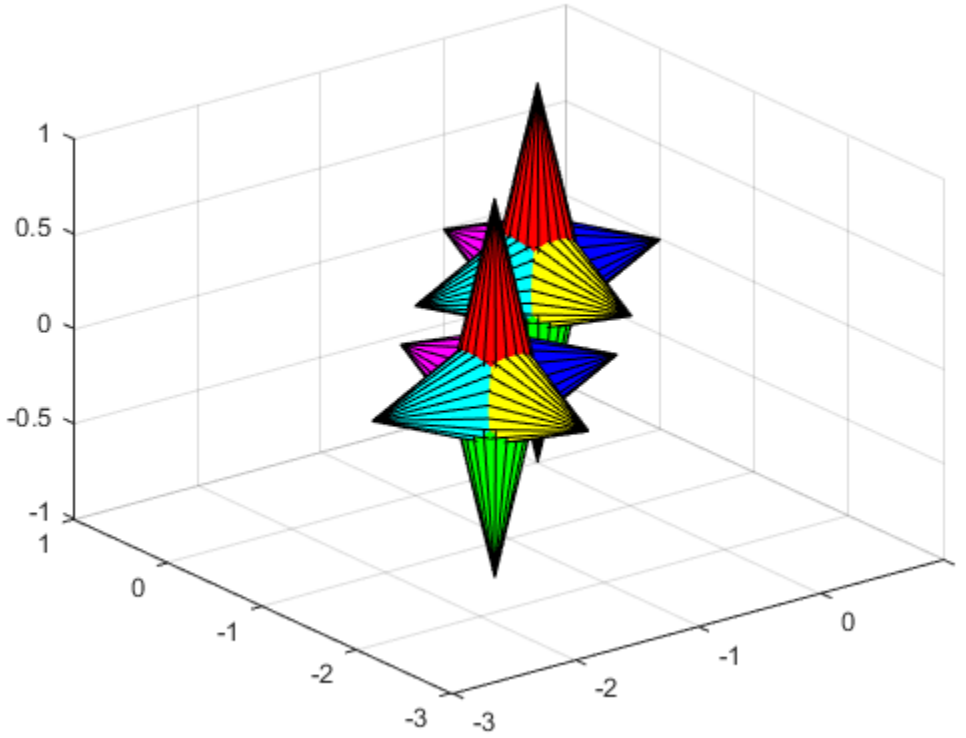
Create the transform objects and parent them to the same axes. Then, parent the surfaces to transform t1. Copy the surface objects and parent the copies to transform t2. This figure should not change.

```
t1 = hgtransform('Parent',ax);
t2 = hgtransform('Parent',ax);

set(h,'Parent',t1)
h2 = copyobj(h,t2);
```



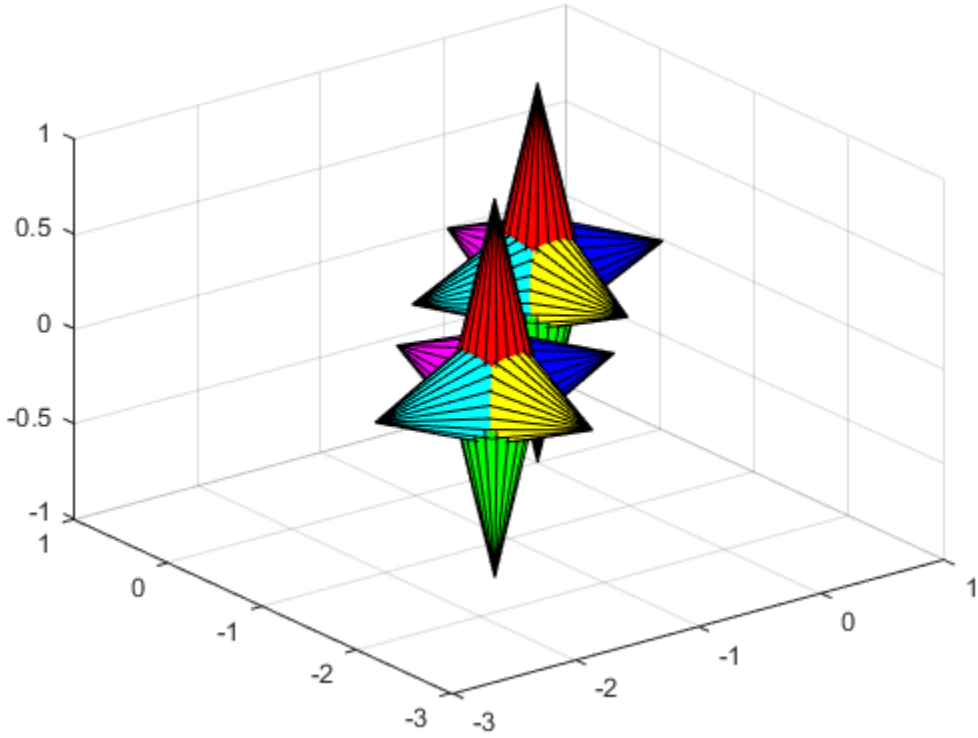
Translate the second transform object away from the first transform object and display the result.

```
Txy = makehgtform('translate',[-1.5 -1.5 0]);
set(t2,'Matrix',Txy)
drawnow
```



Rotate both transform objects in opposite directions.

```
% Rotate 10 times (2pi radians = 1 rotation)
for r = 1:.1:20*pi
    % Form z-axis rotation matrix
    Rz = makehgtform('zrotate',r);
    % Set transforms for both transform objects
    set(t1,'Matrix',Rz)
    set(t2,'Matrix',Txy*inv(Rz))
    drawnow
end
```



More About

- “Create Object Groups”

See Also

`hggroup` | `makehgtform`

Introduced before R2006a

Transform Properties

Control transform appearance and behavior

Transform properties control the behavior of transform objects. By changing property values, you can modify certain aspects of the transform object.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = hgtransform;
c = h.Children;
h.Matrix = makehgtform('scale',0.5);
```

If you are using an earlier release, use the `get` and `set` functions instead.

Transform Matrix

`Matrix` — Transform matrix

4-by-4 matrix

Transform matrix applied to the transform object and its children, specified as a 4-by-4 matrix. For more information about defining this matrix, see "Transforms Supported by hgtransform".

Data Types: `double`

Visibility

`Visible` — Visibility of transform

`'on'` (default) | `'off'`

Visibility of transform, specified as one of these values:

- `'on'` — Display all objects in the transform.
- `'off'` — Hide all objects in the transform. You still can access the properties of invisible transform objects. Setting the `Visible` property for the transform object does not change the `Visible` property for objects in the transform.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'hgtransform'`

Type of graphics object, returned as `'hgtransform'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Tag to associate with transform

`''` (default) | string

Tag to associate with the transform, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with transform

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the transform object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the transform.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the transform object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the transform from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the transform object in the legend as one entry (default).
   - `'off'` — Do not include the transform object in the legend.
   - `'children'` — Include only children of the transform object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of transform

axes object | group object | transform object

Parent of transform, specified as an axes, group, or transform object.

`Children` — Children of transform

empty `GraphicsPlaceholder` array | array of graphics objects

Children of transform, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property of the transform. To add a child to this list, set the `Parent` property of the child graphics object to the transform object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of transform object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The transform object handle is always visible.
- `'off'` — The transform object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The transform object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the transform at the command-line, but allows callback functions to access it.

If the transform object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the transform. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The transform object — You can access properties of the transform object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the transform. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the transform when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the transform.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Children that can capture mouse clicks

`'visible'` (default) | `'none'`

Children that can capture mouse clicks, specified as one of these values:

- `'visible'` — Any child object can capture a mouse click, depending on the `PickableParts` property value of the child.
- `'none'` — No child objects can capture mouse clicks, regardless of the `PickableParts` property value of the child.

`HitTest` — Response to mouse clicks captured by children

`'on'` (default) | `'off'`

Response to mouse clicks captured by children, specified one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of transform. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Do not trigger the callbacks of the transform. Instead, trigger the callbacks for the nearest ancestor of the transform that has a `HitTest` property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

A transform cannot capture mouse clicks. However, if you click a child of the transform and if the child has a `HitTest` property set to `'off'`, then the child passes the click to the transform.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the transform is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the

`BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the transform tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the transform. Setting the `CreateFcn` property on an existing transform has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during transform creation. MATLAB executes the callback after creating the transform and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The transform object — You can access properties of the transform object from within the callback function. You also can access the transform object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the transform. MATLAB executes the callback before destroying the transform so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The transform object — You can access properties of the transform object from within the callback function. You also can access the transform object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of transform

`'off'` (default) | `'on'`

Deletion status of transform, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the transform begins

execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the transform no longer exists.

Check the value of the BeingDeleted property to verify that the transform is not about to be deleted before querying or modifying it.

See Also

`hgtransform`

More About

- “Access Property Values”
- “Graphics Object Properties”

hidden

Remove hidden lines from mesh plot

Syntax

```
hidden on
hidden off
hidden
```

Description

Hidden line removal draws only those lines that are not obscured by other objects in a 3-D view. The `hidden` function only applies to surface plot objects that have a uniform `FaceColor`.

`hidden on` turns on hidden line removal for the current mesh plot so lines in the back of a mesh are hidden by those in front. This is the default behavior.

`hidden off` turns off hidden line removal for the current mesh plot.

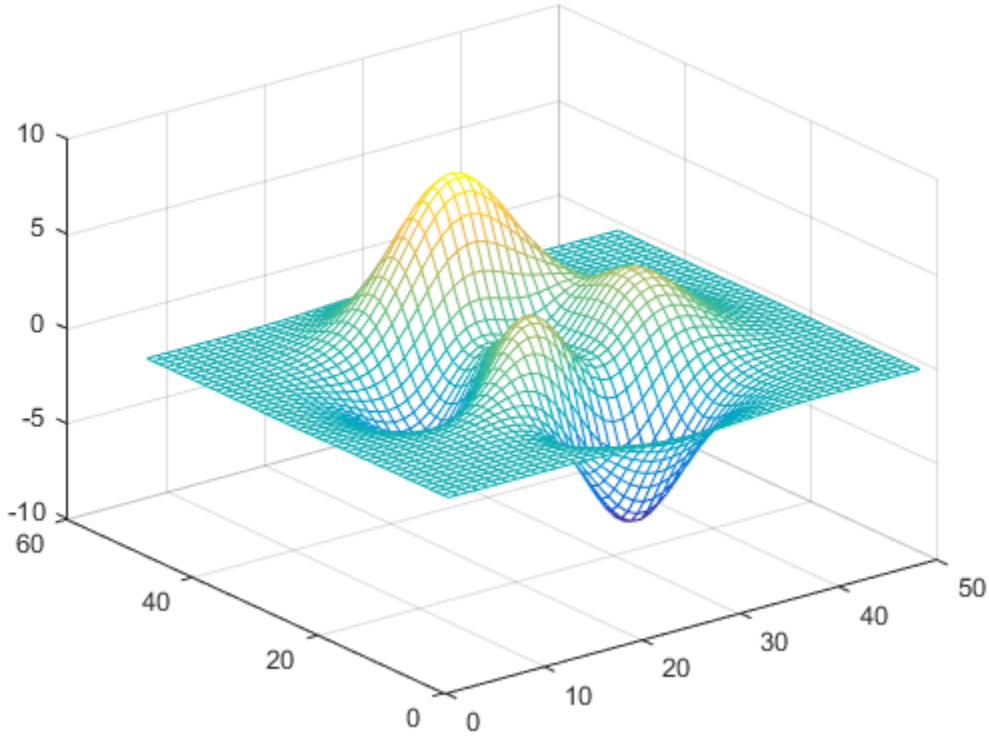
`hidden` toggles the hidden line removal state.

Examples

Show Obscured Lines

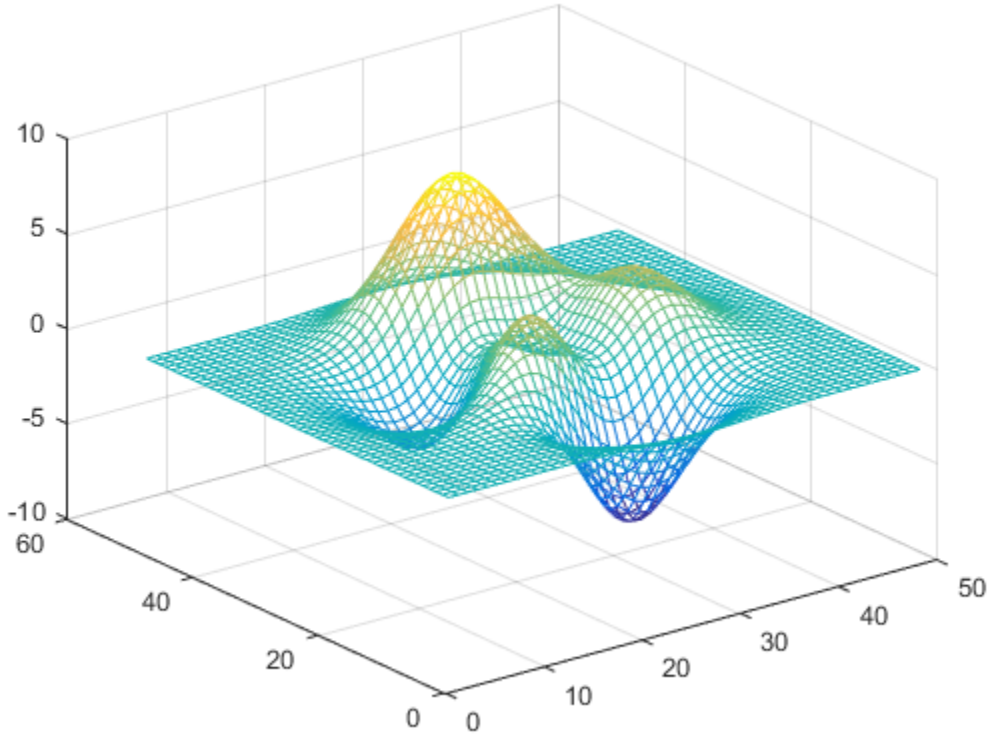
Create a mesh plot of the `peaks` function.

```
figure
mesh(peaks)
```



By default, MATLAB® hides obscured lines from the view. Show the obscured parts of the mesh by setting the hidden line removal to `off`.

```
hidden off
```



More About

Algorithms

When a surface graphics object has a uniform `FaceColor` matching the `Color` property of the axes, `hidden off` sets the `FaceColor` of the surface object to `'none'`.

`hidden on` sets the `FaceColor` property of such surface objects to match the axes `Color` property (or to match that of the figure, if axes `Color` is `'none'`).

See Also

`shading` | `mesh`

Introduced before R2006a

hilb

Hilbert matrix

Syntax

```
H = hilb(n)
```

Description

`H = hilb(n)` returns the Hilbert matrix of order `n`.

Definitions

The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i,j) = 1/(i + j - 1)$.

Examples

Even the fourth-order Hilbert matrix shows signs of poor conditioning.

```
cond(hilb(4)) =
    1.5514e+04
```

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

See Also

`invhilb`

Introduced before R2006a

hist

Histogram plot (not recommended; use `histogram`)

Compatibility

`hist` is not recommended for numeric histograms. Use `histogram` instead. Use `hist` to plot categorical histograms.

For more information, including suggestions on updating code, see "Replace Discouraged Instances of hist and histc".

Syntax

```
hist(x)
hist(x,nbins)
hist(x,xbins)

hist(ax, ___ )

counts = hist( ___ )
[counts,centers] = hist( ___ )
```

Description

`hist(x)` creates a histogram bar chart of the elements in vector `x`. The elements in `x` are sorted into 10 equally spaced bins along the *x*-axis between the minimum and maximum values of `x`. `hist` displays bins as rectangles, such that the height of each rectangle indicates the number of elements in the bin.

If the input is a multi-column array, `hist` creates histograms for each column of `x` and overlays them onto a single plot.

If the input is of data type `categorical`, each bin is a category of `x`.

`hist(x,nbins)` sorts x into the number of bins specified by the scalar nbins.

`hist(x,xbins)` sorts `x` into bins with intervals or categories determined by the vector `xbins`.

- If `xbins` is a vector of evenly spaced values, then `hist` uses the values as the bin centers.
- If `xbins` is a vector of unevenly spaced values, then `hist` uses the midpoints between consecutive values as the bin edges.
- If `x` is of data type `categorical`, then `xbins` must be a categorical vector or cell array of strings that specifies categories. `hist` plots bars only for those categories.

The length of the vector `xbins` is equal to the number of bins.

`hist(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

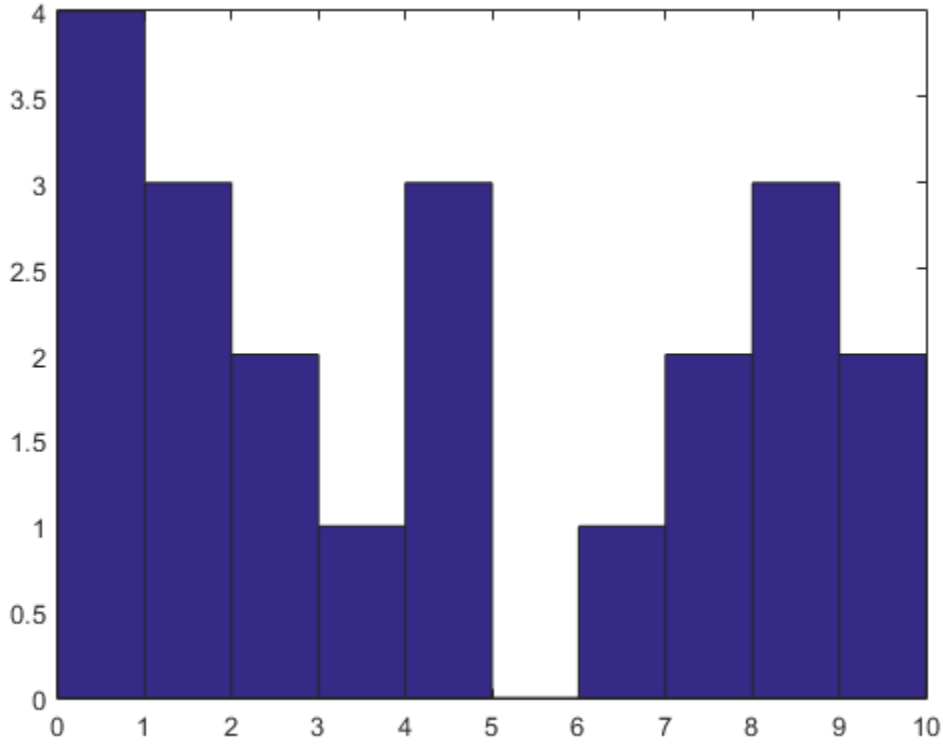
`counts = hist( ___ )` returns a row vector, counts, containing the number of elements in each bin.

`[counts,centers] = hist( ___ )` returns an additional row vector, centers, indicating the location of each bin center on the *x*-axis.

Examples

Histogram of Vector

```
x = [0 2 9 2 5 8 7 3 1 9 4 3 5 8 10 0 1 2 9 5 10];
hist(x)
```

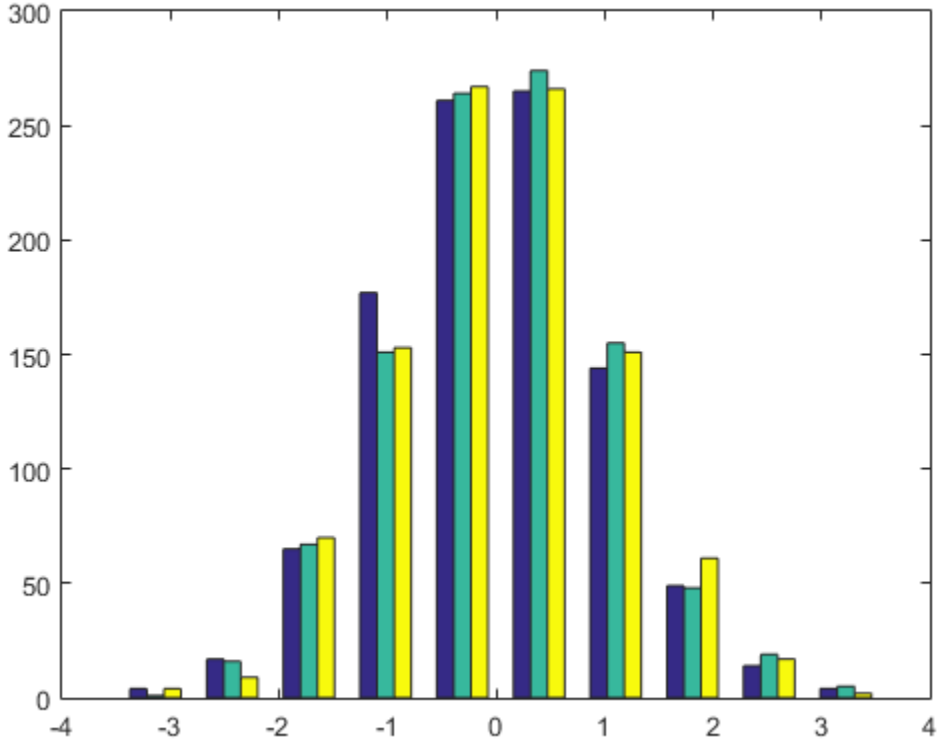


`hist` sorts the values in `x` among 10 equally spaced bins between the minimum and maximum values in the vector, which are 0 and 10 in this example.

Histogram of Multiple Columns

Generate three columns of 1,000 random numbers and plot the three column overlaid histogram.

```
x = randn(1000,3);
hist(x)
```

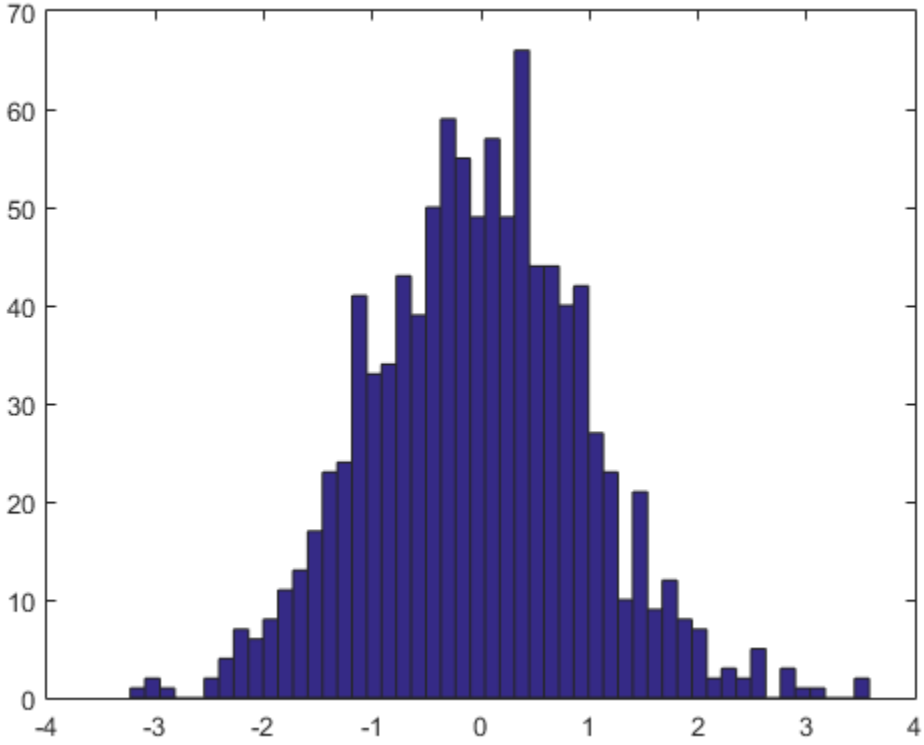


The values in `x` are sorted among 10 equally spaced bins between the minimum and maximum values. `hist` sorts and bins the columns of `x` separately and plots each column with a different color.

Specify Number of Histogram Bins

Plot a histogram of 1,000 random numbers sorted into 50 equally spaced bins.

```
x = randn(1000,1);
nbins = 50;
hist(x,nbins)
```

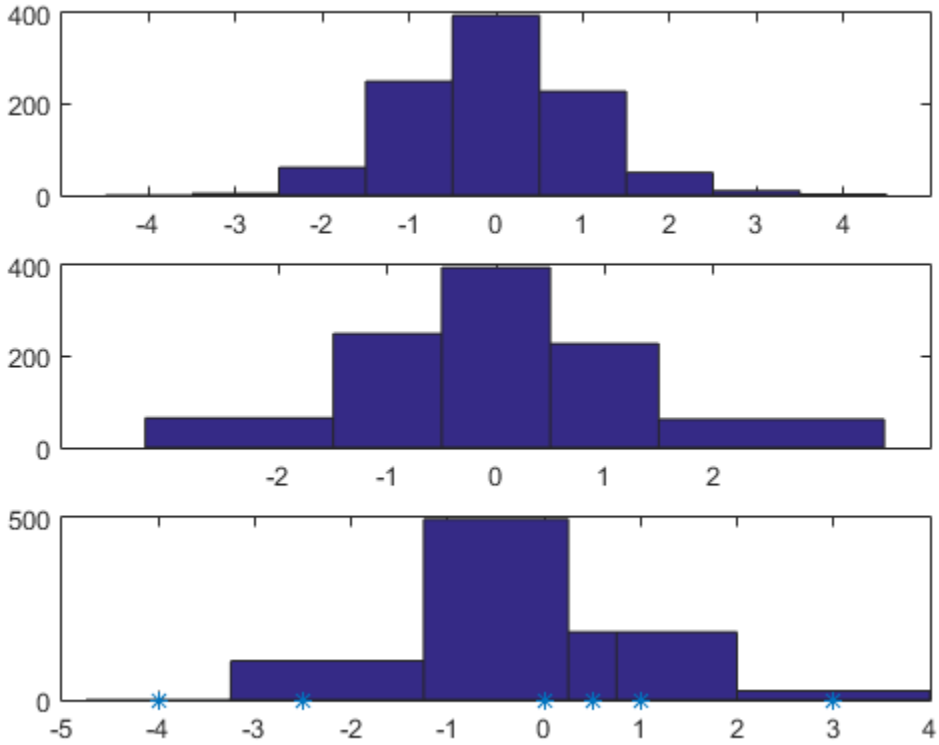


Specify Histogram Bin Intervals

Plot three histograms of the same data using different bin intervals:

- In the upper subplot, specify the bin centers using a vector of evenly spaced values that span the values in `x`.
- In the middle subplot, specify the bin centers using a vector of evenly spaced values that do not span the values in `x`. The first and last bins extend to cover the minimum and maximum values in `x`.
- In the lower subplot, specify the bin intervals using a vector of unevenly spaced values. The `hist` function uses the midpoints between consecutive values as the bin edges and indicates the specified values by markers along the *x*-axis.

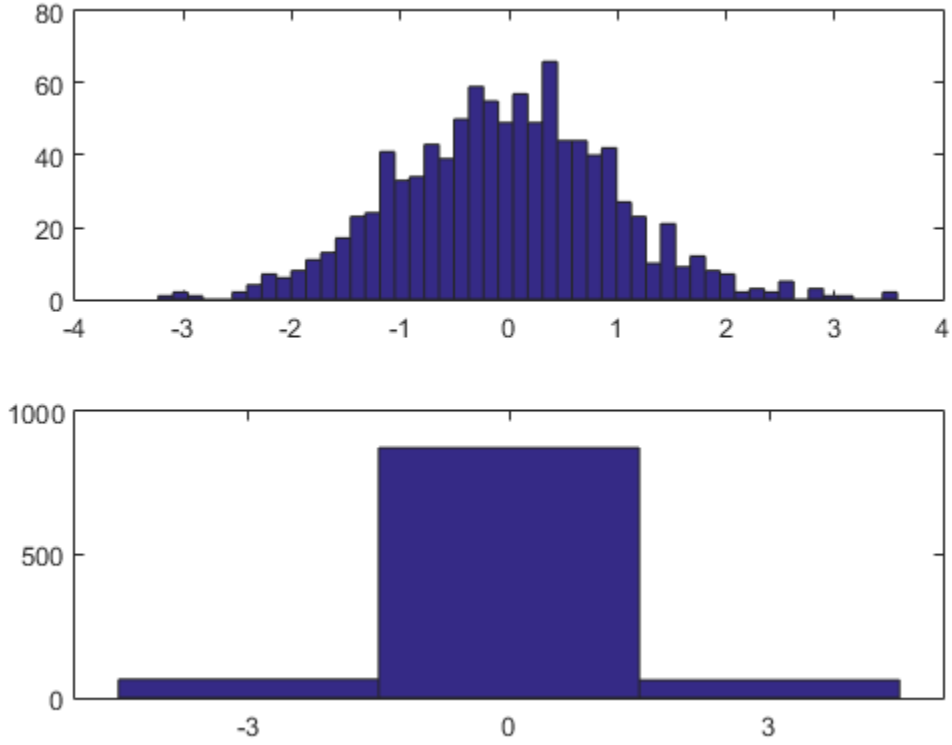
```
x = randn(1000,1);
subplot(3,1,1)
xbins1 = -4:4;
hist(x,xbins1)

subplot(3,1,2)
xbins2 = -2:2;
hist(x,xbins2)

subplot(3,1,3)
xbins3 = [-4 -2.5 0 0.5 1 3];
hist(x,xbins3)
```



Specify Histogram Axes

Create a figure with two subplots. In the upper subplot, plot a histogram of 1,000 random numbers sorted into 50 equally spaced bins. In the lower subplot, plot a histogram of the same data and use bins with centers at -3, 0, and 3.

```
x = randn(1000,1);
ax1 = subplot(2,1,1);
hist(ax1,x,50)

ax2 = subplot(2,1,2);
xbins = [-3 0 3];
hist(ax2,x,xbins)
```



Use hist to Calculate Only

Generate 1,000 random numbers. Count how many numbers are in each of 10 equally spaced bins. Return the bin counts and bin centers.

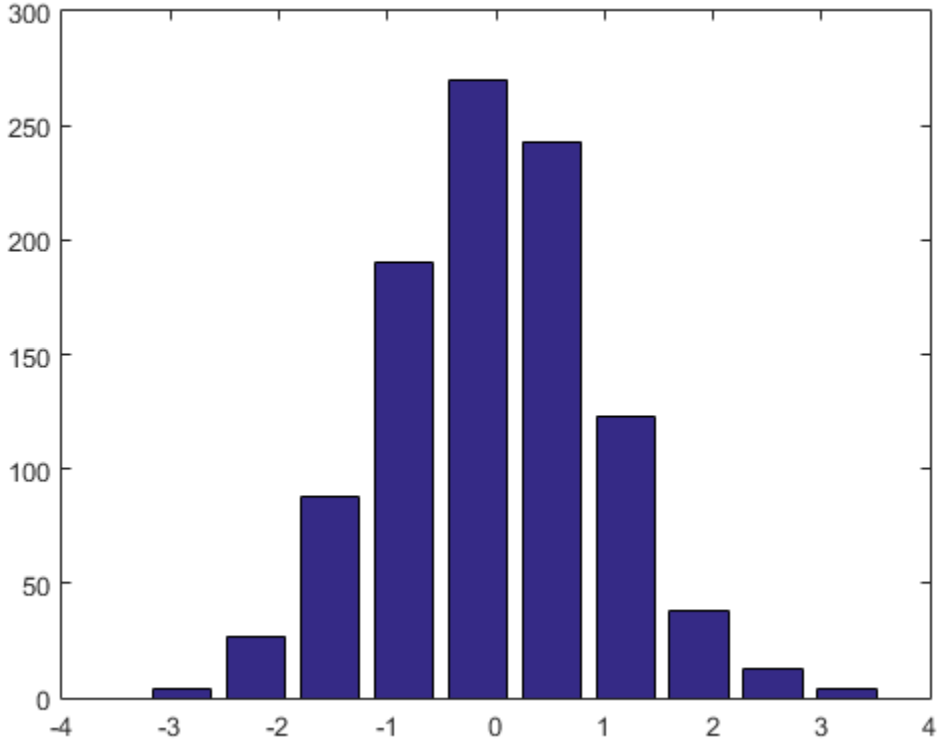
```
x = randn(1000,1);
[counts,centers] = hist(x)
```

```
counts =

     4    27    88   190   270   243   123    38    13     4
```

```
centers =

  Columns 1 through 7

   -2.8915   -2.2105   -1.5294   -0.8484   -0.1673    0.5137    1.1947

  Columns 8 through 10

    1.8758    2.5568    3.2379
```

Use `bar` to plot the histogram.

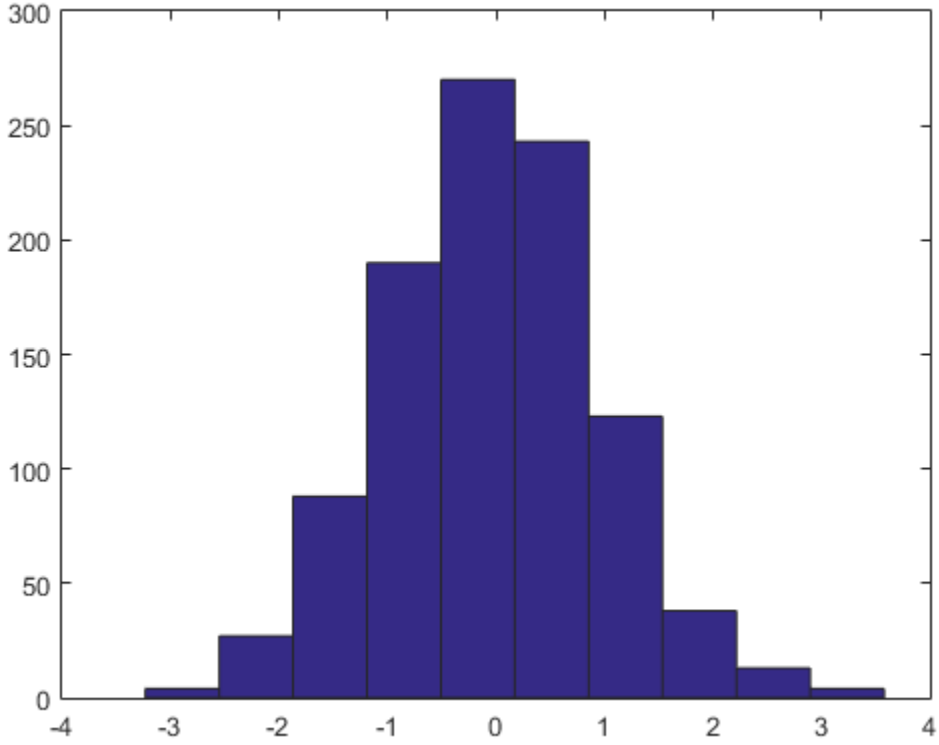
```
bar(centers,counts)
```



Specify Histogram Colors

Generate 1,000 random numbers and create a histogram.

```
data = randn(1000,1);
hist(data)
```

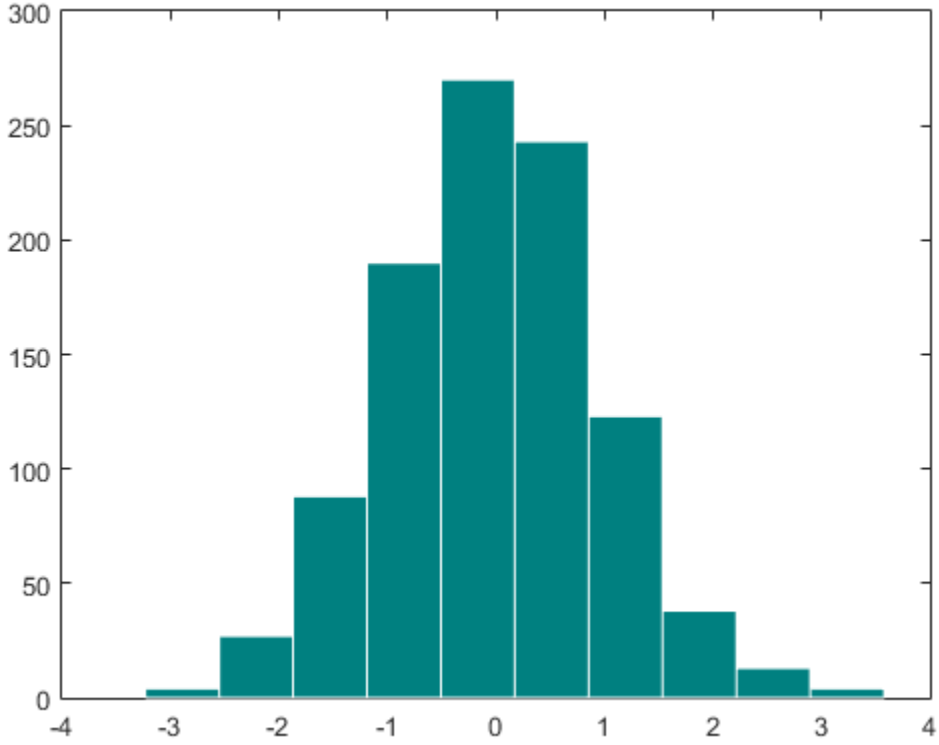


Get the handle to the patch object that creates the histogram plot.

```
h = findobj(gca,'Type','patch');
```

Set the face color of the bars plotted to an RGB triplet value of `[0 0.5 0.5]`. Set the edge color to white.

```
h.FaceColor = [0 0.5 0.5];
h.EdgeColor = 'w';
```



Input Arguments

x — Input array

vector or matrix

Input vector or matrix.

- If x is a vector, then `hist` creates one histogram.
- If x is a matrix, then `hist` creates a separate histogram for each column and plots the histograms using different colors.

If the input array contains `NaN`s or undefined categorical values, `hist` does not include these values in the bin counts.

If the input array contains the infinite values `-Inf` or `Inf`, then `hist` sorts `-Inf` into the first bin and `Inf` into the last bin. If you do not specify the bin intervals, then `hist` calculates the bin intervals using only the finite values in the input array.

Data Types: `single|double|logical|categorical`

`nbins` — Number of bins

10 (default) | scalar

Number of bins. Input `x` must be numeric, not categorical.

Data Types: `single|double|int8|int16|int32|int64|uint8|uint16|uint32|uint64`

`xbins` — Bin locations or categories

vector

Bin locations or categories, specified as a vector.

If x is numeric or logical, then xbins must be of type `single` or `double`.

- If the elements in xbins are equally spaced, then these elements are the bin centers.
- If the elements in xbins are not equally spaced, then these elements are indicated by markers along the *x*-axis, but are not the actual bin centers. Instead, `hist` calculates the bin edges as the midpoints between consecutive elements in vector `xbins`. To specify the bin edges directly, use `histc`.
- xbins must contain only finite values. The first and last bins extend to cover the minimum and maximum values in x.

If x is categorical, then xbins must be a categorical vector or cell array of strings that specifies categories. `hist` plots bars only for those categories specified by xbins.

`ax` — Axes object

axes object

Axes object. Use `ax` to plot the histogram in a specific axes instead of the current axes (`gca`).

Output Arguments

`counts` — Counts of the number of elements in each bin

row vector

Counts of the number of elements in each bin, returned as a row vector.

`centers` — Bin centers or categories

vector

Bin centers or categories, returned as a vector. If used with the syntax `[counts,centers] = hist(x,xbins)`, then the `centers` output has the same elements as the `xbins` input.

- If `x` is numeric or logical, then `centers` is a numeric row vector.
- If `x` is categorical, then `centers` is a cell array of strings.

See Also

`bar` | `histc` | `histcounts` | `histogram` | `mode` | `patch` | `rose` | `stairs`

Introduced before R2006a

histc

Histogram bin counts (not recommended; use `histcounts`)

Compatibility

`histc` is not recommended. Use `histcounts` instead.

For more information, including suggestions on updating code, see "Replace Discouraged Instances of hist and histc".

Syntax

```
bincounts = histc(x,binranges)
bincounts = histc(x,binranges,dim)
[bincounts,ind]= histc( ___ )
```

Description

`bincounts = histc(x,binranges)` counts the number of values in `x` that are within each specified bin range. The input, `binranges`, determines the endpoints for each bin. The output, `bincounts`, contains the number of elements from `x` in each bin.

- If `x` is a vector, then `histc` returns `bincounts` as a vector of histogram bin counts.
- If `x` is a matrix, then `histc` operates along each column of `x` and returns `bincounts` as a matrix of histogram bin counts for each column.

To plot the histogram, use `bar(binranges,bincounts,'histc')`.

`bincounts = histc(x,binranges,dim)` operates along the dimension `dim`.

`[bincounts,ind]= histc( ___ )` returns `ind`, an array the same size as `x` indicating the bin number that each entry in `x` sorts into. Use this syntax with any of the previous input argument combinations.

Examples

Create Histogram Plot

Initialize the random number generator to make the output of `randn` repeatable.

```
rng(0,'twister')
```

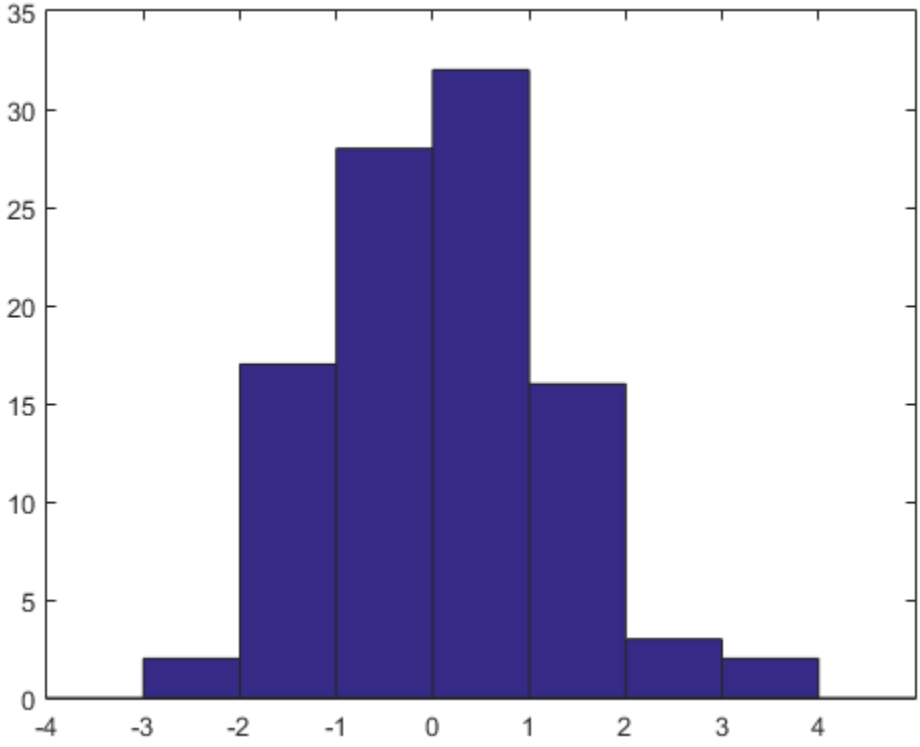
Define `x` as 100 normally distributed random numbers. Define bin ranges between -4 and 4. Determine the number of values in `x` that are within each specified bin range. Return the number of elements in each bin in `bincounts`.

```
x = randn(100,1);
binranges = -4:4;
[bincounts] = histc(x,binranges)
```

```
bincounts =

     0
     2
    17
    28
    32
    16
     3
     2
     0
```

To plot the histogram, use the `bar` function.

```
figure
bar(binranges,bincounts,'histc')
```



Return Bin Numbers for Histogram

Defined `ages` as a vector of ages. Sort `ages` into bins with varying ranges between 0 and 75.

```
ages = [3,12,24,15,5,74,23,54,31,23,64,75];
binranges = [0,10,25,50,75];

[bincounts,ind] = histc(ages,binranges)
```

```
bincounts =

     2     5     1     3     1
```

```
ind =

     1     2     2     2     1     4     2     4     3     2     4     5
```

`bincounts` contains the number of values in each bin. `ind` indicates the bin numbers.

Input Arguments

`x` — Values to be sorted

vector | matrix

Values to be sorted, specified as a vector or a matrix. The bin counts do not include values in `x` that are `NaN` or that lie outside the specified bin ranges. If `x` contains complex values, then `histc` ignores the imaginary parts and uses only the real parts.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`binranges` — Bin ranges

vector | matrix

Bin ranges, specified as a vector of monotonically nondecreasing values or a matrix of monotonically nondecreasing values running down each successive column. The values in `binranges` determine the left and right endpoints for each bin. If `binranges` contains complex values, then `histc` ignores the imaginary parts and uses only the real parts.

If `binranges` is a matrix, then `histc` determines the bin ranges by using values running down successive columns. Each bin includes the left endpoint, but does not include the right endpoint. The last bin consists of the scalar value equal to last value in `binranges`.

For example, if `binranges` equals the vector `[0,5,10,13]`, then `histc` creates four bins. The first bin includes values greater than or equal to 0 and strictly less than 5. The second bin includes values greater than or equal to 5 and less than 10, and so on. The last bin contains the scalar value 13.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`dim` — Dimension along which to operate

scalar

Dimension along which to operate, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

`bincounts` — Number of elements in each bin

vector | matrix

Number of elements in each bin, returned as a vector or a matrix that is the same size as `x`. The last entry in `bincounts` is the number of values in `x` that equal the last entry in `binranges`.

`ind` — Bin index numbers

vector | matrix

Bin index numbers, returned as a vector or a matrix that is the same size as `x`.

More About

Tips

- If values in `x` lie outside the specified bin ranges, then `histc` does not include these values in the bin counts. Start and end the `binranges` vector with `-inf` and `inf` to ensure that all values in `x` are included in the bin counts.

See Also

`bar` | `hist` | `histcounts` | `histogram` | `mode`

Introduced before R2006a

histogram

Histogram plot

Syntax

```
histogram(X)
histogram(X,nbins)
histogram(X,edges)
histogram( ___ ,Name,Value)
histogram(ax, ___ )
h = histogram( ___ )
```

Description

`histogram(X)` creates a histogram plot of `X`. The `histogram` function uses an automatic binning algorithm that returns bins with a uniform width, chosen to cover the range of elements in `X` and reveal the underlying shape of the distribution. The bins display as rectangles such that the height of each rectangle indicates the number of elements in the bin.

`histogram(X,nbins)` uses a number of bins specified by the scalar, `nbins`.

`histogram(X,edges)` sorts `X` into bins with the bin edges specified by the vector, `edges`. Each bin includes the left edge, but does not include the right edge, except for the last bin which includes both edges.

`histogram( ___ ,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments using any of the previous syntaxes. For example, you can specify `'BinWidth'` and a scalar to adjust the width of the bins, or `'Normalization'` with a valid string (`'count'`, `'probability'`, `'countdensity'`, `'pdf'`, `'cumcount'`, or `'cdf'`) to use a different type of normalization.

`histogram(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`h = histogram( ___ )` also returns a histogram object. Use this to inspect and adjust the properties of the histogram.

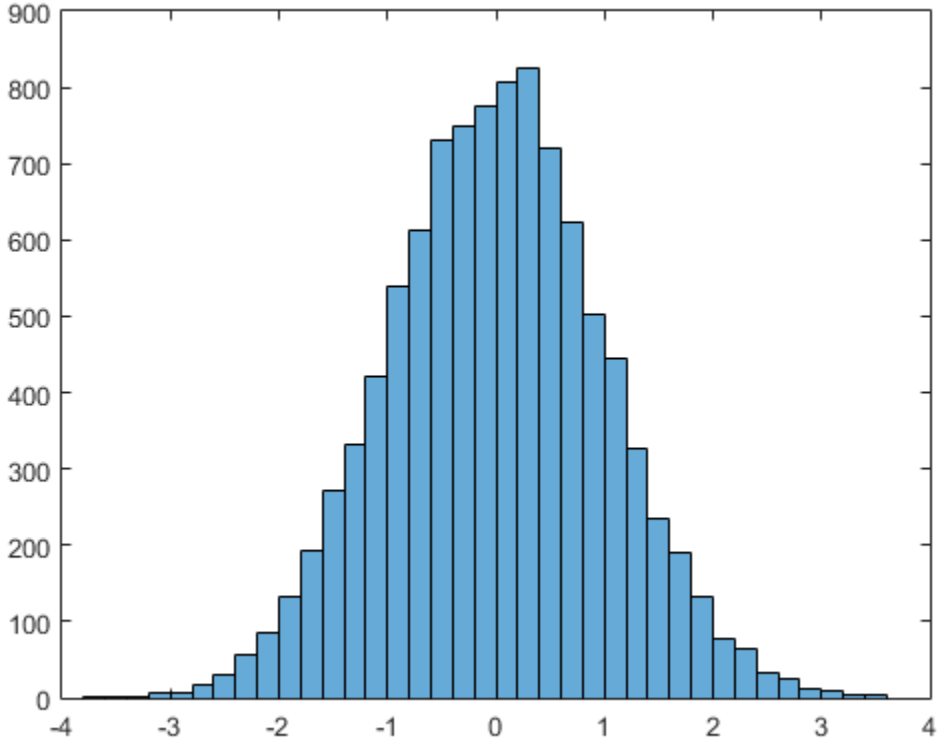
Examples

Histogram of Vector

Generate 10,000 random numbers and create a histogram. The `histogram` function automatically chooses an appropriate number of bins to cover the range of values in `x` and show the shape of the underlying distribution.

```
x = randn(10000,1);
h = histogram(x)
```

```
h =

  Histogram with properties:

            Data: [10000x1 double]
          Values: [1x37 double]
         NumBins: 37
        BinEdges: [1x38 double]
        BinWidth: 0.2000
       BinLimits: [-3.8000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]

  Use GET to show all properties
```



When you specify an output argument to the `histogram` function, it returns a histogram object. You can use this object to inspect the properties of the histogram, such as the number of bins or the width of the bins.

Find the number of histogram bins.

```
nbins = h.NumBins
```

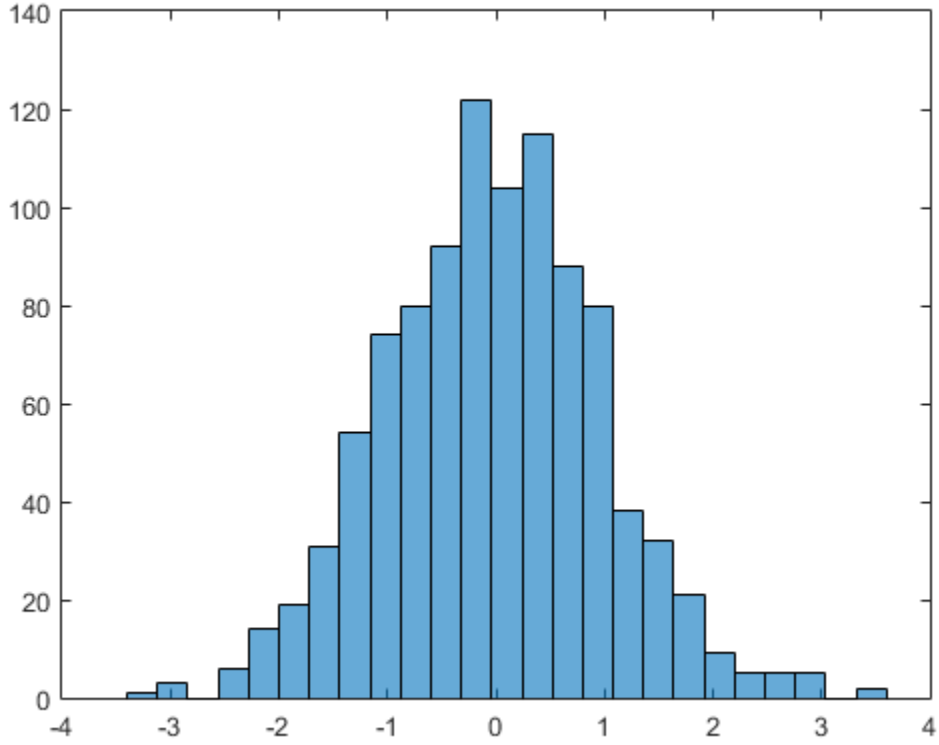
```
nbins =

    37
```

Specify Number of Histogram Bins

Plot a histogram of 1,000 random numbers sorted into 25 equally spaced bins.

```
x = randn(1000,1);
nbins = 25;
h = histogram(x,nbins)
```

```
h =

  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x25 double]
         NumBins: 25
        BinEdges: [1x26 double]
        BinWidth: 0.2800
       BinLimits: [-3.4000 3.6000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]

  Use GET to show all properties
```



Find the bin counts.

```
counts = h.Values
```

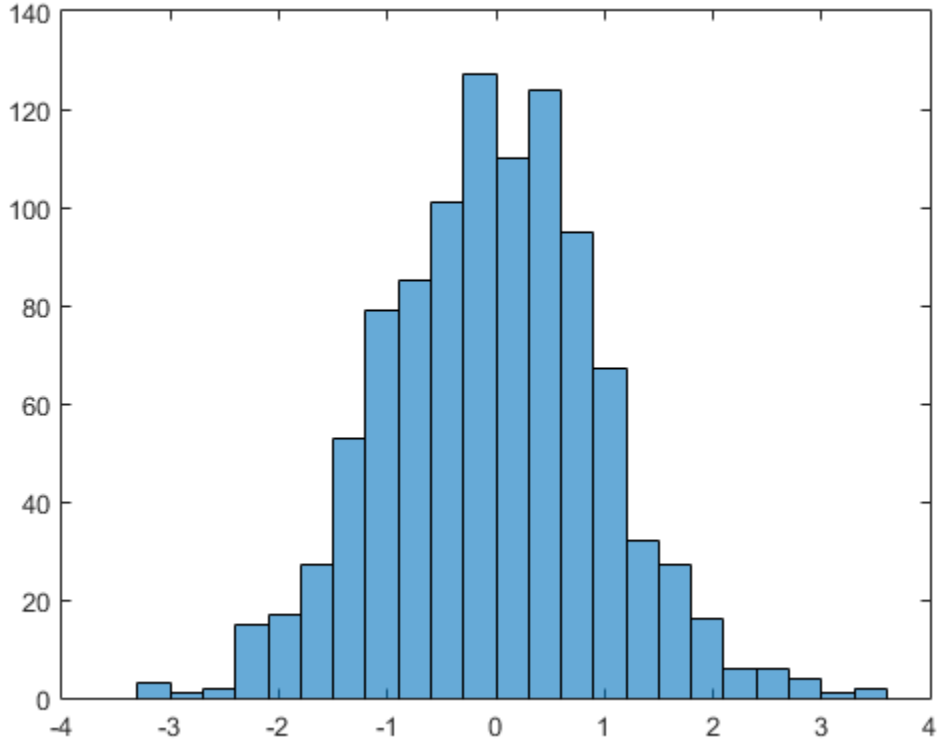
```
counts =

  Columns 1 through 13

     1     3     0     6    14    19    31    54    74    80    92   122   104

  Columns 14 through 25

   115    88    80    38    32    21     9     5     5     5     0     2
```

Change Number of Histogram Bins

Generate 1,000 random numbers and create a histogram.

```
X = randn(1000,1);
h = histogram(X)
```

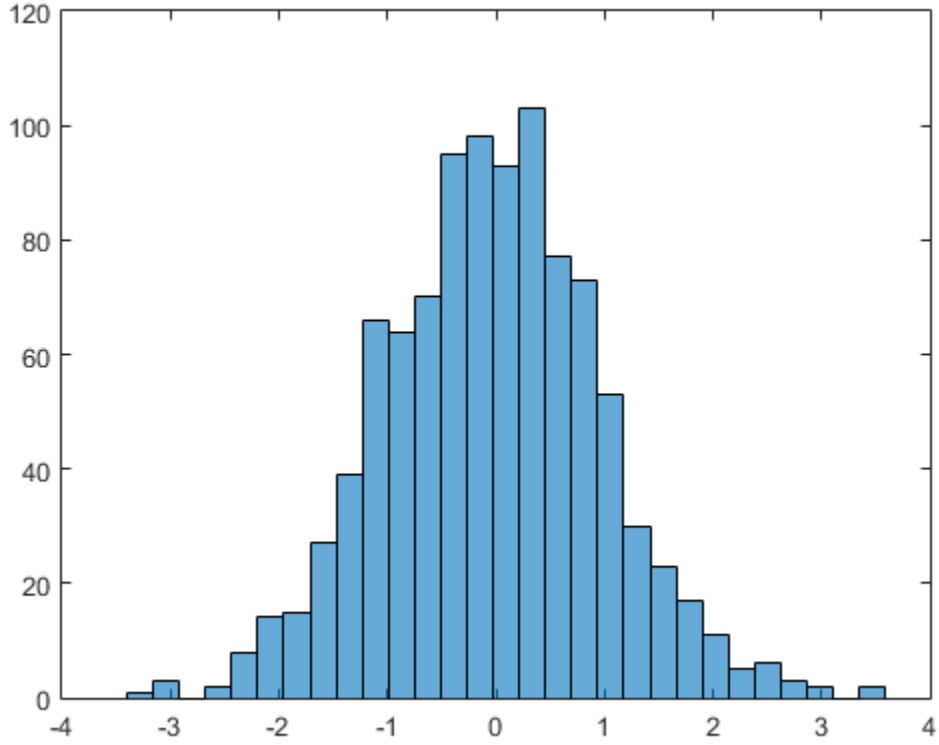
```
h =

  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]

  Use GET to show all properties
```



Use the `morebins` function to coarsely adjust the number of bins.

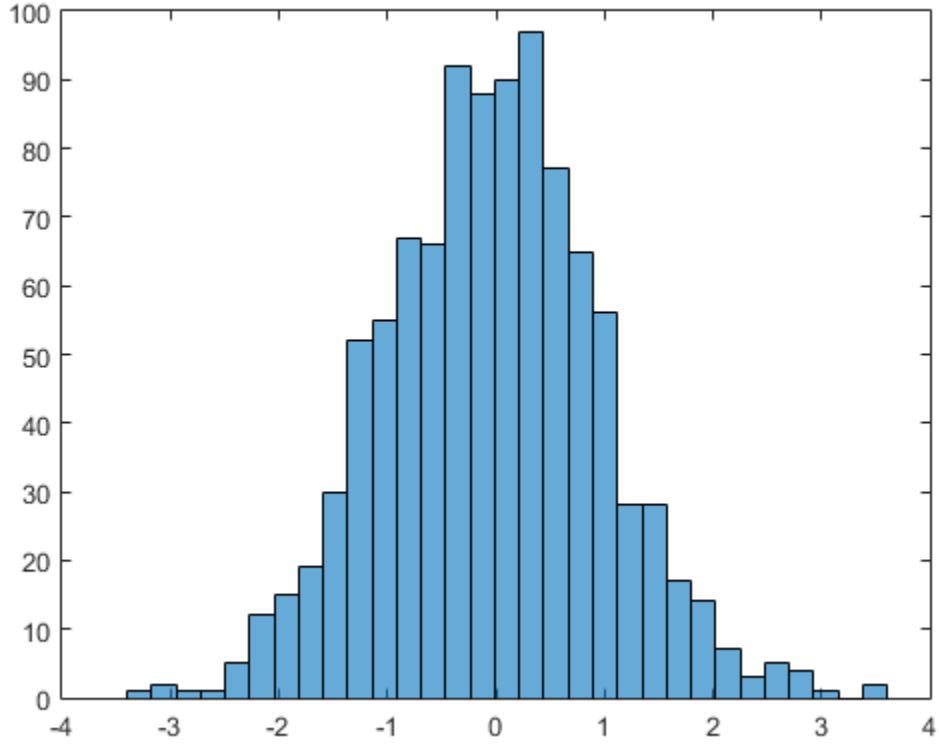
```
Nbins = morebins(h);
Nbins = morebins(h)
```

```
Nbins =

    29
```



Adjust the bins at a fine grain level by explicitly setting the number of bins.

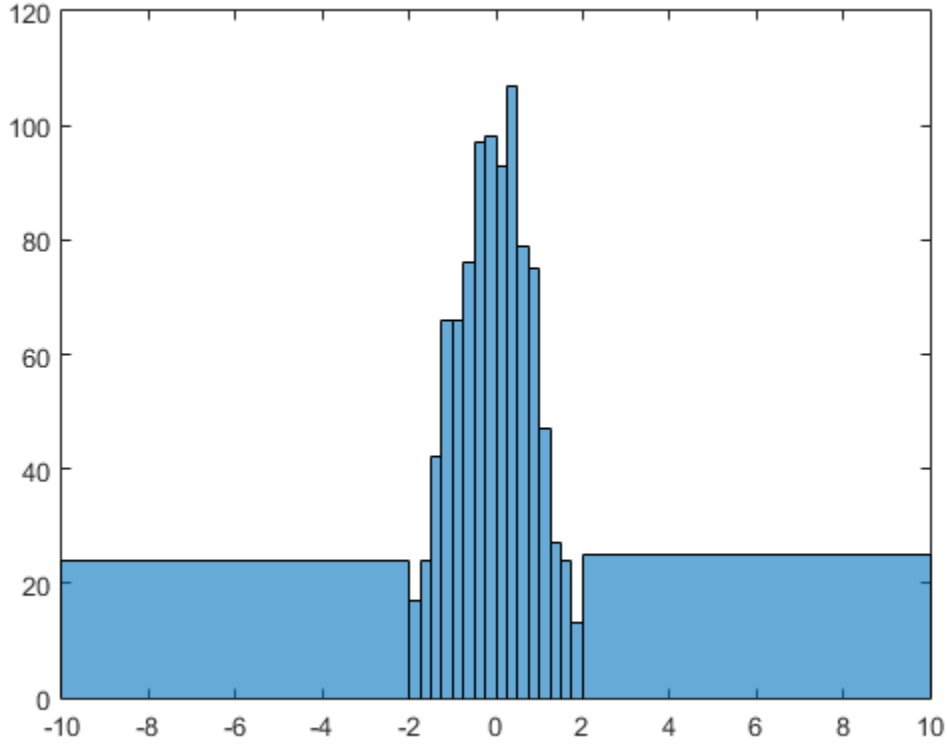
```
h.NumBins = 31;
```



Specify Bin Edges of Histogram

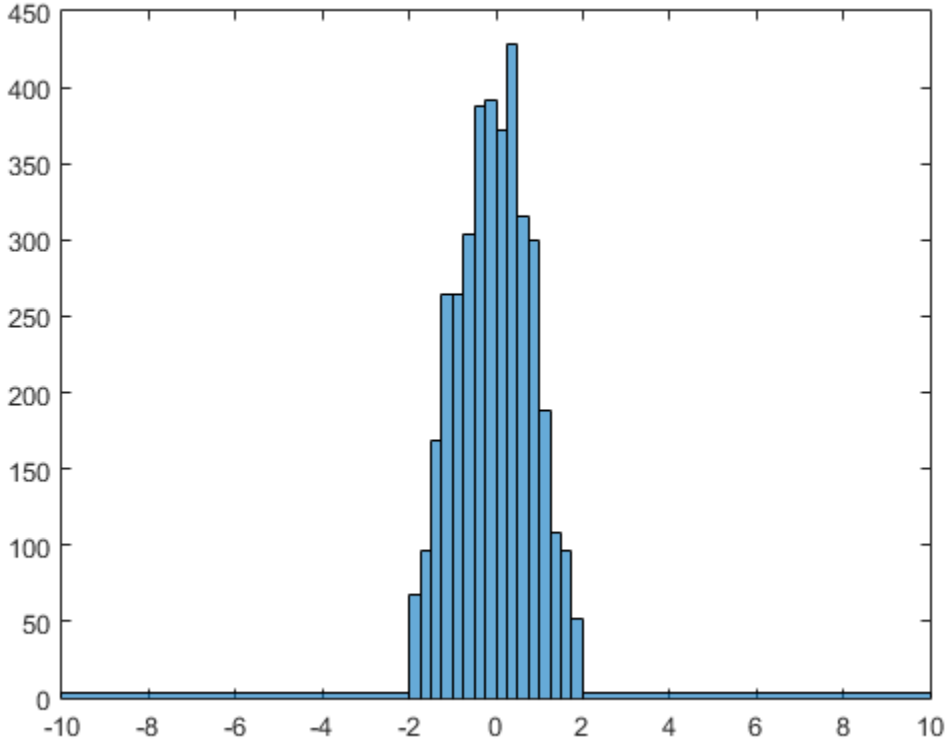
Generate 1,000 random numbers and create a histogram. Specify the bin edges as a vector with wide bins on the edges of the histogram to capture the outliers that do not satisfy $|x| < 2$. The first vector element is the left edge of the first bin, and the last vector element is the right edge of the last bin.

```
x = randn(1000,1);
edges = [-10 -2:0.25:2 10];
h = histogram(x,edges);
```



Specify the `Normalization` property as `'countdensity'` to flatten out the bins containing the outliers. Now, the *area* of each bin (rather than the height) represents the frequency of observations in that interval.

```
h.Normalization = 'countdensity';
```



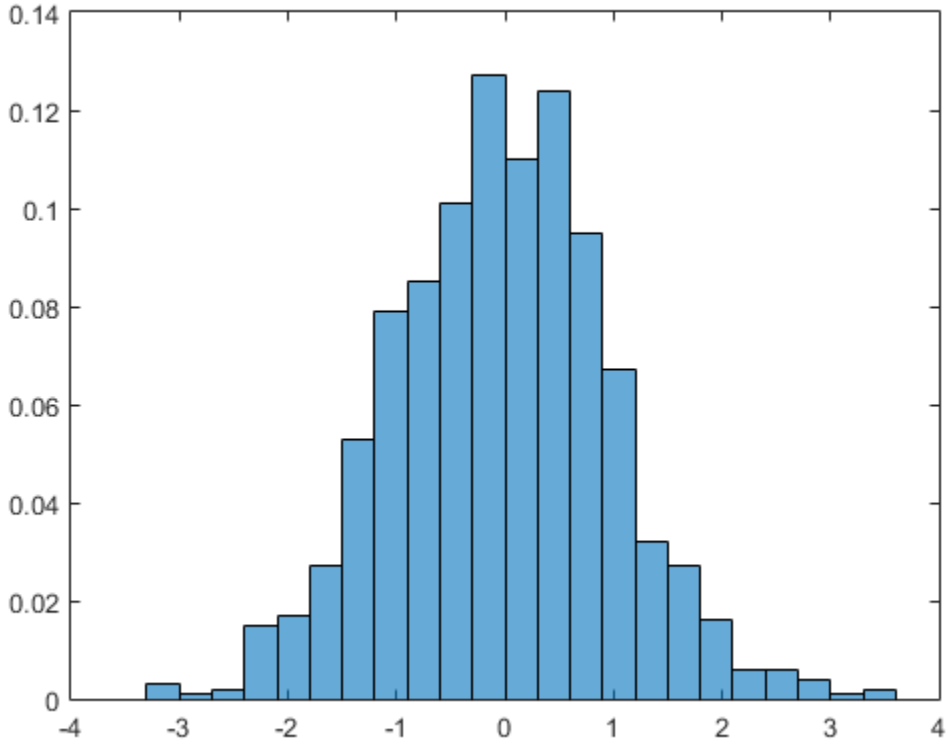
Histogram with Specified Normalization

Generate 1,000 random numbers and create a histogram using the `'probability'` normalization.

```
x = randn(1000,1);
h = histogram(x,'Normalization','probability')
```

```
h =

  Histogram with properties:

            Data: [1000x1 double]
```

```
          Values: [1x23 double]
         NumBins: 23
       BinEdges: [1x24 double]
       BinWidth: 0.3000
      BinLimits: [-3.3000 3.6000]
  Normalization: 'probability'
      FaceColor: 'auto'
      EdgeColor: [0 0 0]

  Use GET to show all properties
```



Compute the sum of the bar heights. With this normalization, the height of each bar is equal to the probability of selecting an observation within that bin interval, and the height of all of the bars sums to 1.

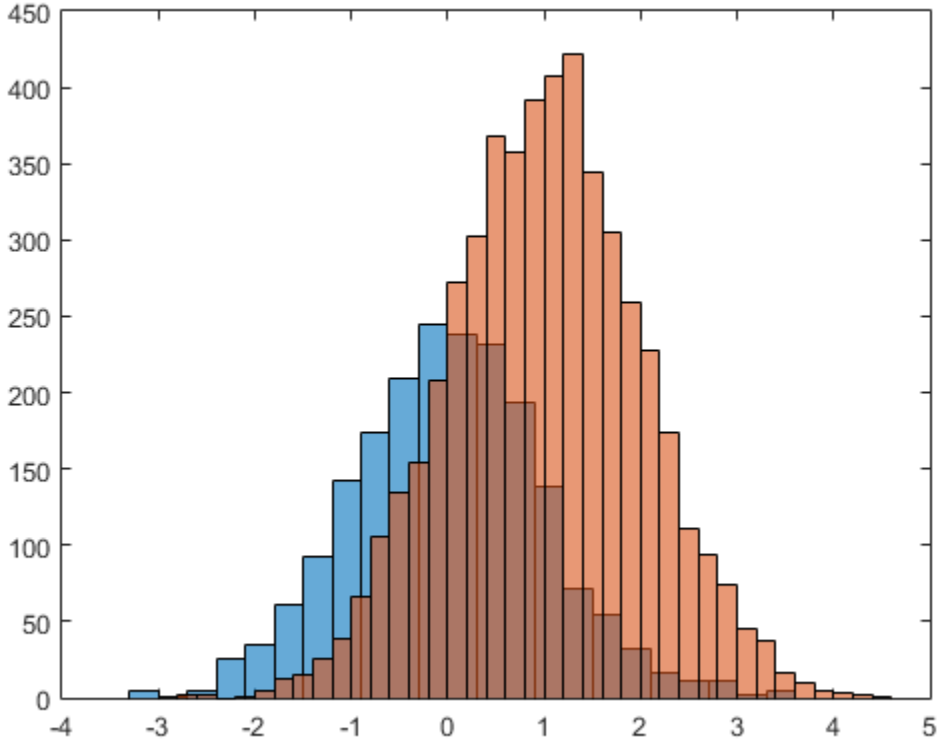
```
S = sum(h.Values)
```

```
S =

     1
```

Plot Multiple Histograms

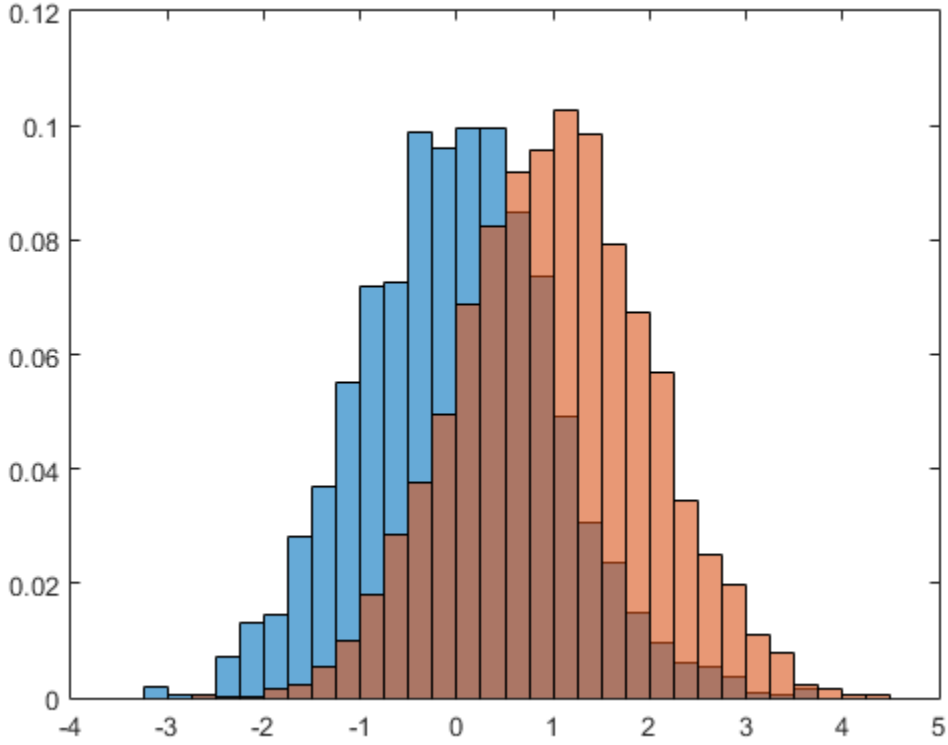
Generate two vectors of random numbers and plot a histogram for each vector in the same figure.

```
x = randn(2000,1);
y = 1 + randn(5000,1);
h1 = histogram(x);
hold on
h2 = histogram(y);
```



Since the sample size and bin width of the histograms are different, it is difficult to compare them. Normalize the histograms so that all of the bar heights add to 1, and use a uniform bin width.

```
h1.Normalization = 'probability';
h1.BinWidth = 0.25;
h2.Normalization = 'probability';
h2.BinWidth = 0.25;
```



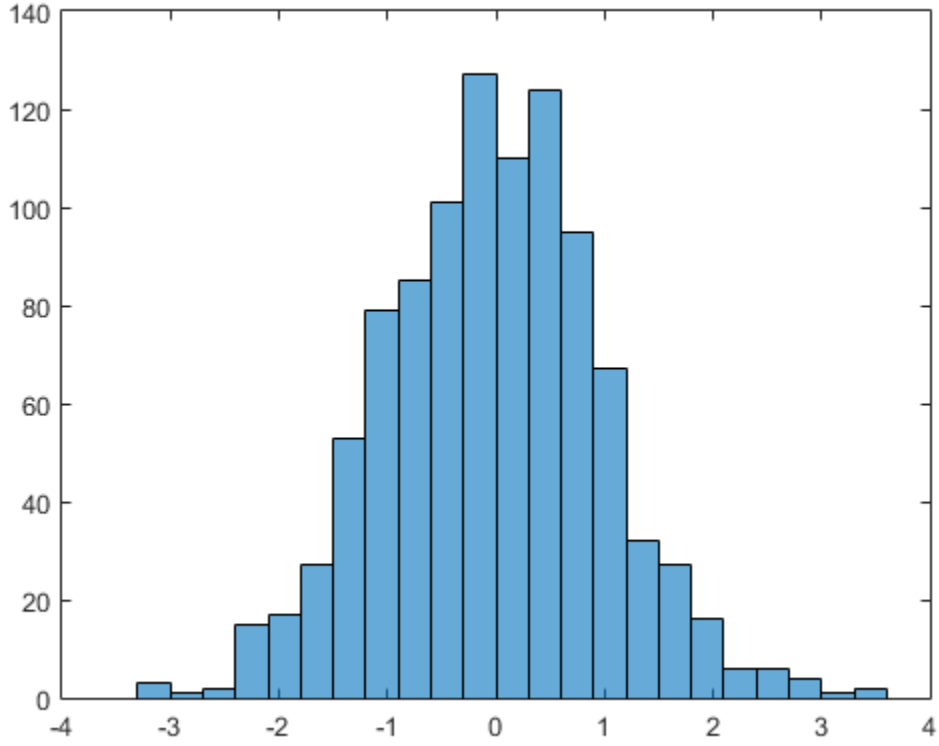
Adjust Histogram Properties

Generate 1,000 random numbers and create a histogram. Return the histogram object to adjust the properties of the histogram without recreating the entire plot.

```
x = randn(1000,1);
h = histogram(x)
```

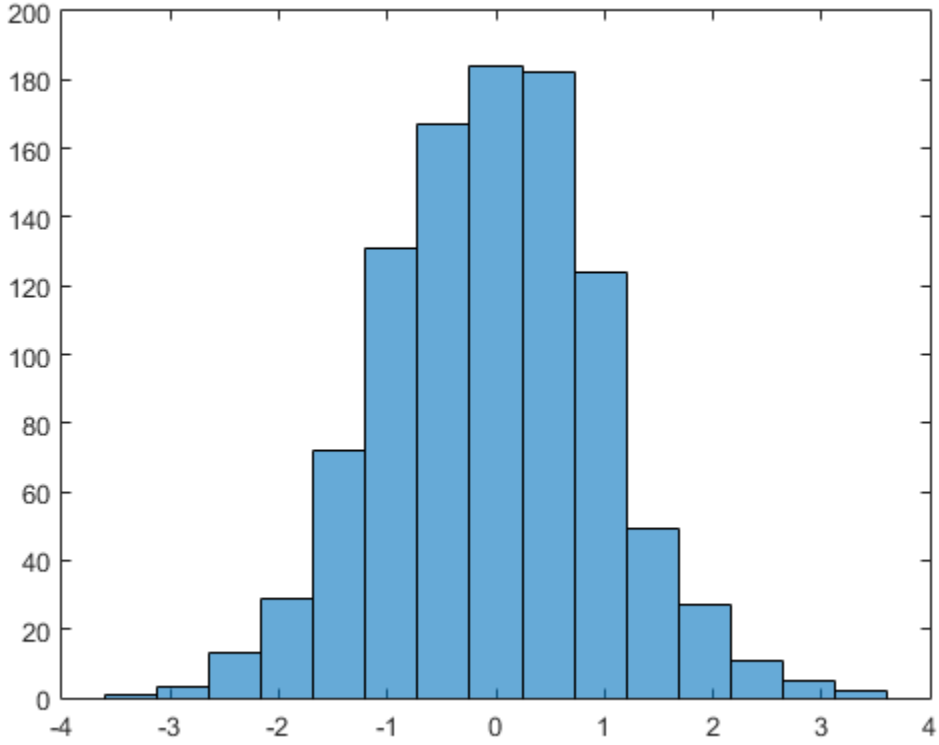
```
h =

  Histogram with properties:

            Data: [1000x1 double]
```

```
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]

  Use GET to show all properties
```



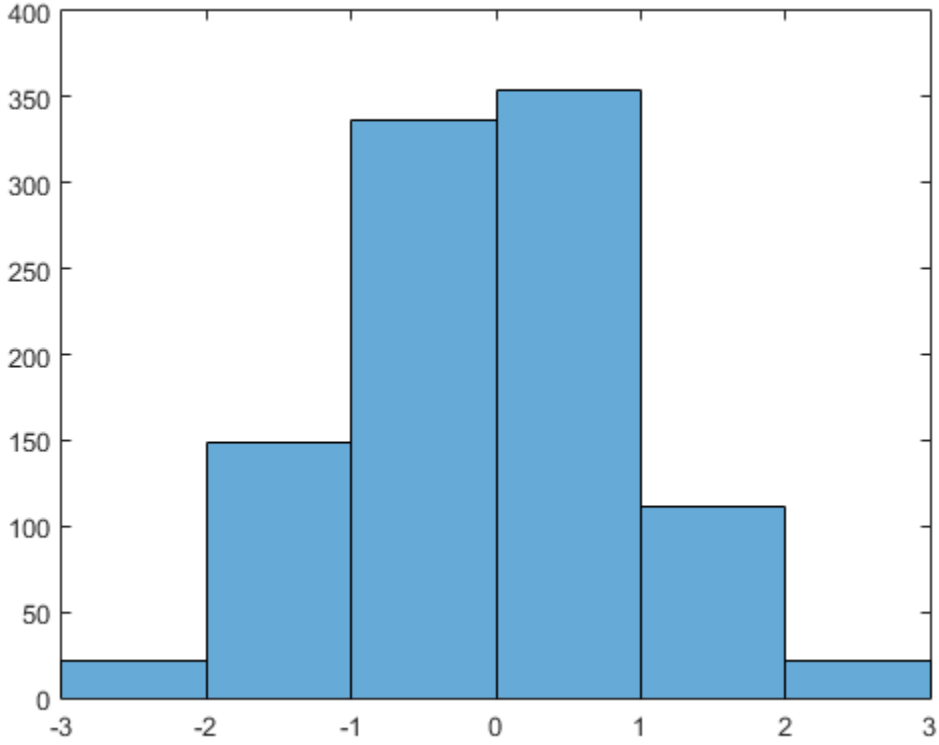
Specify exactly how many bins to use.

```
h.NumBins = 15;
```



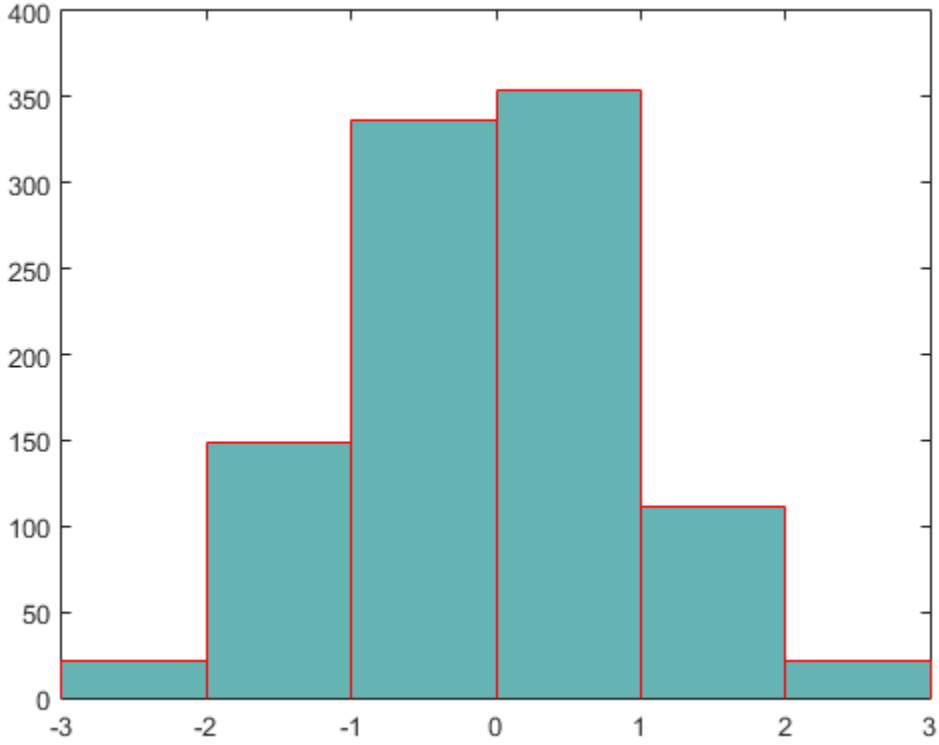
Specify the edges of the bins with a vector. The first value in the vector is the left edge of the first bin. The last value is the right edge of the last bin.

```
h.BinEdges = [-3:3];
```



Change the color of the histogram bars.

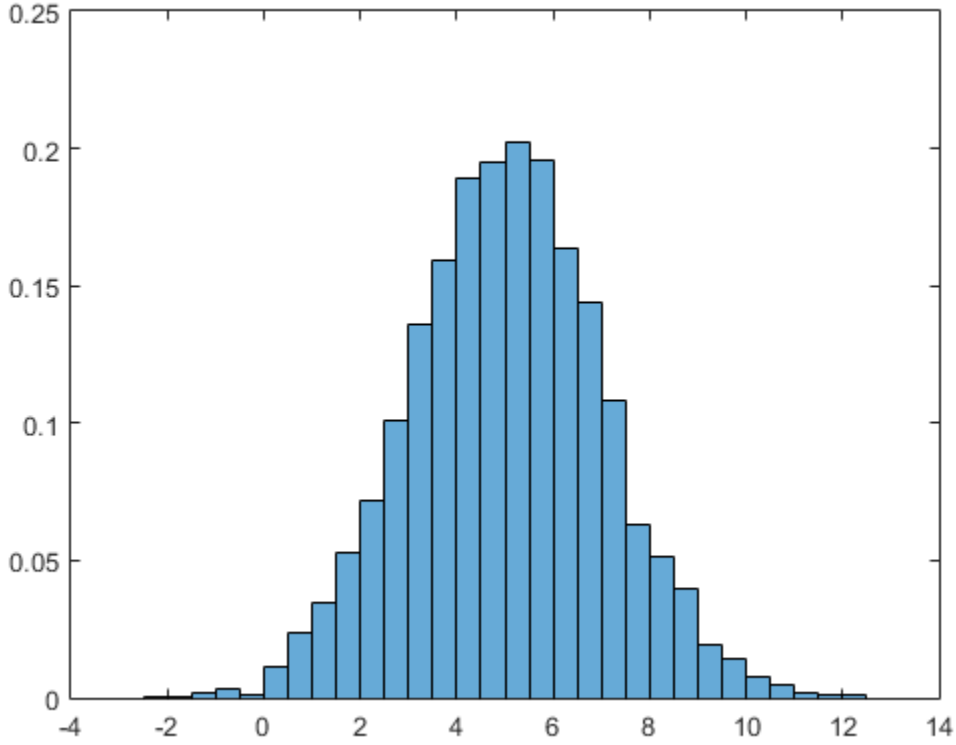
```
h.FaceColor = [0 0.5 0.5];
h.EdgeColor = 'r';
```



Determine Underlying Probability Distribution

Generate 5,000 normally distributed random numbers with a mean of 5 and a standard deviation of 2. Plot a histogram with `Normalization` set to `'pdf'` to produce an estimation of the probability density function.

```
x = 2*randn(5000,1) + 5;
histogram(x,'Normalization','pdf')
```



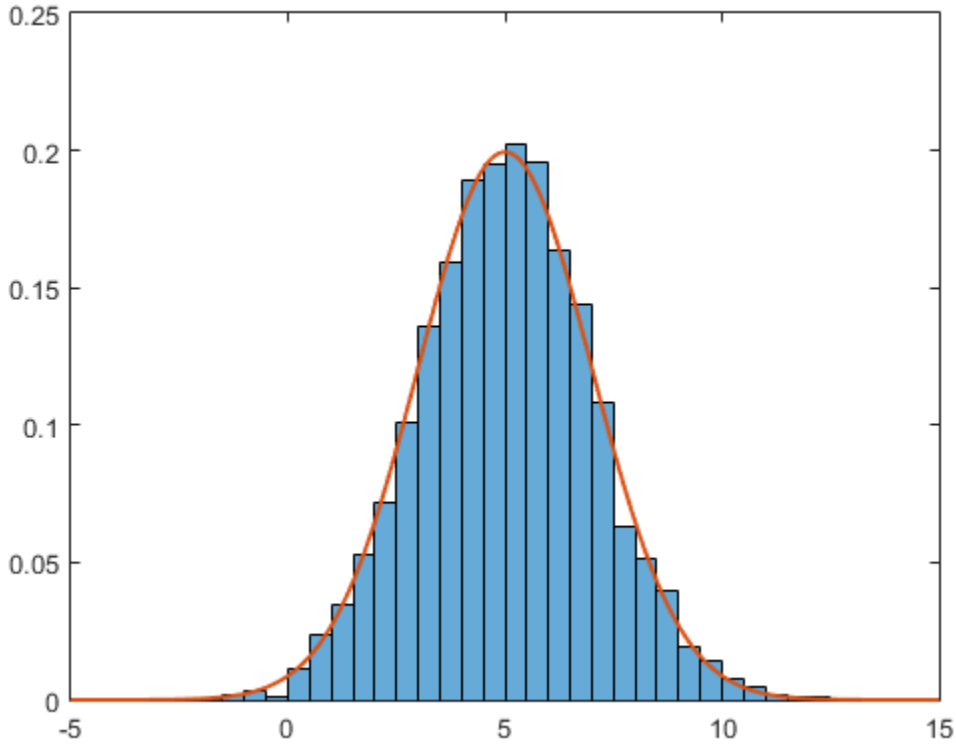
In this example, the underlying distribution for the normally distributed data is known. You can, however, use the `'pdf'` histogram plot to determine the underlying probability distribution of the data by comparing it against a known probability density function.

The probability density function for a normal distribution with mean μ, standard deviation σ, and variance σ^2 is

$$f(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right].$$

Overlay a plot of the probability density function for a normal distribution with a mean of 5 and a standard deviation of 2.

```
hold on
y = -5:0.1:15;
mu = 5;
sigma = 2;
f = exp(-(y-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
plot(y,f,'LineWidth',1.5)
```



Input Arguments

X — Data to distribute among bins

vector | matrix | multidimensional array

Data to distribute among bins, specified as a vector, matrix, or multidimensional array. If `X` is not a vector, then `histogram` treats it as a single column vector, `X(:)`, and plots a single histogram.

`histogram` ignores all `NaN` values. Similarly, `histogram` ignores `Inf` and `-Inf` values, unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge.

Note: If `X` contains integers of type `int64` or `uint64` that are larger than `flintmax`, then it is recommended that you explicitly specify the histogram bin edges. `histogram` automatically bins the input data using double precision, which lacks integer precision for numbers greater than `flintmax`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nbins` — Number of bins

positive integer

Number of bins, specified as a positive integer. If you do not specify `nbins`, then `histogram` automatically calculates how many bins to use based on the values in `X`.

Example: `histogram(X,15)` creates a histogram with 15 bins.

`edges` — Bin edges

vector

Bin edges, specified as a vector. `edges(1)` is the left edge of the first bin, and `edges(end)` is the right edge of the last bin.

The value `X(i)` is in the `k`th bin if `edges(k)` $\le$ `X(i)` < `edges(k+1)`. The last bin also includes the right bin edge, so that it contains `X(i)` if `edges(end-1)` $\le$ `X(i)` $\le$ `edges(end)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ax` — Axes object

axes object

Axes object. If you do not specify an axes, then the histogram function uses the current axes (`gca`).

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `histogram(X,'BinWidth',5)`

The histogram properties listed here are only a subset. For a complete list, see Histogram Properties.

`'BinLimits'` — Bin limits

two-element vector

Bin limits, specified as a two-element vector, `[bmin,bmax]`. This option plots a histogram using the values in the input array, `X`, that fall between `bmin` and `bmax` inclusive. That is, `X(X>=bmin & X<=bmax)`.

Example: `histogram(X,'BinLimits',[1,10])` plots a histogram using only the values in `X` that are between `1` and `10` inclusive.

`'BinLimitsMode'` — Selection mode for bin limits

`'auto'` (default) | `'manual'`

Selection mode for bin limits, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data.

If you explicitly specify either `BinLimits` or `BinEdges`, then `BinLimitsMode` is automatically set to `'manual'`. In that case, specify `BinLimitsMode` as `'auto'` to rescale the bin limits to the data.

`'BinMethod'` — Binning algorithm

`'auto'` (default) | `'scott'` | `'fd'` | `'integers'` | `'sturges'` | `'sqrt'` | `'manual'`

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range

Value	Description
	and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin width of `3.49*std(X(:))*numel(X)^(-1/3)`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and might be more suitable for data with heavy-tailed distributions. It uses a bin width of `2*IQR(X(:))*numel(X)^(-1/3)`, where `IQR` is the interquartile range of `X`.
`'integers'`	The integer rule is useful with integer data, as it creates a bin for each integer. It uses a bin width of 1 and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 65536 bins (2^{16}). If the data range is greater than 65536, then the integer rule uses wider bins instead.
`'sturges'`	Sturges' rule is popular due to its simplicity. It chooses the number of bins to be `ceil(1 + log2(numel(X)))`.
`'sqrt'`	The Square Root rule is widely used in other software packages. It chooses the number of bins to be `ceil(sqrt(numel(X)))`.

If you set the `BinLimits`, `NumBins`, `BinEdges`, or `BinWidth` property, then the `BinMethod` property is set to `'manual'`.

Example: `histogram(X,'BinMethod','integers')` creates a histogram with the bins centered on integers.

`'BinWidth'` — Width of bins

scalar

Width of bins, specified as a scalar. When you specify `BinWidth`, then `histogram` can use a maximum of 65,536 bins (or 2^{16}). If instead the specified bin width requires more bins, then `histogram` uses a larger bin width corresponding to the maximum number of bins.

Example: `histogram(X,'BinWidth',5)` uses bins with a width of 5.

`'DisplayStyle'` — Histogram display style

`'bar'` (default) | `'stairs'`

Histogram display style, specified as either `'bar'` or `'stairs'`. Specify `'stairs'` to display a stairstep plot, which displays the outline of the histogram without filling the interior.

The default value of `'bar'` displays a histogram bar plot.

Example: `histogram(X,'DisplayStyle','stairs')` plots the outline of the histogram.

`'EdgeColor'` — Histogram edge color

`[0 0 0]` or black (default) | `'none'` | `'auto'` | RGB triplet or color string

Histogram edge color, specified as one of these values:

- `'none'` — Edges are not drawn.
- `'auto'` — Color of each edge is chosen automatically.
- RGB triplet or a color string — Edges use the specified color.

  An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`

Long Name	Short Name	RGB Triplet
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `histogram(X,'EdgeColor','r')` creates a histogram plot with red bar edges.

'FaceAlpha' — Transparency of histogram bars

`0.6` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bars, specified as a scalar value between `0` and `1` inclusive. `histogram` uses the same transparency for all the bars of the histogram. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram(X,'FaceAlpha',1)` creates a histogram plot with fully opaque bars.

'FaceColor' — Histogram bar color

`'auto'` (default) | `'none'` | RGB triplet or color string

Histogram bar color, specified as one of these values:

- `'none'` — Bars are not filled.
- `'auto'` — Histogram bar color is chosen automatically (default).
- RGB triplet or a color string — Bars are filled with the specified color.

  An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]

Long Name	Short Name	RGB Triplet
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

If you specify `DisplayStyle` as `'stairs'`, then `histogram` does not use the `FaceColor` property.

Example: `histogram(X,'FaceColor','g')` creates a histogram plot with green bars.

`'Normalization'` — Type of normalization

`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table.

Value	Description
`'count'`	Default normalization scheme. The height of each bar is the number of observations in each bin. The sum of the bar heights is `numel(X)`.
`'probability'`	The height of each bar is the relative number of observations, (number of observations in bin / total number of observations). The sum of the bar heights is 1.
`'countdensity'`	The height of each bar is, (number of observations in bin / width of bin). The area (height * width) of each bar is the number of observations in the bin. The sum of the bar areas is `numel(X)`.
`'pdf'`	Probability density function estimate. The height of each bar is, (number of observations in the bin) / (total number of observations * width of bin). The area of each bar is the relative number of observations. The sum of the bar areas is 1.

Value	Description
`'cumcount'`	The height of each bar is the cumulative number of observations in each bin and all previous bins. The height of the last bar is `numel(X)`.
`'cdf'`	Cumulative density function estimate. The height of each bar is equal to the cumulative relative number of observations in the bin and all previous bins. The height of the last bar is `1`.

Example: `histogram(X,'Normalization','pdf')` plots an estimate of the probability density function for `X`.

`'Orientation'` — Orientation of bars
`'vertical'` (default) | `'horizontal'`

Orientation of bars, specified as `'vertical'` or `'horizontal'`.

Example: `histogram(X,'Orientation','horizontal')` creates a histogram plot with horizontal bars.

Output Arguments

h — Histogram
object

Histogram, returned as an object. For more information, see Using histogram Objects.

More About

- Using histogram Objects
- "Replace Discouraged Instances of hist and histc"

See Also

`fewerbins` | `histcounts` | `morebins`

Introduced in R2014b

Using histogram Objects

Histogram bar plot for numeric data

Histograms are a type of bar plot for numeric data that group the data into bins. After you create a histogram object, you can modify aspects of the histogram by changing its property values. This is particularly useful for quickly modifying the properties of the bins or changing the display.

Examples

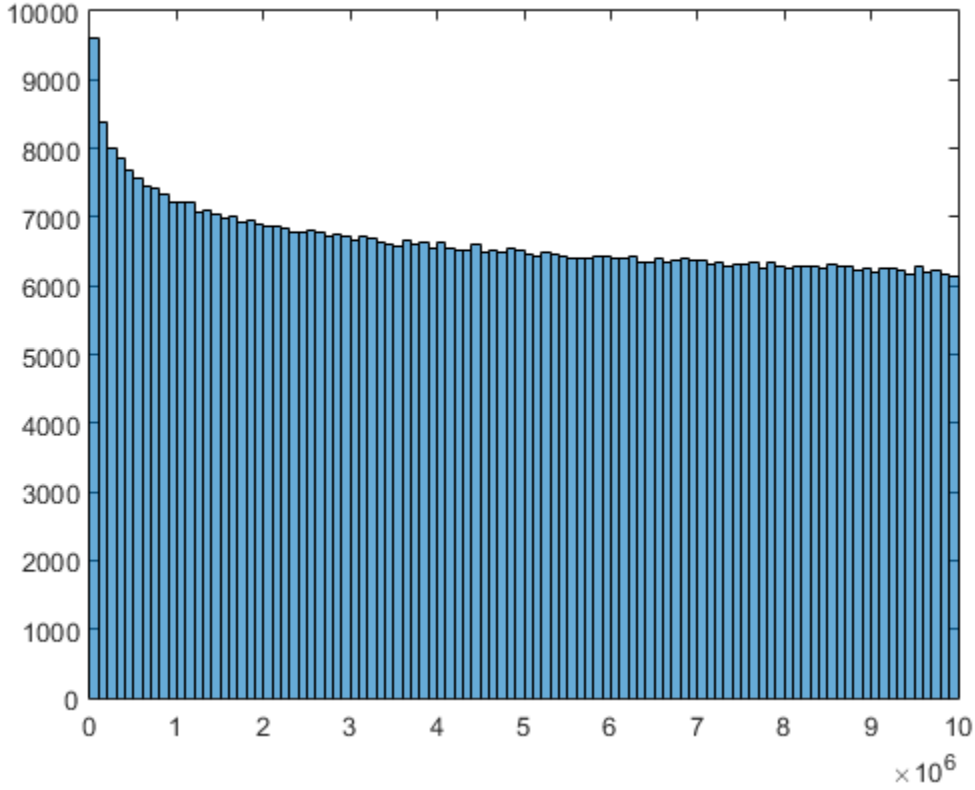
Adjust Properties of Histogram Object

This example creates a histogram object, and then shows how to adjust the properties of the histogram to affect the output display.

Create a histogram for all of the prime numbers less than 10,000,000. Specify an output argument to return the histogram object, `h`.

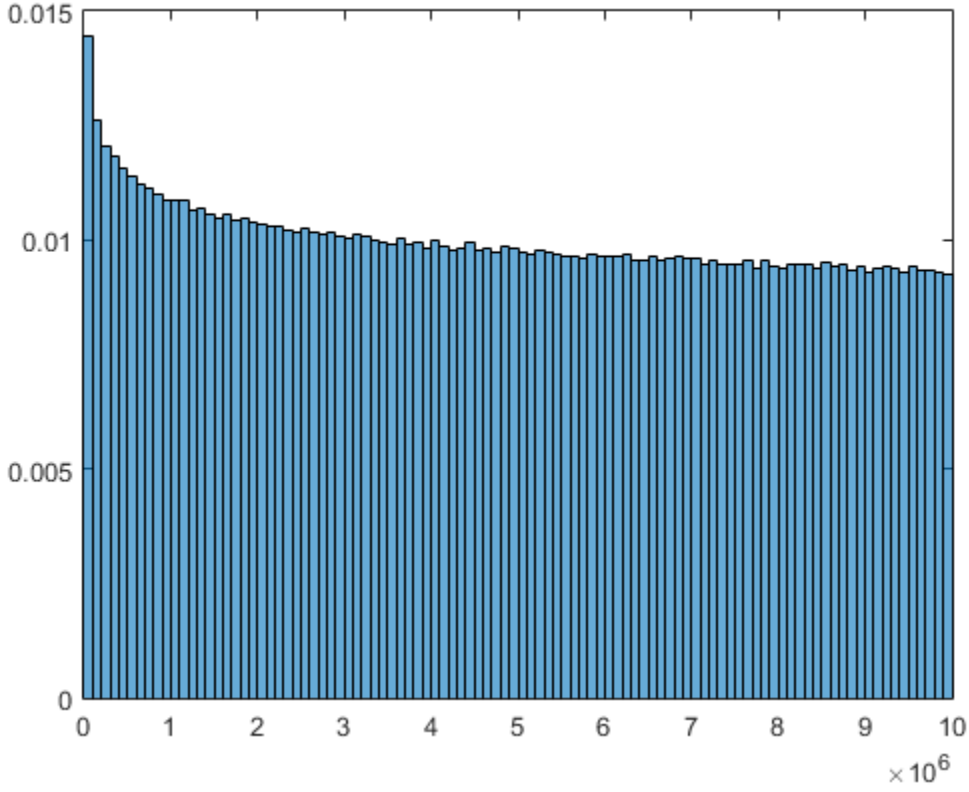
```
x = primes(1e7);
h = histogram(x)
```

```
h =

  Histogram with properties:

            Data: [1x664579 double]
          Values: [1x100 double]
         NumBins: 100
        BinEdges: [1x101 double]
        BinWidth: 100000
       BinLimits: [0 10000000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]

  Use GET to show all properties
```



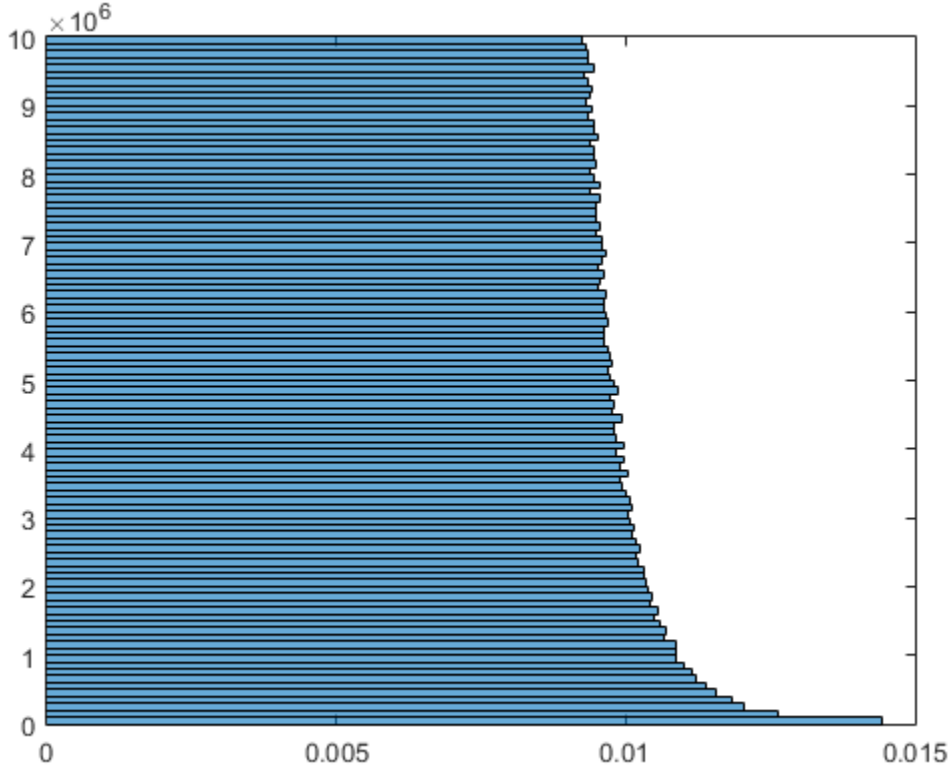
Change the normalization of the histogram by setting the `Normalization` property to `'probability'`.

```
h.Normalization = 'probability';
```



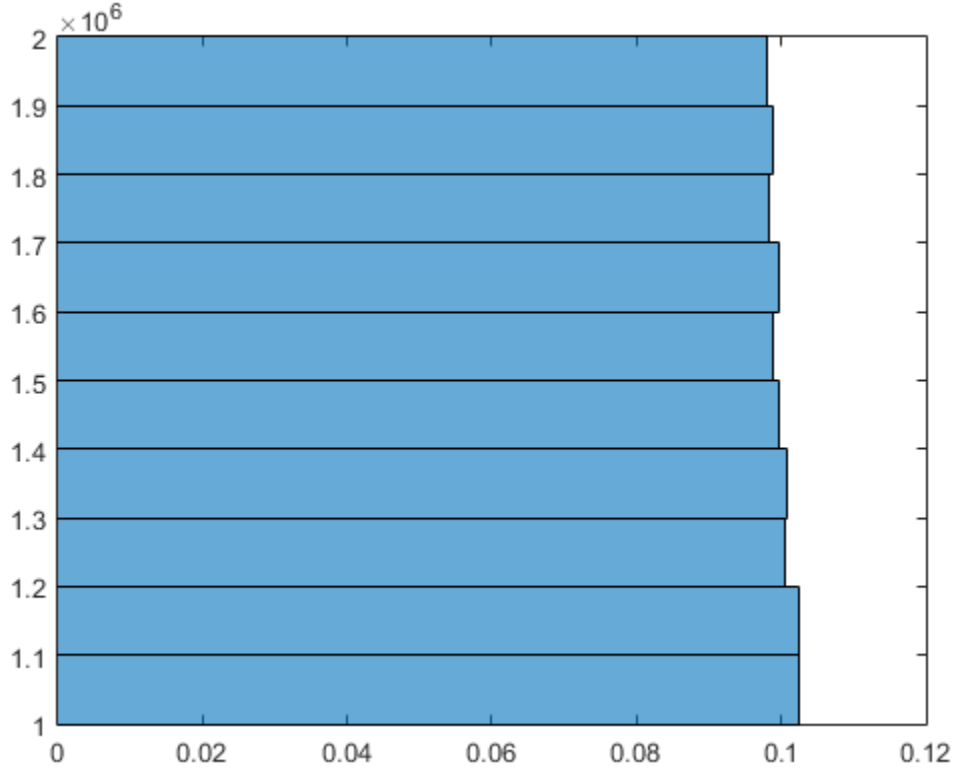
Use horizontal bars for the histogram.

```
h.Orientation = 'horizontal';
```



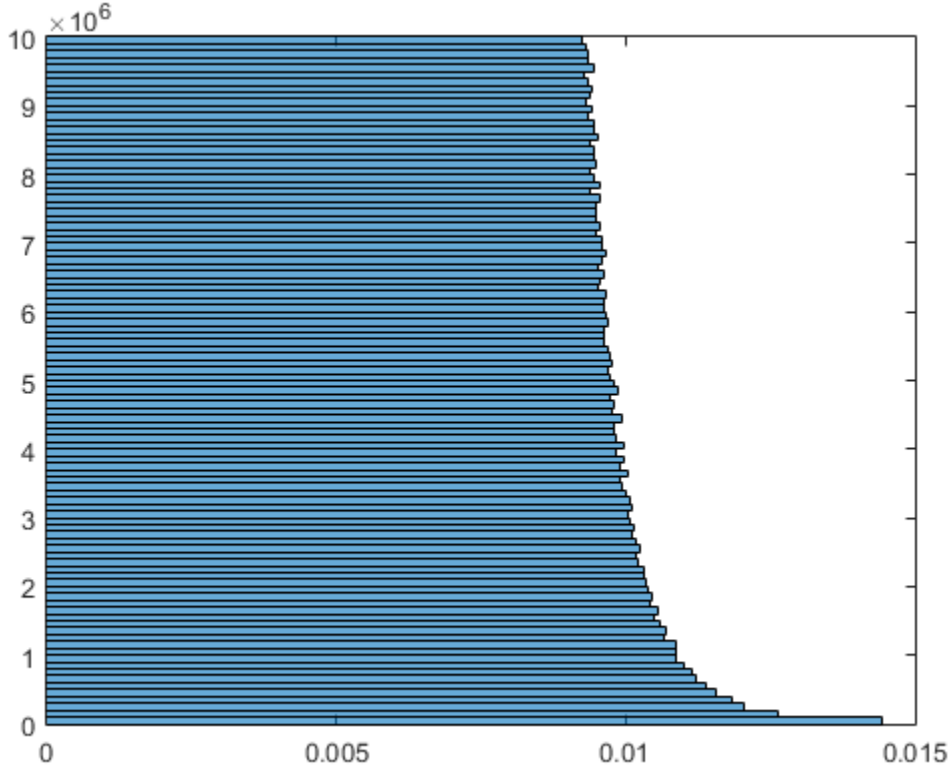
Zoom in to a specific range of bins. When you set `BinLimits`, the `BinLimitsMode` automatically changes to `'manual'`.

```
h.BinLimits = [1e6,2e6];
```



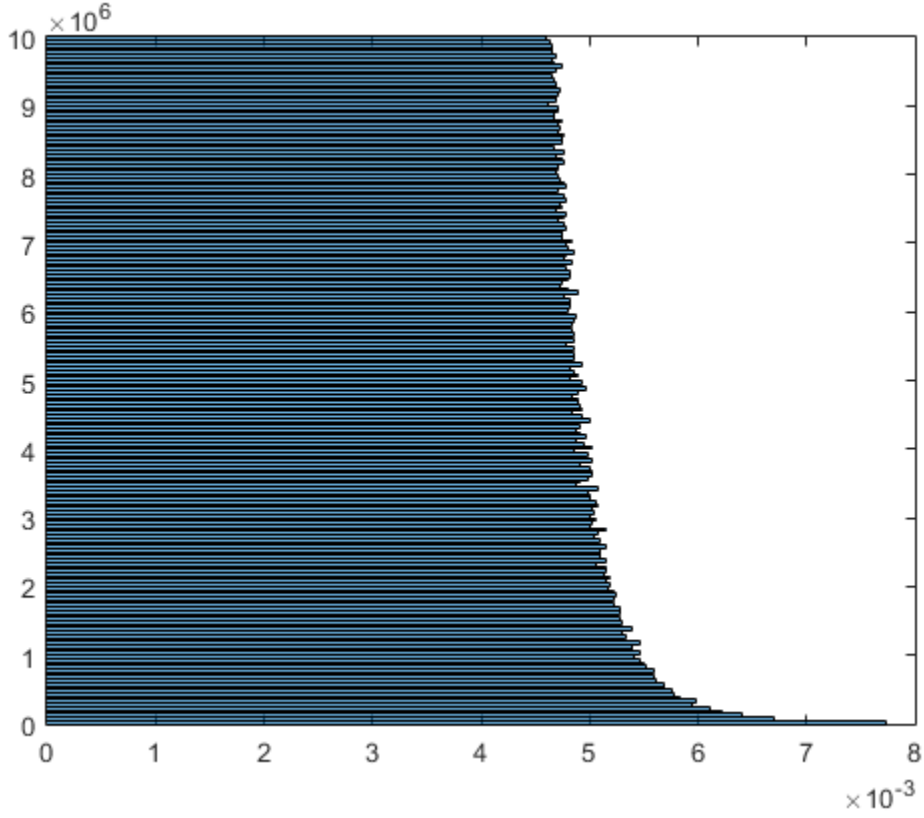
Zoom back out to the original bin scaling by switching `BinLimitsMode` back to `'auto'`.

```
h.BinLimitsMode = 'auto';
```



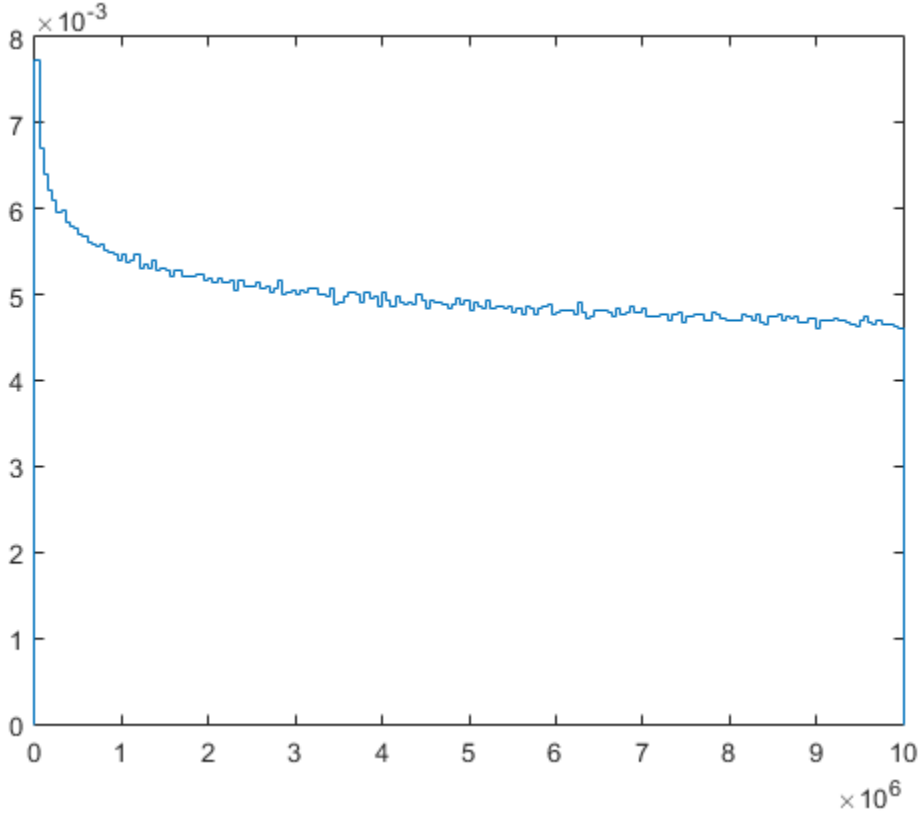
Change the number of bins in the histogram.

```
h.NumBins = 200;
```



Create a vertical stair histogram.

```
h.Orientation = 'vertical';
h.DisplayStyle = 'stairs';
```



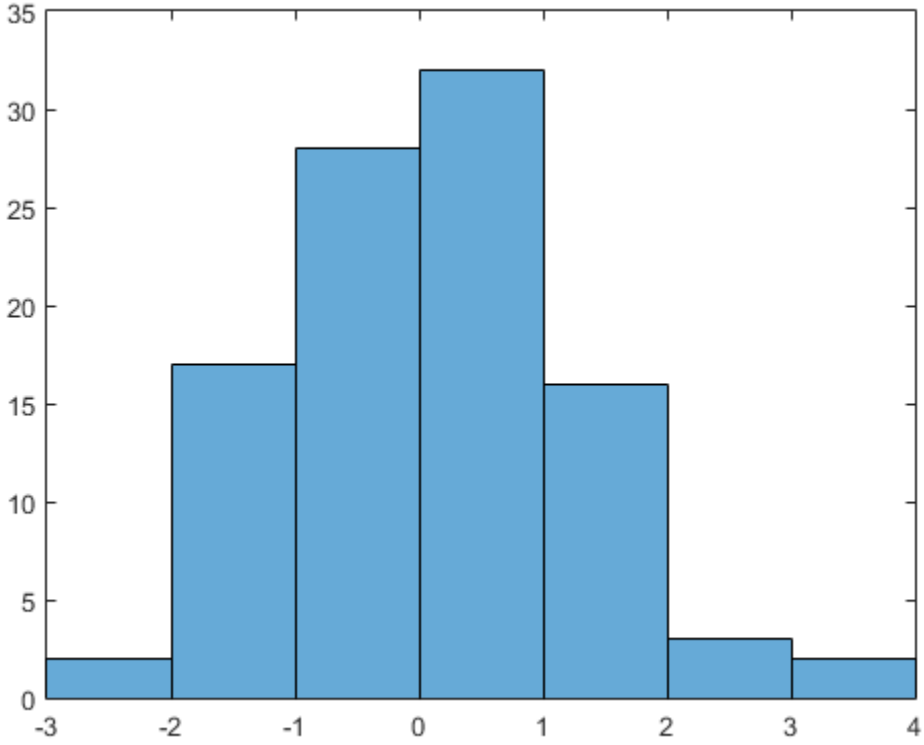
Saving and Loading Histogram Objects

Use the `savefig` function to save a histogram figure.

```
y = histogram(randn(10));
savefig('histogram.fig');
clear all
close all
```

Use `openfig` to load the histogram figure back into MATLAB. `openfig` also returns a handle to the figure, `h`.

```
h = openfig('histogram.fig');
```



Use the `findobj` function to locate the correct object handle from the figure handle. This allows you to continue manipulating the original histogram object used to generate the figure.

```
y = findobj(h, 'type', 'histogram')
```

```
y =

  Histogram with properties:

            Data: [10x10 double]
          Values: [2 17 28 32 16 3 2]
         NumBins: 7
```

```
         BinEdges: [-3 -2 -1 0 1 2 3 4]
         BinWidth: 1
        BinLimits: [-3 4]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]

  Use GET to show all properties
```

Properties

Histogram Properties

Object Functions

`morebins` `fewerbins`

Create Object

Specify an output argument with the `histogram` function to create a histogram object.

See Also

`histcounts` | `histogram`

More About

- Histogram Properties

Histogram Properties

Control histogram appearance and behavior

Histogram properties control the appearance and behavior of the histogram. By changing property values, you can modify aspects of the histogram. Use dot notation to refer to a particular object and property:

```
h = histogram(randn(10,1));
c = h.BinWidth;
h.BinWidth = 2;
```

Bins

`BinEdges` — Edges of bins

numeric vector

Edges of bins, specified as a numeric vector. The first vector element specifies the left edge of the first bin. The last element specifies the right edge of the last bin. If you do not specify the bin edges, then `histogram` automatically determines the location of the bin edges.

`BinLimits` — Bin limits

two-element vector

Bin limits, specified as a two-element vector, `[bmin,bmax]`. This option plots a histogram using the values in the input array, `X`, that fall between `bmin` and `bmax` inclusive. That is, `X(X>=bmin & X<=bmax)`.

Example: `histogram(X,'BinLimits',[1,10])` plots a histogram using only the values in `X` that are between `1` and `10` inclusive.

`BinLimitsMode` — Selection mode for bin limits

`'auto'` (default) | `'manual'`

Selection mode for bin limits, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data.

If you explicitly specify either `BinLimits` or `BinEdges`, then `BinLimitsMode` is automatically set to `'manual'`. In that case, specify `BinLimitsMode` as `'auto'` to rescale the bin limits to the data.

`BinMethod` — Binning algorithm

`'auto'` (default) | `'scott'` | `'fd'` | `'integers'` | `'sturges'` | `'sqrt'` | `'manual'`

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin width of `3.49*std(X(:))*numel(X)^(-1/3)`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and might be more suitable for data with heavy-tailed distributions. It uses a bin width of `2*IQR(X(:))*numel(X)^(-1/3)`, where `IQR` is the interquartile range of `X`.
`'integers'`	The integer rule is useful with integer data, as it creates a bin for each integer. It uses a bin width of 1 and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 65536 bins (2^{16}). If the data range is greater than 65536, then the integer rule uses wider bins instead.
`'sturges'`	Sturges' rule is popular due to its simplicity. It chooses the number of bins to be `ceil(1 + log2(numel(X)))`.
`'sqrt'`	The Square Root rule is widely used in other software packages. It chooses the number of bins to be `ceil(sqrt(numel(X)))`.

If you set the `BinLimits`, `NumBins`, `BinEdges`, or `BinWidth` property, then the `BinMethod` property is set to `'manual'`.

Example: `histogram(X,'BinMethod','integers')` creates a histogram with the bins centered on integers.

`BinWidth` — Width of bins

scalar

Width of bins, specified as a scalar. When you specify `BinWidth`, then `histogram` can use a maximum of 65,536 bins (or 2^{16}). If instead the specified bin width requires more bins, then `histogram` uses a larger bin width corresponding to the maximum number of bins.

Example: `histogram(X,'BinWidth',5)` uses bins with a width of 5.

`Normalization` — Type of normalization

`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table.

Value	Description
`'count'`	Default normalization scheme. The height of each bar is the number of observations in each bin. The sum of the bar heights is `numel(X)`.
`'probability'`	The height of each bar is the relative number of observations, (number of observations in bin / total number of observations). The sum of the bar heights is 1.
`'countdensity'`	The height of each bar is, (number of observations in bin / width of bin). The area (height * width) of each bar is the number of observations in the bin. The sum of the bar areas is `numel(X)`.
`'pdf'`	Probability density function estimate. The height of each bar is, (number of observations in the bin) / (total number

Value	Description
	of observations * width of bin). The area of each bar is the relative number of observations. The sum of the bar areas is 1.
`'cumcount'`	The height of each bar is the cumulative number of observations in each bin and all previous bins. The height of the last bar is `numel(X)`.
`'cdf'`	Cumulative density function estimate. The height of each bar is equal to the cumulative relative number of observations in the bin and all previous bins. The height of the last bar is `1`.

Example: `histogram(X,'Normalization','pdf')` plots an estimate of the probability density function for `X`.

`NumBins` — Number of bins

positive integer

Number of bins, specified as a positive integer. If you do not specify `NumBins`, then `histogram` automatically calculates how many bins to use based on the values in `Data`.

`EdgeColor` — Histogram edge color

`[0 0 0]` or black (default) | `'none'` | `'auto'` | RGB triplet or color string

Histogram edge color, specified as one of these values:

- `'none'` — Edges are not drawn.
- `'auto'` — The color of each edge is chosen automatically.
- RGB triplet or a color string — Edges use the specified color.

  An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`

Long Name	Short Name	RGB Triplet
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `histogram(X,'EdgeColor','r')` creates a histogram plot with red bar edges.

`FaceColor` — Histogram bar color

`'auto'` (default) | `'none'` | RGB triplet or color string

Histogram bar color, specified as one of these values:

- `'none'` — Bars are not filled.
- `'auto'` — The histogram bar color is chosen automatically (default).
- RGB triplet or a color string — Bars are filled with the specified color.

  An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]

Long Name	Short Name	RGB Triplet
'black	'k'	[0 0 0]

If you specify DisplayStyle as 'stairs', then histogram does not utilize the FaceColor property.

Example: histogram(X,'FaceColor','g') creates a histogram plot with green bars.

FaceAlpha — Transparency of histogram bars

0.6 (default) | scalar value between 0 and 1 inclusive

Transparency of histogram bars, specified as a scalar value between 0 and 1 inclusive. histogram uses the same transparency for all the bars of the histogram. A value of 1 means fully opaque and 0 means completely transparent (invisible).

Example: histogram(X,'FaceAlpha',1) creates a histogram plot with fully opaque bars.

Data

Data — Data to distribute among bins

vector | matrix | multidimensional array

Data to distribute among bins, specified as a vector, matrix, or multidimensional array. If Data is not a vector, then histogram treats it as a single column vector, Data(:), and plots a single histogram.

histogram ignores all NaN values. Similarly, histogram ignores Inf and -Inf values unless the bin edges explicitly specify Inf or -Inf as a bin edge.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Values — Bin values

numeric vector

Bin values, specified as a numeric vector. If Normalization is 'count' (the default), then the kth element in Values specifies how many elements of Data fall in the kth bin interval (bin counts). The last bin includes values that are on *either* bin edge, but all other bins only include values that fall on the left edge.

Depending on the value of `Normalization`, the `Values` property can instead contain a normalized variant of the bin counts.

Histogram Type

`DisplayStyle` — Histogram display style

`'bar'` (default) | `'stairs'`

Histogram display style, specified as either `'bar'` or `'stairs'`. Specify `'stairs'` to display a stairstep plot, which displays the outline of the histogram without filling the interior.

The default value of `'bar'` displays a histogram bar plot.

Example: `histogram(X,'DisplayStyle','stairs')` plots the outline of the histogram.

`Orientation` — Orientation of bars

`'vertical'` (default) | `'horizontal'`

Orientation of bars, specified as `'vertical'` or `'horizontal'`.

Example: `histogram(X,'Orientation','horizontal')` creates a histogram plot with horizontal bars.

Visibility

`Visible` — Visibility of histogram

`'on'` (default) | `'off'`

Visibility of histogram, specified as one of these values:

- `'on'` — Display the histogram.
- `'off'` — Hide the histogram without deleting it. You still can access the properties of an invisible histogram object.

Identifiers

`Type` — Type of graphics object

`'histogram'` (default)

Type of graphics object, returned as `'histogram'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

`Tag` — Tag to associate with histogram

`''` (default) | string

Tag to associate with the histogram, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

Data Types: `char`

`UserData` — Data to associate with histogram

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the histogram object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the histogram.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property

does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the histogram object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the histogram from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the histogram object in the legend as one entry (default).
   - `'off'` — Do not include the histogram object in the legend.
   - `'children'` — Include only children of the histogram object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of histogram

axes object | group object | transform object

Parent of histogram, specified as an axes, group, or transform object.

`Children` — Children of histogram

empty `GraphicsPlaceholder` array

The histogram has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of histogram object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The histogram object handle is always visible.
- `'off'` — The histogram object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The histogram object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the histogram at the command-line, but allows callback functions to access it.

If the histogram object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the histogram. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The histogram object — You can access properties of the histogram object from within the callback function.

- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the histogram. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the histogram when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the histogram.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.

- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The Visible property must be set to `'on'`. The HitTest property determines if the histogram responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the histogram passes the click to the object below it in the current view of the figure window. The HitTest property of the histogram has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the histogram. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the histogram that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the histogram object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the histogram is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the

`BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the histogram tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the histogram. Setting the `CreateFcn` property on an existing histogram has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during histogram creation. MATLAB executes the callback after creating the histogram and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The histogram object — You can access properties of the histogram object from within the callback function. You also can access the histogram object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the histogram. MATLAB executes the callback before destroying the histogram so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The histogram object — You can access properties of the histogram object from within the callback function. You also can access the histogram object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of histogram

`'off'` (default) | `'on'`

Deletion status of histogram, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the histogram begins

execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the histogram no longer exists.

Check the value of the BeingDeleted property to verify that the histogram is not about to be deleted before querying or modifying it.

See Also

`histogram`

More About

- “Access Property Values”
- Using histogram Objects

histcounts

Histogram bin counts

Syntax

```
[N,edges] = histcounts(X)
[N,edges] = histcounts(X,nbins)
[N,edges] = histcounts(X,edges)
[N,edges] = histcounts(___,Name,Value)
[N,edges,bin] = histcounts(___)
```

Description

`[N,edges] = histcounts(X)` partitions the `X` values into bins, and returns the count in each bin, as well as the bin edges. The `histcounts` function uses an automatic binning algorithm that returns bins with a uniform width, chosen to cover the range of elements in `X` and reveal the underlying shape of the distribution.

`[N,edges] = histcounts(X,nbins)` uses a number of bins specified by the scalar, `nbins`.

`[N,edges] = histcounts(X,edges)` sorts `X` into bins with the bin edges specified by the vector, `edges`. The value `X(i)` is in the `k`th bin if `edges(k)` $\le$ `X(i)` $<$ `edges(k+1)`. The last bin also includes the right bin edge, so that it contains `X(i)` if `edges(end-1)` $\le$ `X(i)` $\le$ `edges(end)`.

`[N,edges] = histcounts(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, you can specify `'BinWidth'` and a scalar to adjust the width of the bins.

`[N,edges,bin] = histcounts(___)` also returns an index array, `bin`, using any of the previous syntaxes. `bin` is an array of the same size as `X` whose elements are the bin indices for the corresponding elements in `X`. The number of elements in the `k`th bin is `nnz(bin==k)`, which is the same as `N(k)`.

Examples

Bin Counts and Bin Edges

Distribute 100 random values into bins. `histcounts` automatically chooses an appropriate bin width to reveal the underlying distribution of the data.

```
X = randn(100,1);
[N,edges] = histcounts(X)
```

```
N =

     2    17    28    32    16     3     2


edges =

    -3    -2    -1     0     1     2     3     4
```

Specify Number of Bins

Distribute 10 numbers into 6 equally spaced bins.

```
X = [2 3 5 7 11 13 17 19 23 29];
[N,edges] = histcounts(X,6)
```

```
N =

     2     2     2     2     1     1


edges =

         0    4.9000    9.8000   14.7000   19.6000   24.5000   29.4000
```

Specify Bin Edges

Distribute 1,000 random numbers into bins. Define the bin edges with a vector, where the first element is the left edge of the first bin, and the last element is the right edge of the last bin.

```
X = randn(1000,1);
```

```
edges = [-5 -4 -2 -1 -0.5 0 0.5 1 2 4 5];
N = histcounts(X,edges)
```

```
N =

     0    24   149   142   195   200   154   111    25     0
```

Normalized Bin Counts

Distribute all of the prime numbers less than 100 into bins. Specify `'Normalization'` as `'probability'` to normalize the bin counts so that `sum(N)` is `1`. That is, each bin count represents the probability that an observation falls within that bin.

```
X = primes(100);
[N,edges] = histcounts(X, 'Normalization', 'probability')
```

```
N =

    0.4000    0.2800    0.2800    0.0400


edges =

     0    30    60    90   120
```

Determine Bin Placement

Distribute 100 random integers between -5 and 5 into bins, and specify `'BinMethod'` as `'integers'` to use unit-width bins centered on integers. Specify a third output for `histcounts` to return a vector representing the bin indices of the data.

```
X = randi([-5,5],100,1);
[N,edges,bin] = histcounts(X,'BinMethod','integers');
```

Find the bin count for the third bin by counting the occurrences of the number `3` in the bin index vector, `bin`. The result is the same as `N(3)`.

```
count = nnz(bin==3)
```

```
count =
```

```
8
```

Input Arguments

`X` — Data to distribute among bins

vector | matrix | multidimensional array

Data to distribute among bins, specified as a vector, matrix, or multidimensional array. If `X` is not a vector, then `histcounts` treats it as a single column vector, `X(:)`.

`histcounts` ignores all `NaN` values. Similarly, `histcounts` ignores `Inf` and `-Inf` values unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nbins` — Number of bins

positive integer

Number of bins, specified as a positive integer. If you do not specify `nbins`, then `histcounts` automatically calculates how many bins to use based on the values in `X`.

Example: `[N,edges] = histcounts(X,15)` uses 15 bins.

`edges` — Bin edges

vector

Bin edges, specified as a vector. `edges(1)` is the left edge of the first bin, and `edges(end)` is the right edge of the last bin.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[N,edges] = histcounts(X,'Normalization','probability')` normalizes the bin counts in `N`, such that `sum(N)` is 1.

`'BinLimits'` — Bin limits

two-element vector

Bin limits, specified as a two-element vector, `[bmin,bmax]`. This option bins only the values in `X` that fall between `bmin` and `bmax` inclusive; that is, `X(X>=bmin & X<=bmax)`.

Example: `[N,edges] = histcounts(X,'BinLimits',[1,10])` bins only the values in `X` that are between `1` and `10` inclusive.

`'BinMethod'` — Binning algorithm

`'auto'` (default) | `'scott'` | `'fd'` | `'integers'` | `'sturges'` | `'sqrt'`

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being normally distributed, but is also appropriate for most other distributions. It uses a bin width of `3.49*std(X(:))*numel(X)^(-1/3)`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and may be more suitable for data with heavy-tailed distributions. It uses a bin width of `2*IQR(X(:))*numel(X)^(-1/3)`, where `IQR` is the interquartile range of `X`.
`'integers'`	The integer rule is useful with integer data, as it creates a bin for each integer. It uses a bin width of 1 and places bin edges halfway between integers. To prevent from accidentally creating too many bins, a limit of 65536 bins (2^{16}) can be created with this rule. If the data range is greater than 65536, then wider bins are used instead.

Value	Description
`'sturges'`	Sturges' rule is a simple rule that is popular due to its simplicity. It chooses the number of bins to be `ceil(1 + log2(numel(X)))`.
`'sqrt'`	The Square Root rule is another simple rule widely used in other software packages. It chooses the number of bins to be `ceil(sqrt(numel(X)))`.

Example: `[N,edges] = histcounts(X,'BinMethod','integers')` uses bins centered on integers.

`'BinWidth'` — Width of bins

scalar

Width of bins, specified as a scalar. When you specify `BinWidth`, `histcounts` can use a maximum of 65,536 bins (or 2^{16}). If the specified bin width requires more bins, then `histcounts` uses a larger bin width corresponding to the maximum number of bins.

Example: `[N,edges] = histcounts(X,'BinWidth',5)` uses bins with a width of 5.

`'Normalization'` — Type of normalization

`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table.

Value	Description
`'count'`	Default normalization scheme. Each `N` value is equal to the number of observations in the bin, and `sum(N)` is equal to `numel(X)`.
`'probability'`	Each `N` value is equal to the relative number of observations in the bin such that `sum(N)` is `1`.
`'countdensity'`	Each `N` value is equal to the number of observations in the bin divided by the bin width.

Value	Description
`'pdf'`	Probability density function estimate. Each `N` value is equal to the relative number of observations in the bin divided by the bin width.
`'cumcount'`	Each `N` value is equal to the cumulative number of observations in the bin and all previous bins. `N(end)` is equal to `numel(X)`.
`'cdf'`	Cumulative density function estimate. Each `N` value is equal to the cumulative relative number of observations in the bin and all previous bins. `N(end)` is equal to `1`.

Example: `[N,edges] = histcounts(X,'Normalization','pdf')` bins the data using the probability density function estimate.

Output Arguments

`N` — Bin counts

row vector

Bin counts, returned as a row vector.

`edges` — Bin edges

vector

Bin edges, returned as a vector. `edges(1)` is the left edge of the first bin, and `edges(end)` is the right edge of the last bin.

`bin` — Bin indices

array

Bin indices, returned as an array of the same size as `X`. Each element in `bin` describes which numbered bin contains the corresponding element in `X`.

A value of `0` in `bin` indicates an element which does not belong to any of the bins (for example, a `NaN` value).

More About

Tips

- The behavior of `histcounts` is similar to that of the `discretize` function. Use `histcounts` to find the number of elements in each bin. On the other hand, use `discretize` to find which bin each element belongs to (without counting).
- "Replace Discouraged Instances of hist and histc"

See Also

`discretize` | `histogram`

Introduced in R2014b

hms

Hour, minute, and second numbers of duration

Syntax

```
[h,m,s] = hms(t)
```

Description

`[h,m,s] = hms(t)` returns the hour, minute, and second values of the datetime values in `t` as separate numeric arrays. The `h` and `m` outputs contain integer values, and the `s` output can contain a fractional part. `h`, `m`, and `s` are the same size as `t`.

Calling `hms` is equivalent to calling the `hour`, `minute`, and `second` functions.

Examples

Find Hour, Minute, and Second Numbers of datetime Array

```
t1 = datetime('now','Format','HH:mm:ss.SSS');
t = t1 + minutes(0:45:135)

[h,m,s] = hms(t)
```

```
t =

   09:58:49.720   10:43:49.720   11:28:49.720   12:13:49.720


h =

     9    10    11    12


m =
```

```
    58    43    28    13

s =

   49.7200   49.7200   49.7200   49.7200
```

`hms` returns the hour, minute, and second numbers in separate arrays.

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

Output Arguments

`h` — Hour numbers

numeric array

Hour numbers, returned as a numeric array of integer values from 0 to 23. The `h` output is of type `double` and is the same size as `t`.

`m` — Minute numbers

numeric array

Minute numbers, returned as a numeric array of integer values from 0 to 59. The `m` output is of type `double` and is the same size as `t`.

`s` — Second numbers

numeric array

Second numbers, returned as a numeric array of values from 0 to less than 60, and can include a fractional part. For datetime values whose time zone is `UTCLeapSeconds`, the `s` output can contain a value between 60 and 61 for times that fall during a leap second occurrence. The `s` output is of type `double` and is the same size as `t`.

See Also

`hour` | `minute` | `second` | `ymd`

Introduced in R2014b

hold

Retain current plot when adding new plots

Syntax

```
hold on
hold off
hold all
hold

hold(ax,'on')
hold(ax,'off')
hold(ax)
```

Description

`hold on` retains plots in the current axes so that new plots added to the axes do not delete existing plots. New plots use the next colors and line styles based on the `ColorOrder` and `LineStyleOrder` properties of the axes. MATLAB adjusts axes limits, tick marks, and tick labels to display the full range of data.

`hold off` sets the hold state to off so that new plots added to the axes clear existing plots and reset all axes properties. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyleOrder` properties of the axes. This is the default behavior.

`hold all` is the same as `hold on`.

Note: This syntax will be removed in a future release. Use `hold on` instead.

`hold` toggles the hold state between on and off.

`hold(ax,'on')` sets the hold state to on for the axes specified by `ax` instead of the current axes.

`hold(ax,'off')` sets the hold state to off for the specified axes.

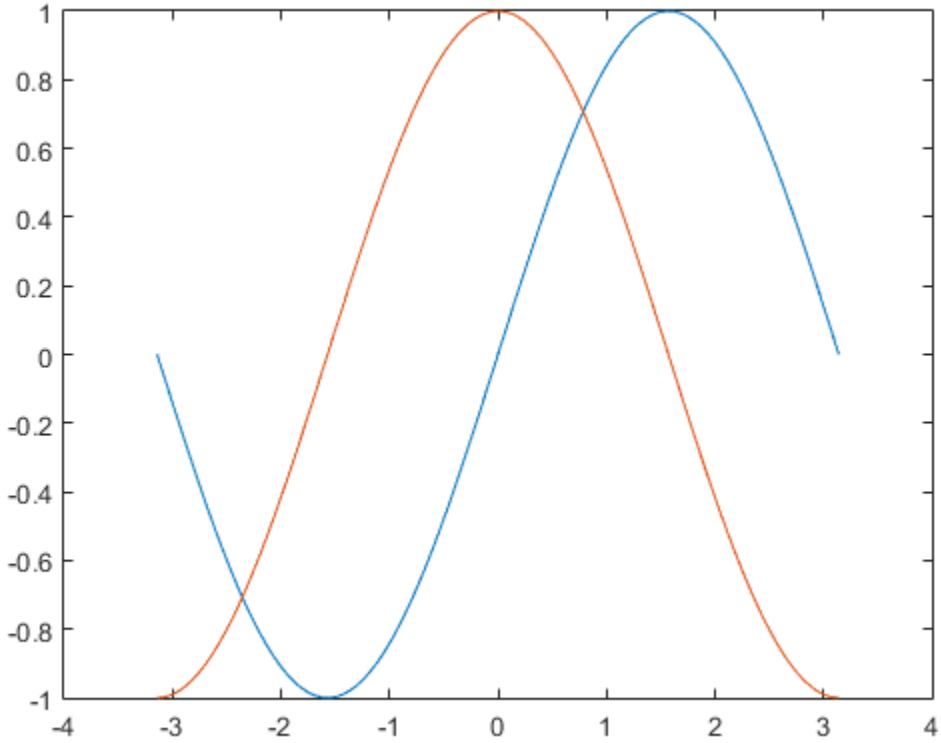
`hold(ax)` toggles the hold state for the specified axes.

Examples

Add Line Plot to Existing Graph

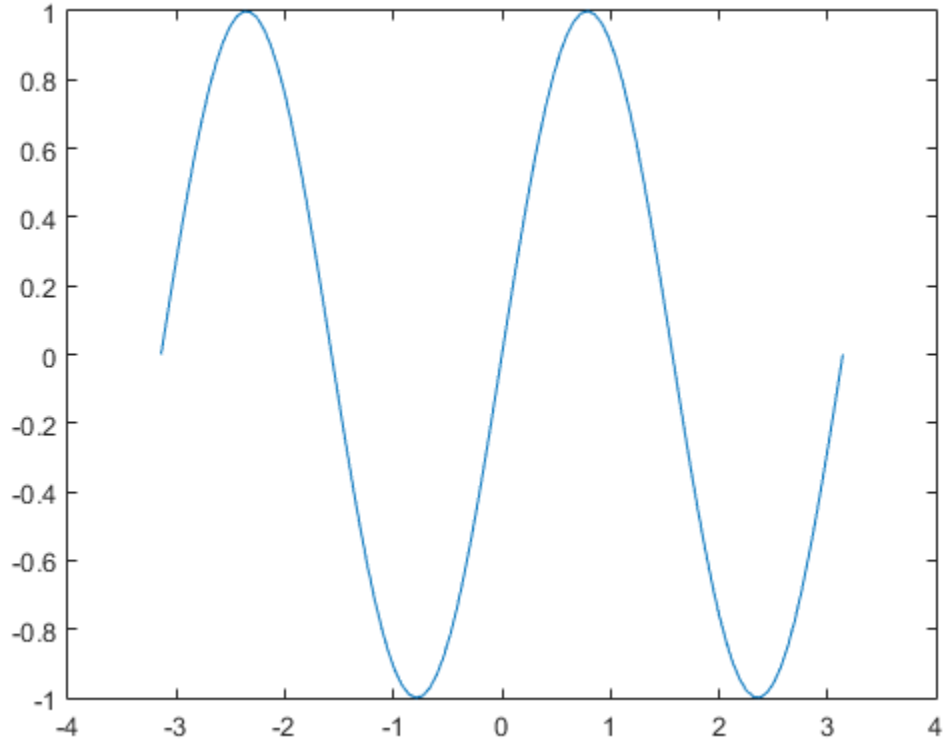
Create a line plot. Then, use `hold on` to add a second line plot without deleting the existing plot. The new plot uses the next color and line style based on the `ColorOrder` and `LineStyleOrder` properties of the axes.

```
x = linspace(-pi,pi);
y1 = sin(x);
y2 = cos(x);

plot(x,y1)
hold on
plot(x,y2)
```



Reset the hold state to off so that new plots delete existing plots. New plots start from the beginning of the color order and line style order.

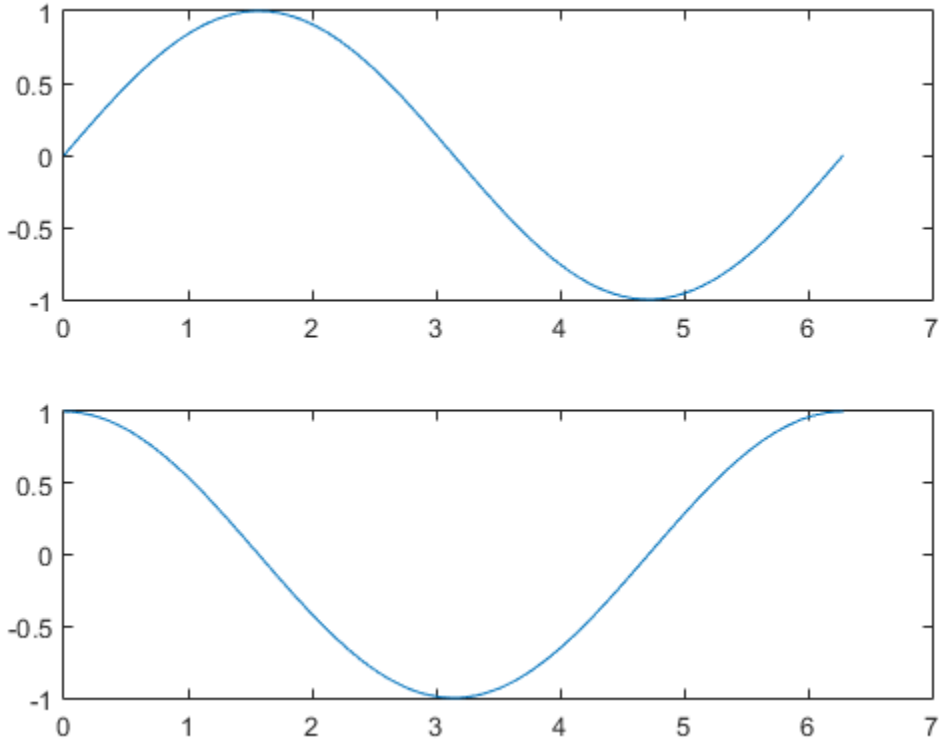
```
hold off
y3 = sin(2*x);
plot(x,y3)
```



Specify Hold State for Specific Axes

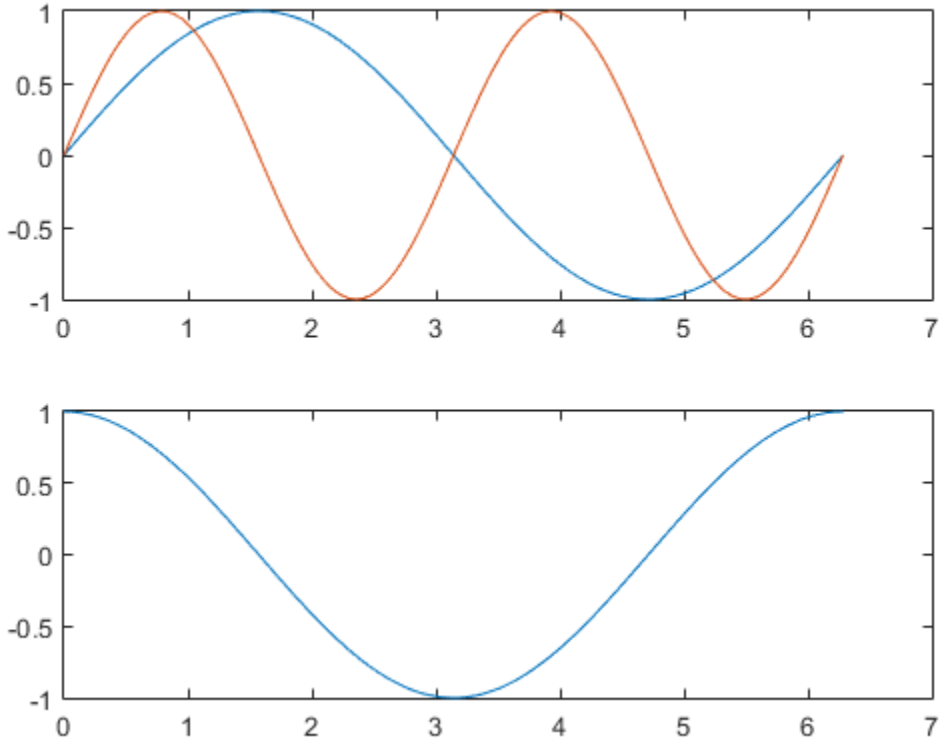
Create a figure with two subplots and add a line plot to each subplot.

```
ax1 = subplot(2,1,1);
x = linspace(0,2*pi);
y1 = sin(x);
plot(x,y1)

ax2 = subplot(2,1,2);
y2 = cos(x);
plot(x,y2)
```



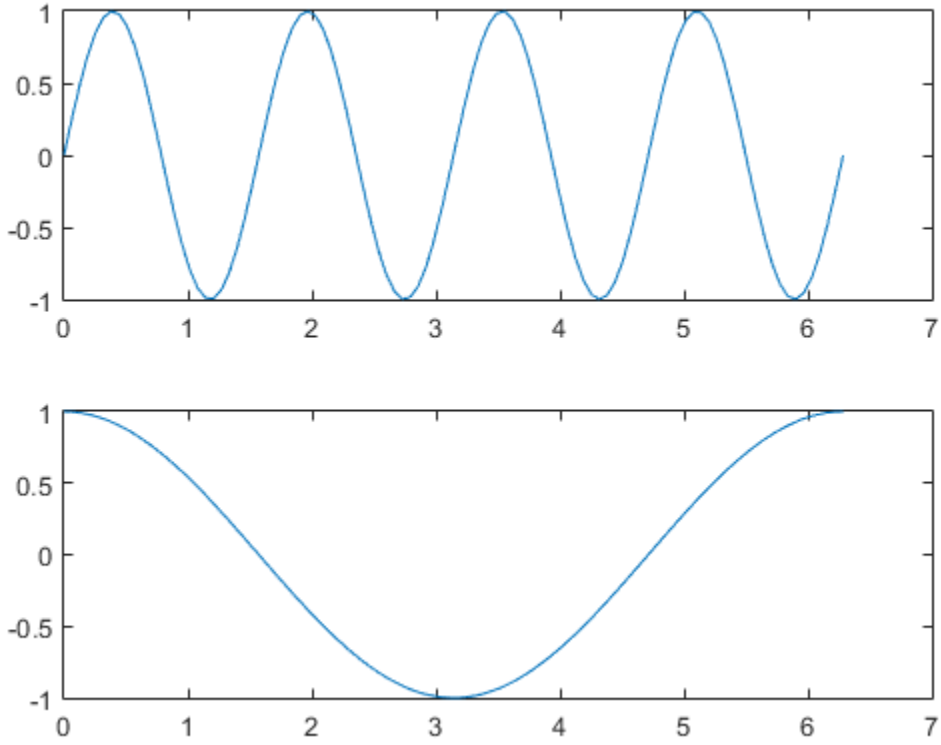
Add a second line plot to the upper subplot.

```
hold(ax1,'on')
y3 = sin(2*x);
plot(ax1,x,y3)
```



Reset the hold state to off for the upper subplot so that new plots replace the existing plots, which is the default behavior.

```
hold(ax1,'off')
y4 = sin(4*x);
plot(ax1,x,y4)
```



Input Arguments

ax — Axes object

axes object

Axes object. Use `ax` to set the hold state for a specific axes, instead of the current axes.

More About

Tips

- Use the `ishold` function to test the hold state.
- The `hold` function toggles the `NextPlot` property of the axes between `'add'` and `'replace'`.
- If an axes does not exist, then the `hold` command creates one.

See Also

`axes` | `cla` | `figure` | `ishold` | `newplot` | `subplot`

Introduced before R2006a

home

Send cursor home

Syntax

```
home
```

Description

`home` moves the cursor to the upper-left corner of the window. When using the MATLAB desktop, `home` also scrolls the visible text in the window up and out of view. You can use the scroll bar to see what was previously on the screen.

Examples

Execute a MATLAB command that displays something in the Command Window and then run the `home` function. `home` moves the cursor to the upper-left corner of the screen and clears the screen.

```
magic(5)

ans =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

home
```

See Also

`clc`

Introduced before R2006a

horzcat

Concatenate arrays horizontally

Syntax

```
C = horzcat(A1,...,AN)
```

Description

`C = horzcat(A1,...,AN)` horizontally concatenates arrays `A1,...,AN`. All arrays in the argument list must have the same number of rows.

- If the inputs are multidimensional arrays, `horzcat` concatenates along the second dimension. The first and remaining dimensions must match.
- If the inputs are tables, `horzcat` concatenates by matching row names when present, or by matching position for tables that do not have row names. All the table inputs must have unique variable names and the row names for all tables that have them must be identical, except for order.

  `horzcat` assigns values for the `Description` and `UserData` properties in `C` using the first nonempty value for the corresponding property in the tables `A1,...,AN`.

MATLAB calls `C = horzcat(A1, A2,...)` for the syntax `C = [A1 A2 ...]` when any of the inputs are an object.

Tips

You can concatenate categorical arrays with cell arrays of strings. For more information, see “Combine Categorical Arrays”.

If all the input arrays are ordinal categorical arrays, they must have the same sets of categories including their order. For more information, see “Ordinal Categorical Arrays”.

You can concatenate datetime arrays with cell arrays of strings.

You can concatenate duration arrays and calendar duration arrays. The result is a calendar duration array.

You can concatenate duration or calendar duration arrays with numeric arrays. Prior to concatenation, MATLAB converts the numeric array to an array of equivalent days using the `days` function.

For information on combining unlike integer types, integers with nonintegers, or cell arrays with non-cell arrays, see "Valid Combinations of Unlike Classes".

Examples

Horizontally Concatenate Two Matrices

Create a 3-by-5 matrix, `A`.

```
A = magic(5);
A(4:5,:) = []
```

```
A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
```

Create a 3-by-3 matrix, `B`.

```
B = magic(3)*100
```

```
B =

   800   100   600
   300   500   700
   400   900   200
```

Horizontally concatenate `A` and `B`.

```
C = horzcat(A,B)
```

```
C =
```

```
    17    24     1     8    15   800   100   600
    23     5     7    14    16   300   500   700
     4     6    13    20    22   400   900   200
```

Horizontally Concatenate Two Tables

Create a table, A, with three rows and two variables.

```
A = table([5;6;5],['M';'M';'M'],...
    'VariableNames',{'Age' 'Gender'},...
    'RowNames',{'Thomas' 'Gordon' 'Percy'})
```

```
A =

              Age    Gender
              ---    ------
    Thomas    5      M
    Gordon    6      M
    Percy     5      M
```

Create a table, B, with three rows and three variables.

```
B = table([45;41;40],[45;32;34],{'NY';'CA';'MA'},...
    'VariableNames',{'Height' 'Weight' 'Birthplace'},...
    'RowNames',{'Percy' 'Gordon' 'Thomas'})
```

```
B =

              Height    Weight    Birthplace
              ------    ------    ----------
    Percy     45        45        'NY'
    Gordon    41        32        'CA'
    Thomas    40        34        'MA'
```

Horizontally concatenate A and B.

```
C = horzcat(A,B)
```

```
C =

              Age    Gender    Height    Weight    Birthplace
              ---    ------    ------    ------    ----------
    Thomas    5      M         40        34        'MA'
    Gordon    6      M         41        32        'CA'
    Percy     5      M         45        45        'NY'
```

The order of rows in `C` matches the order in `A`.

More About

- "Concatenation Methods"

See Also

`vertcat` | `cat` | `strcat` | `char` | `special character`

Introduced before R2006a

horzcat (tscollection)

Horizontal concatenation for `tscollection` objects

Syntax

```
tsc = horzcat(tsc1,tsc2,...)
```

Description

`tsc = horzcat(tsc1,tsc2,...)` performs horizontal concatenation for `tscollection` objects:

```
tsc = [tsc1 tsc2 ...]
```

This operation combines multiple `tscollection` objects, which must have the same time vectors, into one `tscollection` containing `timeseries` objects from all concatenated collections.

See Also

`tscollection` | `vertcat (tscollection)`

Introduced before R2006a

hour

Hour number

Syntax

```
h = hour(t)
```

Description

`h = hour(t)` returns the hour numbers of the datetime values in `t`. The `h` output is a `double` array the same size as `t` and contains integer values from 0 to 23.

The `hour` function returns the hour numbers of datetime values. To assign hour values to a datetime array, `t`, use `t.Hour` and modify the `Hour` property.

Examples

Find Hour Number of Datetime Values

```
t = datetime('today'):hours(8):datetime('tomorrow');
t.Format = 'MMM dd, HH:mm'
```

```
t =

   Feb 23, 00:00   Feb 23, 08:00   Feb 23, 16:00   Feb 24, 00:00
```

```
h = hour(t)
```

```
h =

     0     8    16     0
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

`hms` | `minute` | `second` | `timeofday`

Introduced in R2014b

hours

Duration in hours

Syntax

```
H = hours(X)
```

Description

`H = hours(X)` returns an array of hours equivalent to the values in `X`.

- If `X` is a numeric array, then `H` is a `duration` array in units of hours.
- If `X` is a `duration` array, then `H` is a `double` array with each element equal to the number of hours in the corresponding element of `X`.

The `hours` function converts between `duration` and `double` values. To display a duration in units of hours, set its `Format` property to `'h'`.

Examples

Create Duration Array of Hours

```
X = magic(4);
H = hours(X)
```

```
H =

   16 hrs    2 hrs    3 hrs   13 hrs
    5 hrs   11 hrs   10 hrs    8 hrs
    9 hrs    7 hrs    6 hrs   12 hrs
    4 hrs   14 hrs   15 hrs    1 hr
```

Convert Durations to Numeric Array of Hours

Create a `duration` array.

```
X = hours(2:10:38) + minutes(30)

X =

    2.5 hrs   12.5 hrs   22.5 hrs   32.5 hrs
```

Convert each duration in X to a number of hours.

```
H = hours(X)

H =

    2.5000   12.5000   22.5000   32.5000
```

View the data type of H

```
whos H
  Name      Size             Bytes  Class     Attributes

  H         1x4                 32  double
```

Input Arguments

X — Input array

numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

More About

- “Set Date and Time Display Format”

See Also

duration

Introduced in R2014b

hsv2rgb

Convert HSV colormap to RGB colormap

Syntax

```
M = hsv2rgb(H)
rgb_image = hsv2rgb(hsv_image)
```

Description

`M = hsv2rgb(H)` converts a hue-saturation-value (HSV) colormap to a red-green-blue (RGB) colormap. `H` is an *m*-by-3 matrix, where *m* is the number of colors in the colormap. The columns of `H` represent hue, saturation, and value, respectively. `M` is an *m*-by-3 matrix. Its columns are intensities of red, green, and blue, respectively.

`rgb_image = hsv2rgb(hsv_image)` converts the HSV image to the equivalent RGB image. HSV is an *m*-by-*n*-by-3 image array whose three planes contain the hue, saturation, and value components for the image. RGB is returned as an *m*-by-*n*-by-3 image array whose three planes contain the red, green, and blue components for the image.

More About

Tips

As `H(:,1)` varies from 0 to 1, the resulting color varies from red through yellow, green, cyan, blue, and magenta, and returns to red. When `H(:,2)` is 0, the colors are unsaturated (i.e., shades of gray). When `H(:,2)` is 1, the colors are fully saturated (i.e., they contain no white component). As `H(:,3)` varies from 0 to 1, the brightness increases.

The MATLAB `hsv` colormap uses `hsv2rgb([huesaturationvalue])` where `hue` is a linear ramp from 0 to 1, and `saturation` and `value` are all 1's.

See Also

`brighten` | `colormap` | `rgb2hsv`

Introduced before R2006a

hypot

Square root of sum of squares

Syntax

```
c = hypot(a,b)
```

Description

`c = hypot(a,b)` returns the element-wise result of the following equation, computed to avoid underflow and overflow:

```
c = sqrt(abs(a).^2 + abs(b).^2)
```

Inputs `a` and `b` must follow these rules:

- Both `a` and `b` must be single- or double-precision, floating-point arrays.
- The sizes of the `a` and `b` arrays must either be equal, or one a scalar and the other nonscalar. In the latter case, `hypot` expands the scalar input to match the size of the nonscalar input.
- If `a` or `b` is an empty array (0-by-N or N-by-0), the other must be the same size or a scalar. The result `c` is an empty array having the same size as the empty input(s).

`hypot` returns the following in output `c`, depending upon the types of inputs:

- If the inputs to `hypot` are complex (`w+xi` and `y+zi`), then the statement `c = hypot(w+xi,y+zi)` returns the *positive real* result

  ```
  c = sqrt(abs(w).^2+abs(x).^2+abs(y).^2+abs(z).^2)
  ```

- If `a` or `b` is `±Inf`, `hypot` returns `Inf`.
- If neither `a` nor `b` is `Inf`, but one or both inputs is `NaN`, `hypot` returns `NaN`.
- If all inputs are finite, the result is finite. The one exception is when both inputs are very near the value of the MATLAB constant `realmax`. The reason for this is that the equation `c = hypot(realmax,realmax)` is theoretically `sqrt(2)*realmax`, which overflows to `Inf`.

Examples

Example 1

To illustrate the difference between using the `hypot` function and coding the basic `hypot` equation in M-code, create an anonymous function that performs the same function as `hypot`, but without the consideration to underflow and overflow that `hypot` offers:

```
myhypot = @(a,b)sqrt(abs(a).^2+abs(b).^2);
```

Find the upper limit at which your coded function returns a useful value. You can see that this test function reaches its maximum at about `1e154`, returning an infinite result at that point:

```
myhypot(1e153,1e153)
ans =
    1.4142e+153

myhypot(1e154,1e154)
ans =
   Inf
```

Do the same using the `hypot` function, and observe that `hypot` operates on values up to about `1e308`, which is approximately equal to the value for `realmax` on your computer (the largest double-precision floating-point number you can represent on a particular computer):

```
hypot(1e308,1e308)
ans =
  1.4142e+308

hypot(1e309,1e309)
ans =
   Inf
```

Example 2

`hypot(a,a)` theoretically returns `sqrt(2)*abs(a)`, as shown in this example:

```
x = 1.271161e308;

y = x * sqrt(2)
```

```
y =
  1.7977e+308

y = hypot(x,x)
y =
  1.7977e+308
```

See Also

`sqrt` | `abs` | `norm`

Introduced before R2006a

i

Imaginary unit

Syntax

```
1i
z = a + bi
z = x + 1i*y
```

Description

`1i` returns the basic imaginary unit. `i` is equivalent to `sqrt(-1)`.

You can use `i` to enter complex numbers. You also can use the character `j` as the imaginary unit. To create a complex number without using `i` and `j`, use the `complex` function.

`z = a + bi` returns a complex numerical constant, `z`.

`z = x + 1i*y` returns a complex array, `z`.

Examples

Complex Scalar

Create a complex scalar and use the character, `i`, without a multiplication sign as a suffix in forming a complex numerical constant.

```
z = 1+2i
```

```
z =

   1.0000 + 2.0000i
```

Complex Vector

Create a complex vector from two 4-by-1 vectors of real numbers.

```
x = [1:4]';
y = [8:-2:2]';

z = x+1i*y

z =

   1.0000 + 8.0000i
   2.0000 + 6.0000i
   3.0000 + 4.0000i
   4.0000 + 2.0000i
```

`z` is a 4-by-1 complex vector.

Complex Exponential

Create a complex scalar representing a complex vector with radius, `r`, and angle from the origin, `theta`.

```
r = 4;
theta = pi/4;

z = r*exp(1i*theta)

z =

   2.8284 + 2.8284i
```

Input Arguments

a — Real component of complex scalar

scalar

Real component of a complex scalar, specified as a scalar.

Data Types: `single | double`

b — Imaginary component of complex scalar

scalar

Imaginary component of a complex scalar, specified as a scalar.

If `b` is `double`, you can use the character, `i`, without a multiplication sign as a suffix in forming the complex numerical constant.

Example: `7i`

If `b` is `single`, you must use a multiplication sign when forming the complex numerical constant.

Example: `single(7)*i`

Data Types: `single` | `double`

x — Real component of complex array

scalar | vector | matrix | multidimensional array

Real component of a complex array, specified as a scalar, vector, matrix, or mulitdimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

y — Imaginary component of complex array

scalar | vector | matrix | multidimensional array

Imaginary component of a complex array, specified as a scalar, vector, matrix, or mulitdimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

Output Arguments

z — Complex array

scalar | vector | matrix | multidimensional array

Complex array, returned as a scalar, vector, matrix, or multidimensional array.

The size of `z` is the same as the input arguments.

`z` is `single` if at least one input argument is `single`. Otherwise, `z` is `double`.

More About

Tips

- For speed and improved robustness in complex arithmetic, use `1i` and `1j` instead of `i` and `j`.
- Since `i` is a function, it can be overridden and used as a variable. However, it is best to avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.
- Use the `complex` function to create a complex output in the following cases:
  - When the names `i` and `j` might be used for other variables (and do not equal `sqrt(-1)`)
  - When the inputs are not `double` or `single`
  - When the imaginary component is all zeros
- "Complex Numbers"

See Also

`complex` | `conj` | `imag` | `j` | `real`

Introduced before R2006a

ichol

Incomplete Cholesky factorization

Syntax

```
L = ichol(A)
L = ichol(A,opts)
```

Description

`L = ichol(A)` performs the incomplete Cholesky factorization of A with zero-fill.

`L = ichol(A,opts)` performs the incomplete Cholesky factorization of A with options specified by opts.

By default, `ichol` references the lower triangle of `A` and produces lower triangular factors.

Input Arguments

`A`

Sparse matrix

`opts`

Structure with up to five fields:

Field Name	Summary	Description
`type`	Type of factorization	String indicating which flavor of incomplete Cholesky to perform. Valid values of this field are `'nofill'` and `'ict'`. The `'nofill'` variant performs *incomplete Cholesky with zero-fill* (`IC(0)`). The `'ict'` variant performs

Field Name	Summary	Description
		incomplete Cholesky with threshold dropping (ICT). The default value is `'nofill'`.
`droptol`	Drop tolerance when type is `'ict'`	Nonnegative scalar used as a drop tolerance when performing ICT. Elements which are smaller in magnitude than a local drop tolerance are dropped from the resulting factor except for the diagonal element which is never dropped. The local drop tolerance at step `j` of the factorization is `norm(A(j:end,j),1)*droptol`. `'droptol'` is ignored if `'type'` is `'nofill'`. The default value is `0`.
`michol`	Indicates whether to perform modified incomplete Cholesky	Indicates whether or not *modified incomplete Cholesky* (MIC) is performed. The field may be `'on'` or `'off'`. When performing MIC, the diagonal is compensated for dropped elements to enforce the relationship `A*e = L*L'*e` where `e = ones(size(A,2),1)`. The default value is `'off'`.
`diagcomp`	Perform compensated incomplete Cholesky with the specified coefficient	Real nonnegative scalar used as a global diagonal shift `alpha` in forming the incomplete Cholesky factor. That is, instead of performing incomplete Cholesky on `A`, the factorization of `A + alpha*diag(diag(A))` is formed. The default value is `0`.

Field Name	Summary	Description
`shape`	Determines which triangle is referenced and returned	Valid values are `'upper'` and `'lower'`. If `'upper'` is specified, only the upper triangle of `A` is referenced and `R` is constructed such that `A` is approximated by `R'*R`. If `'lower'` is specified, only the lower triangle of `A` is referenced and `L` is constructed such that `A` is approximated by `L*L'`. The default value is `'lower'`.

Examples

Incomplete Cholesky Factorization

This example generates an incomplete Cholesky factorization.

Start with a symmetric positive definite matrix, `A`:

```
N = 100;
A = delsq(numgrid('S',N));
```

`A` is the two-dimensional, five-point discrete negative Laplacian on a 100-by-100 square grid with Dirichlet boundary conditions. The size of `A` is `98*98 = 9604` (not 10000 as the borders of the grid are used to impose the Dirichlet conditions).

The no-fill incomplete Cholesky factorization is a factorization which contains only nonzeros in the same position as `A` contains nonzeros. This factorization is extremely cheap to compute. Although the product `L*L'` is typically very different from `A`, the product `L*L'` will match `A` on its pattern up to round-off.

```
L = ichol(A);
norm(A-L*L','fro')./norm(A,'fro')
```

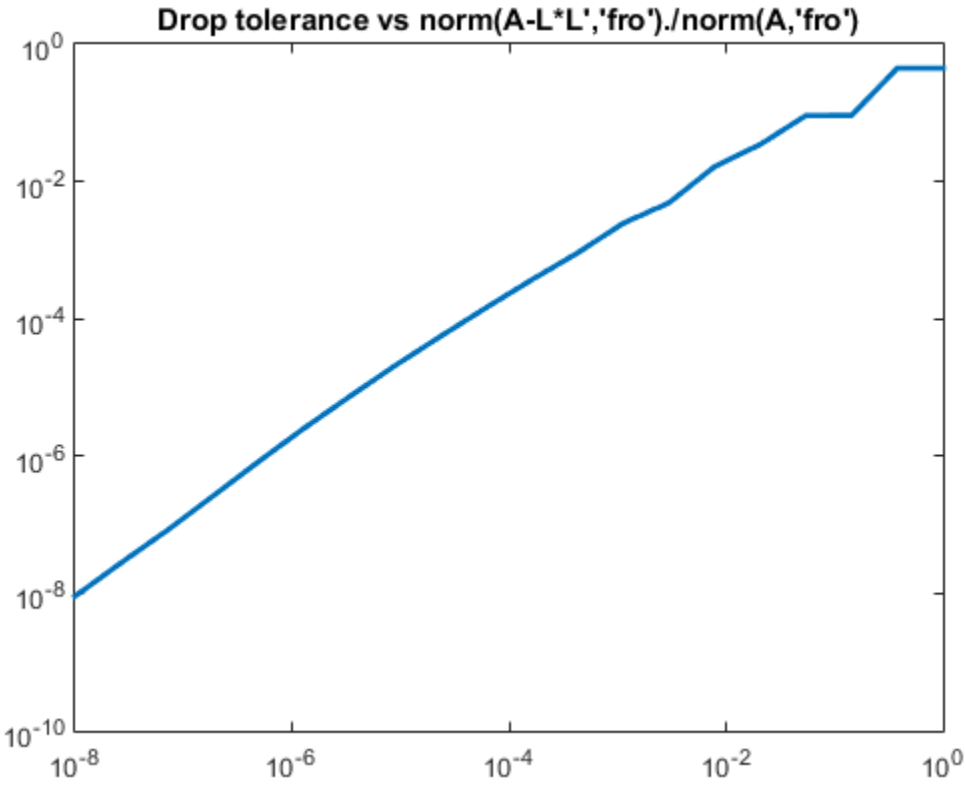
```
ans =

    0.0916
```

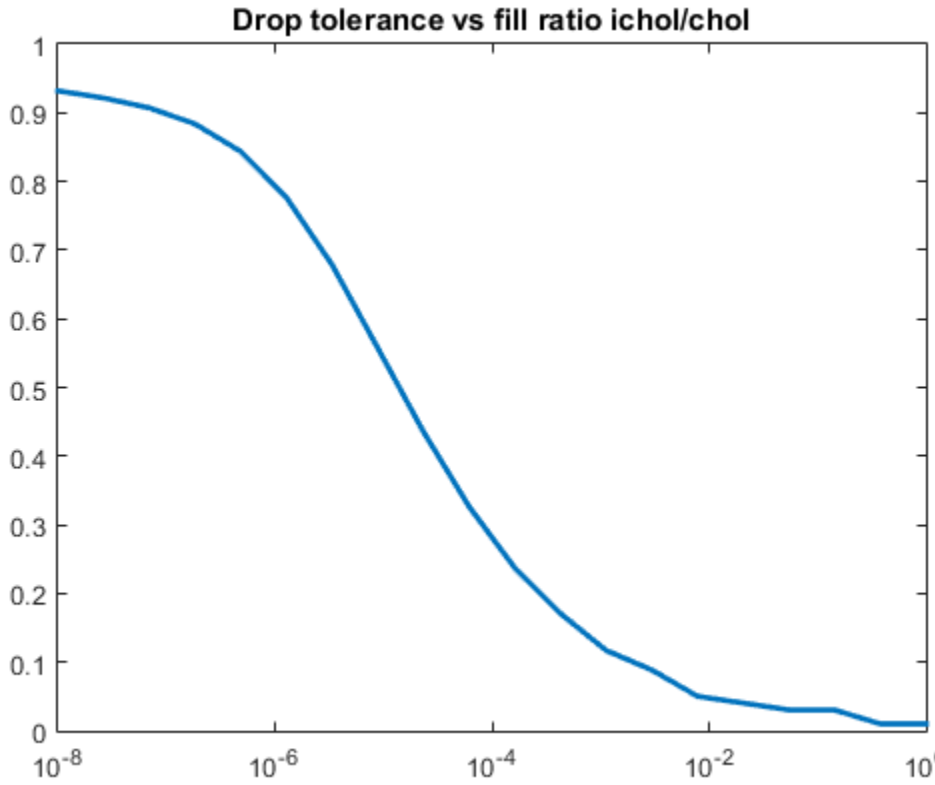
```
norm(A-(L*L').*spones(A),'fro')./norm(A,'fro')
```

```
ans =

   4.9606e-17
```

`ichol` may also be used to generate incomplete Cholesky factorizations with threshold dropping. As the drop tolerance decreases, the factor tends to get more dense and the product `L*L'` tends to be a better approximation of `A`. The following plots show the relative error of the incomplete factorization plotted against the drop tolerance as well as the ratio of the density of the incomplete factors to the density of the complete Cholesky factor.

```
n = size(A,1);
ntols = 20;
droptol = logspace(-8,0,ntols);
nrm = zeros(1,ntols);
nz = zeros(1,ntols);
nzComplete = nnz(chol(A,'lower'));
for k = 1:ntols
    L = ichol(A,struct('type','ict','droptol',droptol(k)));
    nz(k) = nnz(L);
    nrm(k) = norm(A-L*L','fro')./norm(A,'fro');
end
figure
loglog(droptol,nrm,'LineWidth',2)
title('Drop tolerance vs norm(A-L*L'',''fro'')./norm(A,''fro'')')
figure
semilogx(droptol,nz./nzComplete,'LineWidth',2)
title('Drop tolerance vs fill ratio ichol/chol')
```


Drop tolerance vs norm(A-L*L','fro')./norm(A,'fro')
10^0
10^-2
10^-4
10^-6
10^-8
10^-10
10^-8
10^-6
10^-4
10^-2
10^0



The relative error is typically on the same order as the drop tolerance, although this is not guaranteed.

Using ichol as a Preconditioner

This example shows how to use an incomplete Cholesky factorization as a preconditioner to improve convergence.

Create a symmetric positive definite matrix, A.

```
N = 100;
A = delsq(numgrid('S',N));
```

Create an incomplete Cholesky factorization as a preconditioner for `pcg`. Use a constant vector as the right hand side. As a baseline, execute `pcg` without a preconditioner.

```
b = ones(size(A,1),1);
tol = 1e-6;
maxit = 100;
[x0,fl0,rr0,it0,rv0] = pcg(A,b,tol,maxit);
```

Note that `fl0 = 1` indicating that `pcg` did not drive the relative residual to the requested tolerance in the maximum allowed iterations. Try the no-fill incomplete Cholesky factorization as a preconditioner.

```
L1 = ichol(A);
[x1,fl1,rr1,it1,rv1] = pcg(A,b,tol,maxit,L1,L1');
```

`fl1 = 0`, indicating that `pcg` converged to the requested tolerance and did so in 59 iterations (the value of `it1`). Since this matrix is a discretized Laplacian, however, using modified incomplete Cholesky can create a better preconditioner. A modified incomplete Cholesky factorization constructs an approximate factorization that preserves the action of the operator on the constant vector. That is, `norm(A*e-L*(L'*e))` will be approximately zero for `e = ones(size(A,2),1)` even though `norm(A-L*L','fro')/norm(A,'fro')` is not close to zero. It is not necessary to specify type for this syntax since `nofill` is the default, but it is good practice.

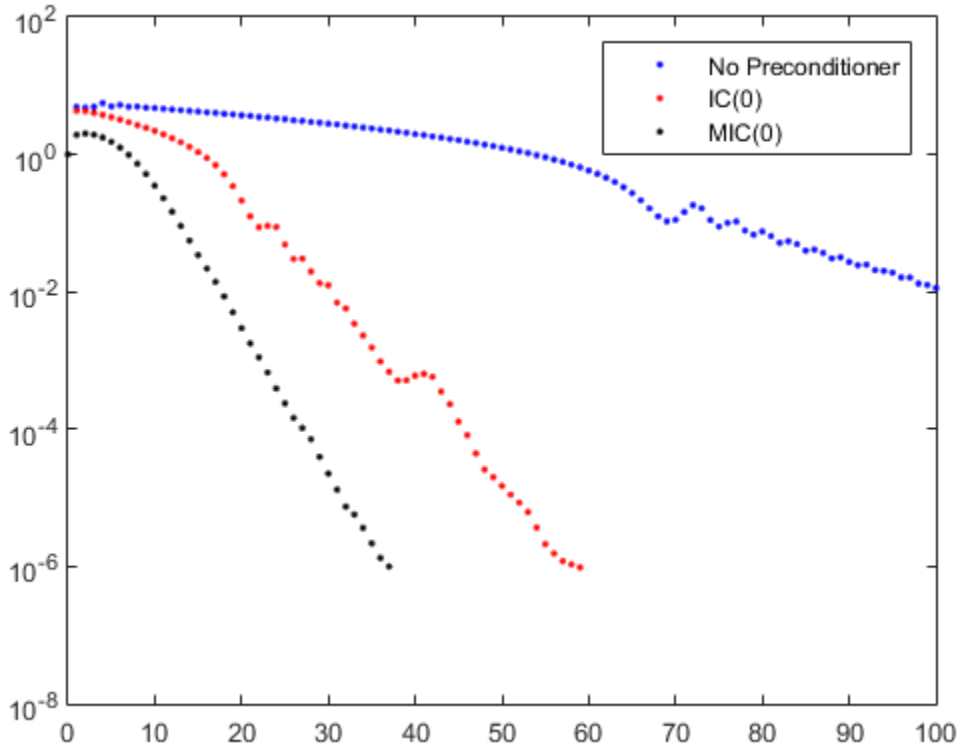
```
opts.type = 'nofill';
opts.michol = 'on';
L2 = ichol(A,opts);
e = ones(size(A,2),1);
norm(A*e-L2*(L2'*e))
[x2,fl2,rr2,it2,rv2] = pcg(A,b,tol,maxit,L2,L2');
```

```
ans =

   3.7983e-14
```

`pcg` converges (`fl2 = 0`) but in only 38 iterations. Plotting all three convergence histories shows the convergence.

```
semilogy(0:maxit,rv0./norm(b),'b.');
hold on
semilogy(0:it1,rv1./norm(b),'r.');
semilogy(0:it2,rv2./norm(b),'k.');
```

```
legend('No Preconditioner','IC(0)','MIC(0)');
```

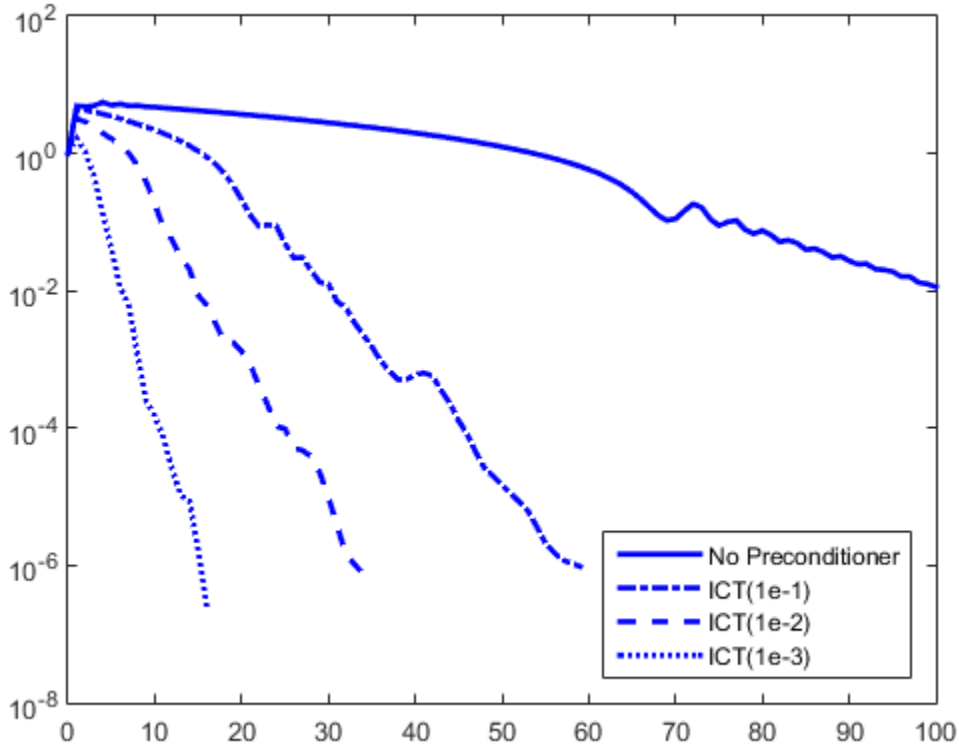


The plot shows that the modified incomplete Cholesky preconditioner creates a much faster convergence.

You can also try incomplete Cholesky factorizations with threshold dropping. The following plot shows convergence of `pcg` with preconditioners constructed with various drop tolerances.

```
L3 = ichol(A, struct('type','ict','droptol',1e-1));
[x3,fl3,rr3,it3,rv3] = pcg(A,b,tol,maxit,L3,L3');
L4 = ichol(A, struct('type','ict','droptol',1e-2));
[x4,fl4,rr4,it4,rv4] = pcg(A,b,tol,maxit,L4,L4');
```

```
L5 = ichol(A, struct('type','ict','droptol',1e-3));
[x5,fl5,rr5,it5,rv5] = pcg(A,b,tol,maxit,L5,L5');

figure
semilogy(0:maxit,rv0./norm(b),'b-','linewidth',2);
hold on
semilogy(0:it3,rv3./norm(b),'b-.','linewidth',2);
semilogy(0:it4,rv4./norm(b),'b--','linewidth',2);
semilogy(0:it5,rv5./norm(b),'b:','linewidth',2);
legend('No Preconditioner','ICT(1e-1)','ICT(1e-2)', ...
   'ICT(1e-3)','Location','SouthEast');
```



Note the incomplete Cholesky preconditioner constructed with drop tolerance `1e-2` is denoted as `ICT(1e-2)`.

As with the zero-fill incomplete Cholesky, the threshold dropping factorization can benefit from modification (i.e. `opts.michol = 'on'`) since the matrix arises from an elliptic partial differential equation. As with `MIC(0)`, the modified threshold based dropping incomplete Cholesky will preserve the action of the preconditioner on constant vectors, that is `norm(A*e-L*(L'*e))` will be approximately zero.

Using the diagcomp Option

This example illustrates the use of the `diagcomp` option of `ichol`.

Incomplete Cholesky factorizations of positive definite matrices do not always exist. The following code constructs a random symmetric positive definite matrix and attempts to solve a linear system using `pcg`.

```
S = rng('default');
A = sprandsym(1000,1e-2,1e-4,1);
rng(S);
b = full(sum(A,2));
[x0,fl0,rr0,it0,rv0] = pcg(A,b,1e-6,100);
```

Since convergence is not attained, try to construct an incomplete Cholesky preconditioner.

```
L = ichol(A,struct('type','ict','droptol',1e-3));
```

```
Error using ichol
Encountered nonpositive pivot.
```

If `ichol` breaks down as above, you can use the `diagcomp` option to construct a shifted incomplete Cholesky factorization. That is, instead of constructing `L` such that `L*L'` approximates `A`, `ichol` with diagonal compensation constructs `L` such that `L*L'` approximates `M = A + alpha*diag(diag(A))` without explicitly forming `M`. As incomplete factorizations always exist for diagonally dominant matrices, `alpha` can be found to make `M` diagonally dominant.

```
alpha = max(sum(abs(A),2)./diag(A))-2
```

```
alpha =

   62.9341
```

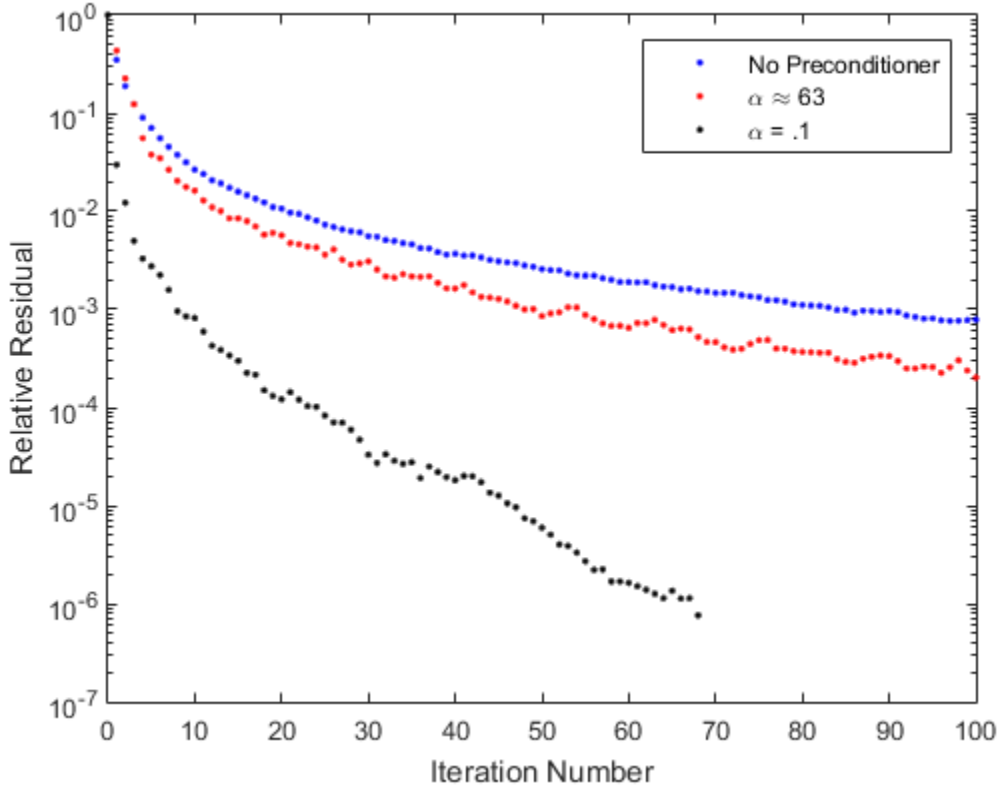
```
L1 = ichol(A, struct('type','ict','droptol',1e-3,'diagcomp',alpha));
[x1,fl1,rr1,it1,rv1] = pcg(A,b,1e-6,100,L1,L1');
```

Here, `pcg` still fails to converge to the desired tolerance within the desired number of iterations, but as the plot below shows, convergence is better for `pcg` with this preconditioner than with no preconditioner. Choosing a smaller `alpha` may help. With some experimentation, we can settle on an appropriate value for `alpha`.

```
alpha = .1;
L2 = ichol(A, struct('type','ict','droptol',1e-3,'diagcomp',alpha));
[x2,fl2,rr2,it2,rv2] = pcg(A,b,1e-6,100,L2,L2');
```

Now, `pcg` converges and a plot can show the convergence histories with each preconditioner.

```
semilogy(0:100,rv0./norm(b),'b.');
hold on;
semilogy(0:100,rv1./norm(b),'r.');
semilogy(0:it2,rv2./norm(b),'k.');
legend('No Preconditioner','\alpha \approx 63','\alpha = .1');
xlabel('Iteration Number');
ylabel('Relative Residual');
```



More About

Tips

- The factor given by this routine may be useful as a preconditioner for a system of linear equations being solved by iterative methods such as `pcg` or `minres`.
- `ichol` works only for sparse square matrices

References

[1] Saad, Yousef. “Preconditioning Techniques.” *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[2] Manteuffel, T.A. “An incomplete factorization technique for positive definite linear systems.” *Math. Comput.* 34, 473–497, 1980.

See Also

`ilu` | `chol` | `pcg` | `minres`

idivide

Integer division with rounding option

Syntax

```
C = idivide(A, B, opt)
C = idivide(A, B)
C = idivide(A, B, 'fix')
C = idivide(A, B, 'round')
C = idivide(A, B, 'floor')
C = idivide(A, B, 'ceil')
```

Description

`C = idivide(A, B, opt)` is the same as `A./B` for integer classes except that fractional quotients are rounded to integers using the optional rounding mode specified by `opt`. The default rounding mode is `'fix'`. Inputs `A` and `B` must be real and must have the same dimensions unless one is a scalar. The arguments `A` and `B` must belong to the same integer class. Alternatively, one of the arguments can be a scalar double, while the other can be any integer type except `int64` or `uint64`. The result `C` belongs to the integer class of the input arguments.

`C = idivide(A, B)` is the same as `A./B` except that fractional quotients are rounded toward zero to the nearest integers.

`C = idivide(A, B, 'fix')` is the same as the syntax shown immediately above.

`C = idivide(A, B, 'round')` is the same as `A./B` for integer classes. Fractional quotients are rounded to the nearest integers.

`C = idivide(A, B, 'floor')` is the same as `A./B` except that fractional quotients are rounded toward negative infinity to the nearest integers.

`C = idivide(A, B, 'ceil')` is the same as `A./B` except that the fractional quotients are rounded toward infinity to the nearest integers.

Examples

```
a = int32([-2 2]);
b = int32(3);

idivide(a,b)             % Returns [0 0]
idivide(a,b,'floor')     % Returns [-1 0]
idivide(a,b,'ceil')      % Returns [0 1]
idivide(a,b,'round')     % Returns [-1 1]
```

More About

- "Integers"
- "Floating-Point Numbers"

See Also

`ceil` | `fix` | `floor` | `ldivide` | `rdivide` | `round`

if, elseif, else

Execute statements if condition is true

Syntax

```
if expression
    statements
elseif expression
    statements
else
    statements
end
```

Description

`if` *`expression`*`, `*`statements`*`, end` evaluates an expression, and executes a group of statements when the expression is true. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

The `elseif` and `else` blocks are optional. The statements execute only if previous expressions in the `if...end` block are false. An `if` block can include multiple `elseif` blocks.

Examples

Use if, elseif, and else for Conditional Assignment

Create a matrix of 1s.

```
nrows = 4;
ncols = 6;
A = ones(nrows,ncols);
```

Loop through the matrix and assign each element a new value. Assign `2` on the main diagonal, `-1` on the adjacent diagonals, and `0` everywhere else.

```
for c = 1:ncols
```

```
    for r = 1:nrows

        if r == c
            A(r,c) = 2;
        elseif abs(r-c) == 1
            A(r,c) = -1;
        else
            A(r,c) = 0;
        end

    end
end
A
```

```
A =

     2    -1     0     0     0     0
    -1     2    -1     0     0     0
     0    -1     2    -1     0     0
     0     0    -1     2    -1     0
```

Compare Arrays

Expressions that include relational operators on arrays, such as `A > 0`, are true only when every element in the result is nonzero.

Test if any results are true using the `any` function.

```
limit = 0.75;
A = rand(10,1)
```

```
A =

    0.8147
    0.9058
    0.1270
    0.9134
    0.6324
    0.0975
    0.2785
    0.5469
    0.9575
```

```
    0.9649
```

```
if any(A > limit)
    disp('There is at least one value above the limit.')
else
    disp('All values are below the limit.')
end
```

```
There is at least one value above the limit.
```

Test Arrays for Equality

Compare arrays using `isequal` rather than the `==` operator to test for equality, because `==` results in an error when the arrays are different sizes.

Create two arrays.

```
A = ones(2,3);
B = rand(3,4,5);
```

If `size(A)` and `size(B)` are the same, concatenate the arrays; otherwise, display a warning and return an empty array.

```
if isequal(size(A),size(B))
   C = [A; B];
else
   disp('A and B are not the same size.')
   C = [];
end
```

```
A and B are not the same size.
```

Determine if Strings Match

Use `strcmp` to compare strings. Using `==` to test for equality results in an error when the strings are different sizes.

```
reply = input('Would you like to see an echo? (y/n): ','s');
if strcmp(reply,'y')
  disp(reply)
end
```

Evaluate Multiple Conditions in Expression

Determine if a value falls within a specified range.

```
x = 10;
minVal = 2;
maxVal = 6;

if (x >= minVal) && (x <= maxVal)
    disp('Value within specified range.')
elseif (x > maxVal)
    disp('Value exceeds maximum value.')
else
    disp('Value is below minimum value.')
end
```

```
Value exceeds maximum value.
```

More About

Expression

An expression can include relational operators (such as < or ==) and logical operators (such as &&, ||, or ~). Use the logical operators `and` and `or` to create compound expressions. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Within the conditional expression of an `if...end` block, logical operators & and | behave as short-circuit operators. This behavior is the same as && and ||, respectively. Since && and || consistently short-circuit in conditional expressions and statements, it is good practice to use && and || instead of & and | within the expression. For example,

```
x = 42;
if exist('myfunction.m','file') && (myfunction(x) >= pi)
    disp('Expressions are true')
end
```

The first part of the expression evaluates to false. Therefore, MATLAB does not need to evaluate the second part of the expression, which would result in an undefined function error.

Tips

- You can nest any number of `if` statements. Each `if` statement requires an `end` keyword.

- Avoid adding a space after `else` within the `elseif` keyword (`else if`). The space creates a nested `if` statement that requires its own `end` keyword.
- "Operators and Elementary Operations"

See Also

`for` | `return` | `switch` | `while`

Introduced before R2006a

ifft

Inverse fast Fourier transform

Syntax

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
y = ifft(..., 'symmetric')
y = ifft(..., 'nonsymmetric')
```

Description

`y = ifft(X)` returns the inverse discrete Fourier transform (DFT) of vector `X`, computed with a fast Fourier transform (FFT) algorithm. If `X` is a matrix, `ifft` returns the inverse DFT of each column of the matrix.

`ifft` tests `X` to see whether vectors in `X` along the active dimension are *conjugate symmetric*. If so, the computation is faster and the output is real. An `N`-element vector `x` is conjugate symmetric if `x(i) = conj(x(mod(N-i+1,N)+1))` for each element of `x`.

If `X` is a multidimensional array, `ifft` operates on the first non-singleton dimension.

`y = ifft(X,n)` returns the `n`-point inverse DFT of vector `X`.

`y = ifft(X,[],dim)` and `y = ifft(X,n,dim)` return the inverse DFT of `X` across the dimension `dim`.

`y = ifft(..., 'symmetric')` causes `ifft` to treat `X` as conjugate symmetric along the active dimension. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft(..., 'nonsymmetric')` is the same as calling `ifft(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft(fft(X))` equals `X` to within roundoff error.

Data Type Support

`ifft` supports inputs of data types `double` and `single`. If you call `ifft` with the syntax `y = ifft(X, ...)`, the output `y` has the same data type as the input `X`.

More About

Algorithms

The algorithm for `ifft(X)` is the same as the algorithm for `fft(X)`, except for a sign change and a scale factor of `n = length(X)`. As for `fft`, the execution time for `ifft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

See Also

`fft` | `fft2` | `ifft2` | `ifftn` | `ifftshift` | `fftw`

Introduced before R2006a

ifft2

2-D inverse fast Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
y = ifft2(..., 'symmetric')
y = ifft2(..., 'nonsymmetric')
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifft2` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `M`-by-`N` matrix `X` is conjugate symmetric if `X(i,j) = conj(X(mod(M-i+1, M) + 1, mod(N-j+1, N) + 1))` for each element of `X`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

`y = ifft2(..., 'symmetric')` causes `ifft2` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft2(..., 'nonsymmetric')` is the same as calling `ifft2(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error.

Data Type Support

`ifft2` supports inputs of data types `double` and `single`. If you call `ifft2` with the syntax `y = ifft2(X, ...)`, the output `y` has the same data type as the input `X`.

More About

Algorithms

The algorithm for `ifft2(X)` is the same as the algorithm for `fft2(X)`, except for a sign change and scale factors of `[m,n] = size(X)`. The execution time for `ifft2` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft2` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

See Also

`fft2` | `fftw` | `fftshift` | `ifft` | `ifftn` | `ifftshift`

Introduced before R2006a

ifftn

N-D inverse fast Fourier transform

Syntax

```
Y = ifftn(X)
Y = ifftn(X,siz)
y = ifftn(..., 'symmetric')
y = ifftn(..., 'nonsymmetric')
```

Description

`Y = ifftn(X)` returns the n-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifftn` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `N1`-by-`N2`-by- ... `Nk` array `X` is conjugate symmetric if

```
X(i1,i2, ...,ik) = conj(X(mod(N1-i1+1,N1)+1, mod(N2-i2+1,N2)+1,
... mod(Nk-ik+1,Nk)+1))
```

for each element of `X`.

`Y = ifftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the inverse transform. The size of the result `Y` is `siz`.

`y = ifftn(..., 'symmetric')` causes `ifftn` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifftn(..., 'nonsymmetric')` is the same as calling `ifftn(...)` without the argument `'nonsymmetric'`.

Data Type Support

`ifftn` supports inputs of data types `double` and `single`. If you call `ifftn` with the syntax `y = ifftn(X, ...)`, the output `y` has the same data type as the input `X`.

More About

Tips

For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error.

Algorithms

`ifftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = ifft(Y,[],p);
end
```

This computes in-place the one-dimensional inverse DFT along each dimension of `X`.

The execution time for `ifftn` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifftn` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

See Also

`fftn` | `fftw` | `ifft` | `ifft2` | `ifftshift`

Introduced before R2006a

ifftshift

Inverse FFT shift

Syntax

```
ifftshift(X)
ifftshift(X,dim)
```

Description

`ifftshift(X)` swaps the left and right halves of the vector `X`. For matrices, `ifftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth. If `X` is a multidimensional array, `ifftshift(X)` swaps "half-spaces" of `X` along each dimension.

`ifftshift(X,dim)` applies the `ifftshift` operation along the dimension `dim`.

Note: `ifftshift` undoes the results of `fftshift`. If the matrix `X` contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original `X`. Simply performing `fftshift(X)` twice will not produce `X`.

See Also

`fft` | `fft2` | `fftn` | `fftshift`

Introduced before R2006a

ilu

Sparse incomplete LU factorization

Syntax

```
ilu(A,setup)
[L,U] = ilu(A,setup)
[L,U,P] = ilu(A,setup)
```

Description

`ilu` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`ilu(A,setup)` computes the incomplete LU factorization of `A`. `setup` is an input structure with up to five setup options. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`type`	Type of factorization. Values for `type` include: • `'nofill'`(default)—Performs ILU factorization with `0` level of fill in, known as ILU(0). With `type` set to `'nofill'`, only the `milu` setup option is used; all other fields are ignored. • `'crout'`—Performs the Crout version of ILU factorization, known as ILUC. With `type` set to `'crout'`, only the `droptol` and `milu` setup options are used; all other fields are ignored. • `'ilutp'` —Performs ILU factorization with threshold and pivoting. If `type` is not specified, the ILU factorization with `0` level of fill in is performed. Pivoting is only performed with `type` set to `'ilutp'`.
`droptol`	Drop tolerance of the incomplete LU factorization. `droptol` is a non-negative scalar. The default value is `0`, which produces the complete LU factorization.

Field Name	Description

	The nonzero entries of `U` satisfy
	`abs(U(i,j)) >= droptol*norm(A(:,j)),`
	with the exception of the diagonal entries, which are retained regardless of satisfying the criterion. The entries of `L` are tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`
	`abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).`

`milu`	Modified incomplete LU factorization. Values for `milu` include: • `'row'`—Produces the row-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `A*e = L*U*e`, where `e` is the vector of ones. • `'col'`—Produces the column-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `e'*A = e'*L*U`. • `'off'` (default)—No modified incomplete LU factorization is produced.
`udiag`	If `udiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is `0`.
`thresh`	Pivot threshold between `0` (forces diagonal pivoting) and `1`, the default, which always chooses the maximum magnitude entry in the column to be the pivot.

`ilu(A,setup)` returns `L+U-speye(size(A))`, where `L` is a unit lower triangular matrix and `U` is an upper triangular matrix.

`[L,U] = ilu(A,setup)` returns a unit lower triangular matrix in `L` and an upper triangular matrix in `U`.

`[L,U,P] = ilu(A,setup)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

Limitations

`ilu` works on sparse square matrices only.

Examples

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'crout';
setup.milu = 'row';
setup.droptol = 0.1;
[L,U] = ilu(A,setup);
e = ones(size(A,2),1);
norm(A*e-L*U*e)

ans =

  1.4251e-014
```

This shows that `A` and `L*U`, where `L` and `U` are given by the modified Crout `ILU`, have the same row-sum.

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'nofill';
nnz(A)
ans =

        7840

nnz(lu(A))
ans =

      126478

nnz(ilu(A,setup))
ans =

        7840
```

This shows that A has 7840 nonzeros, the complete LU factorization has 126478 nonzeros, and the incomplete LU factorization, with 0 level of fill-in, has 7840 nonzeros, the same amount as A.

More About

Tips

These incomplete factorizations may be useful as preconditioners for a system of linear equations being solved by iterative methods such as BICG (BiConjugate Gradients), GMRES (Generalized Minimum Residual Method).

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

See Also

`bicg` | `ichol` | `gmres`

im2double

Convert image to double precision

Syntax

```
I2 = im2double(I)
I2 = im2double(I,'indexed')
```

Description

`I2 = im2double(I)` converts the intensity image `I` to double precision, rescaling the data if necessary. `I` can be a grayscale intensity image, a truecolor image, or a binary image.

If the input image is of class `double`, then the output image is identical.

`I2 = im2double(I,'indexed')` converts the indexed image `I` to double precision, offsetting the data if necessary.

Examples

Convert Image to Double Precision

```
I = imread('peppers.png');
whos I
```

```
  Name        Size                Bytes  Class    Attributes

  I         384x512x3            589824  uint8
```

```
I2 = im2double(I);
whos I2
```

```
  Name        Size                Bytes  Class     Attributes

  I2        384x512x3            4718592  double
```

Convert Image to Double Precision on GPU

Convert an array to class `double` on the GPU. This example requires the Parallel Computing Toolbox.

```
I1 = gpuArray(reshape(uint8(linspace(1,255,25)),[5 5]));
I2 = im2double(I1);
```

Input Arguments

`I` — Input image

scalar | vector | matrix | multidimensional array

Input image, specified as a scalar, vector, matrix, or multidimensional array.

- If `I` is an intensity or truecolor image, it can be `uint8`, `uint16`, `double`, `logical`, `single`, or `int16`.
- If `I` is an indexed image, it can be `uint8`, `uint16`, `double` or `logical`.
- If `I` is a binary image, it must be `logical`.

If the Parallel Computing Toolbox is installed, `I` can be a gpuArray and `im2double` converts `I` on a GPU.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

More About

Tips

- `im2double` supports the generation of efficient, production-quality C/C++ code if you have MATLAB Coder™ installed.

See Also

`double` | `gpuArray` | `im2int16` | `im2single` | `im2uint16` | `im2uint8`

im2frame

Convert image to movie frame

Syntax

```
f = im2frame(X,map)
f = im2frame(X)
```

Description

`f = im2frame(X,map)` converts the indexed image, `X`, and the associated colormap, `map`, into a movie frame `f`.

- If you specify `X` as an m-by-n array of integers, then `im2frame` uses the associated colormap, `map`, where `map` is a three-column array of values in the range [0,1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.
- If you specify `X` as an m-by-n-by-3 truecolor image, then `map` is optional and has no effect.

`f = im2frame(X)` converts the indexed image, `X`, into a movie frame `f` using the current colormap if `X` contains an indexed image.

For more information on image types, see "Image Types".

Class Support

`X` must be of class `uint8` or `double`.

Examples

Convert Image Sequence to Movie

Use `im2frame` to convert a sequence of images into a movie.

```
F(1) = im2frame(X1,map);
F(2) = im2frame(X2,map);
 ...
F(n) = im2frame(Xn,map);
movie(F)
```

See Also

`frame2im` | `movie`

Introduced before R2006a

im2java

Convert image to Java image

Syntax

```
jimage = im2java(I)
jimage = im2java(X,MAP)
jimage = im2java(RGB)
```

Description

To work with a MATLAB image in the Java environment, you must convert the image from its MATLAB representation into an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(I)` converts the intensity image `I` to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(X,MAP)` converts the indexed image `X`, with colormap `MAP`, to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(RGB)` converts the RGB image `RGB` to an instance of the Java image class, `java.awt.Image`.

Class Support

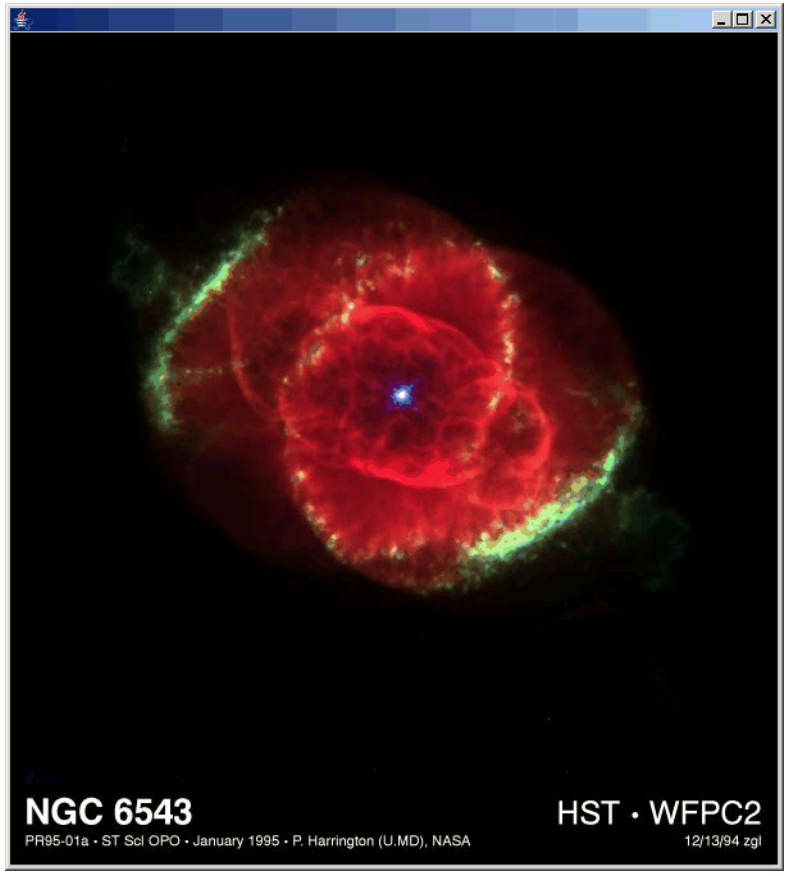
The input image can be of class `uint8`, `uint16`, or `double`.

Note Java requires `uint8` data to create an instance of the Java image class, `java.awt.Image`. If the input image is of class `uint8`, `jimage` contains the same `uint8` data. If the input image is of class `double` or `uint16`, `im2java` makes an equivalent image of class `uint8`, rescaling or offsetting the data as necessary, and then converts this `uint8` representation to an instance of the Java image class, `java.awt.Image`.

Examples

This example reads an image into the MATLAB workspace and then uses `im2java` to convert it into an instance of the Java image class.

```
I = imread('ngc6543a.jpg');
javaImage = im2java(I);
frame = javax.swing.JFrame;
icon = javax.swing.ImageIcon(javaImage);
label = javax.swing.JLabel(icon);
frame.getContentPane.add(label);
frame.pack
frame.show
```



Introduced before R2006a

imag

Imaginary part of complex number

Syntax

```
Y = imag(Z)
```

Description

`Y = imag(Z)` returns the imaginary part of the elements of array `Z`.

Examples

```
imag(2+3i)

ans =

    3
```

See Also

`conj` | `i` | `j` | `real`

Introduced before R2006a

image

Display image from array

Syntax

```
image(C)
image(x,y,C)
image('CData',C)
image('XData',x,'YData',y,'CData',C)

image(___,Name,Value)

im = image(___)
```

Description

`image(C)` displays the data in array C as an image. Each element of C specifies the color for one pixel of the image. The resulting image is an `m`-by-`n` grid of pixels where `m` is the number of columns and `n` is the number of rows in `C`. The row and column indices of the elements determine the centers of the corresponding pixels.

`image(x,y,C)` specifies the image location. Use `x` and `y` to specify the locations of the corners corresponding to `C(1,1)` and `C(m,n)`. To specify both corners, set `x` and `y` as two-element vectors. To specify the first corner and let `image` determine the other, set `x` and `y` as scalar values. The image is stretched and oriented as applicable.

`image('CData',C)` adds the image to the current axes without replacing existing plots. This syntax is the low-level version of `image(C)`. For more information, see "High-Level Versus Low-Level Version of Image" on page 1-3811.

`image('XData',x,'YData',y,'CData',C)` specifies the image location. This syntax is the low-level version of `image(x,y,C)`.

`image(___,Name,Value)` specifies image properties using one or more name-value pair arguments. You can specify image properties with any of the input argument combinations in the previous syntaxes.

`im = image(___)` returns the image object created. Use `im` to set properties of the image after it is created. You can specify this output with any of the input argument

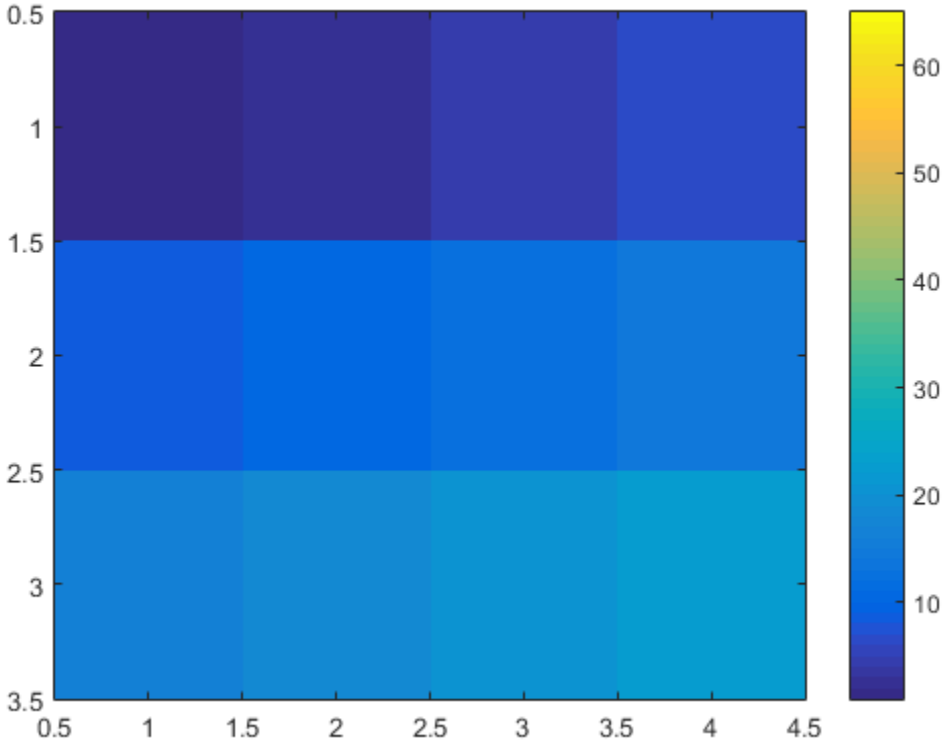
combinations in the previous syntaxes. For a list of image properties and descriptions, see Image Properties.

Examples

Display Image of Matrix Data

Create matrix `C`. Display an image of the data in `C`. Add a colorbar to the graph to show the current colormap.

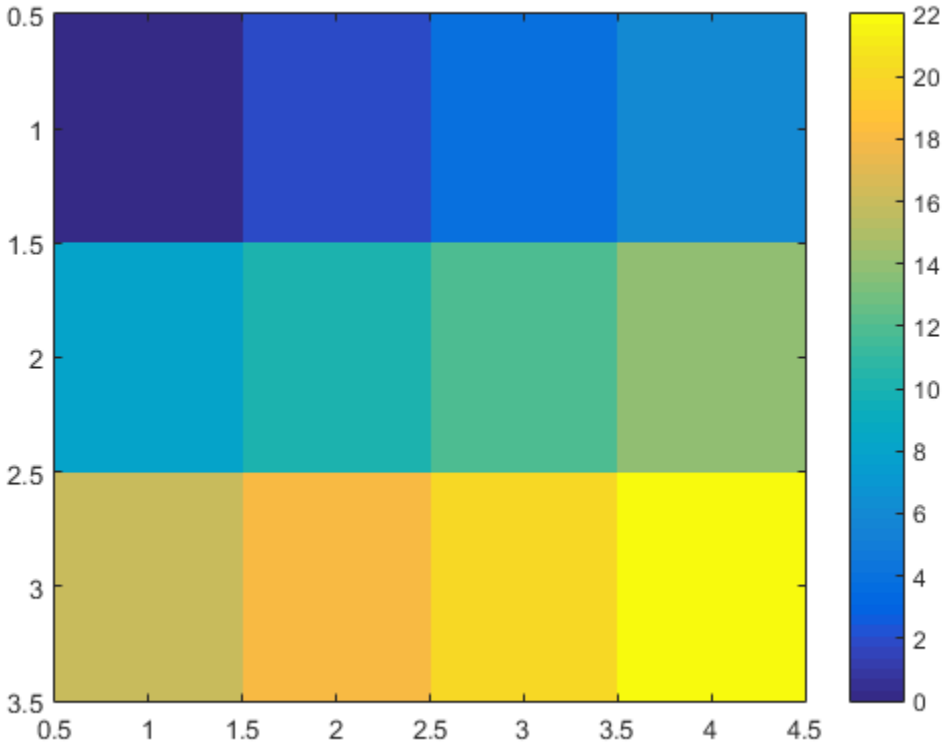
```
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
image(C)
colorbar
```



By default, the `CDataMapping` property for the image is set to `'direct'` so `image` interprets values in `C` as indices into the colormap. For example, the bottom right pixel corresponding to the last element in `C`, 22, uses the 22nd color of the colormap.

Scale the values to the full range of the current colormap by setting the `CDataMapping` property to `'scaled'` when creating the image.

```
image(C,'CDataMapping','scaled')
colorbar
```

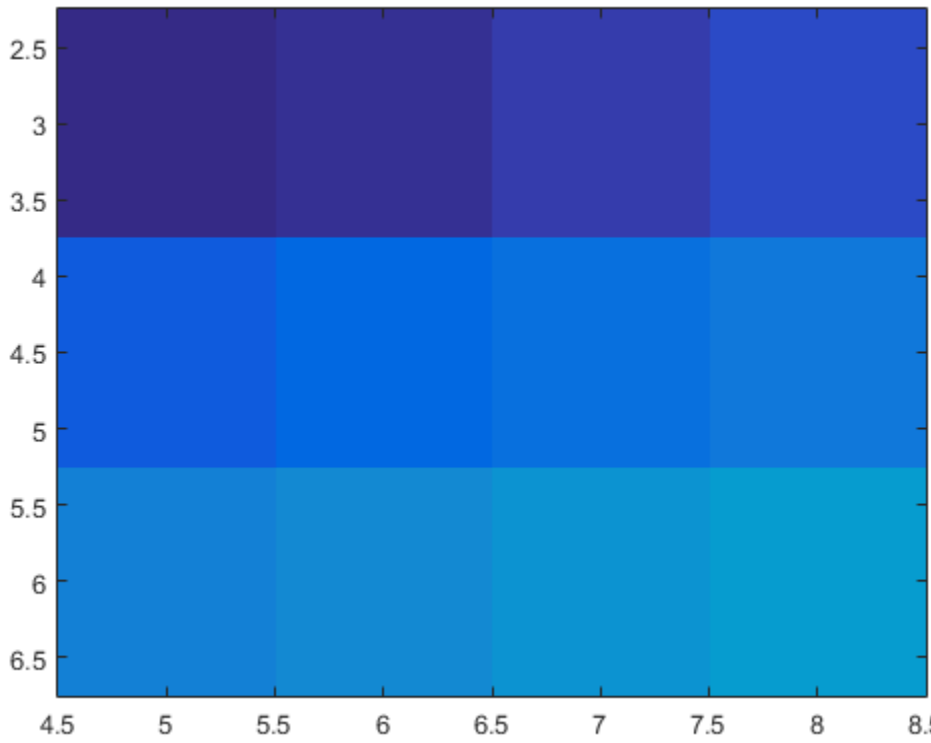


Alternatively, you can use the `imagesc` function to scale the values instead of using `image(C,'CDataMapping','scaled')`. For example, use `imagesc(C)`.

Control Image Placement

Place the image so that it lies between 5 and 8 on the *x*-axis and 3 and 6 on the *y*-axis.

```
x = [5 8];
y = [3 6];
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
image(x,y,C)
```



Notice that the pixel corresponding to C(1,1) is centered over the point (5,3). The pixel corresponding to C(3,4) is centered over the point (8,6). `image` positions and orients the rest of the image between those two points.

Display Image of 3-D Array of True Colors

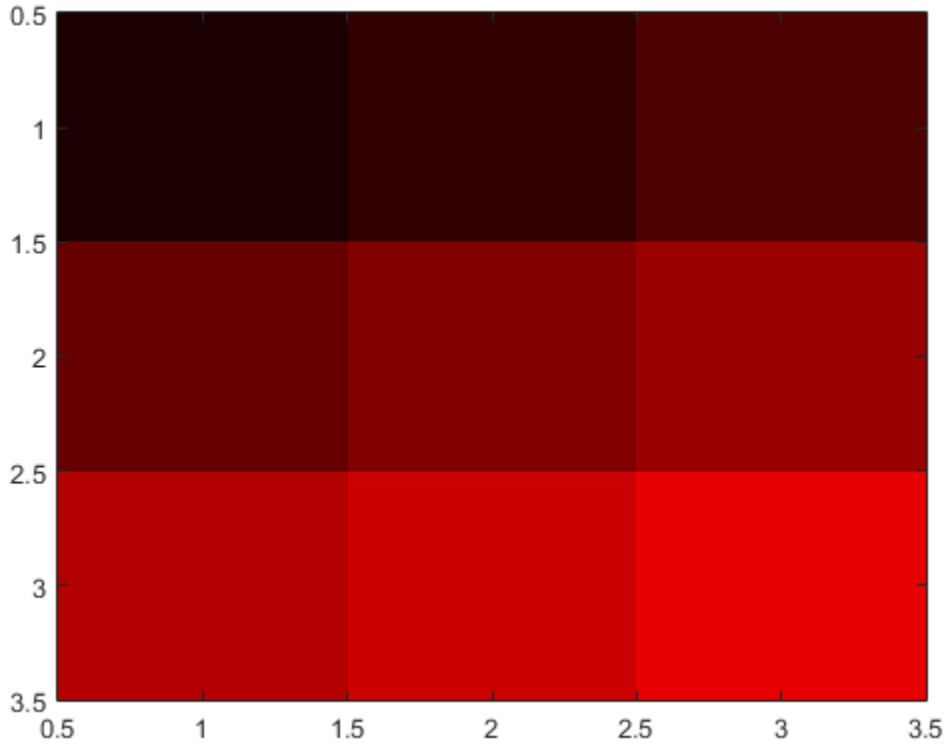
Create `C` as a 3-D array of true colors. Use only red colors by setting the last two pages of the array to zeros.

```
C = zeros(3,3,3);
C(:,:,1) = [.1 .2 .3; .4 .5 .6; .7 .8 .9]
```

```
C(:,:,1) =

    0.1000    0.2000    0.3000
    0.4000    0.5000    0.6000
    0.7000    0.8000    0.9000


C(:,:,2) =

     0     0     0
     0     0     0
     0     0     0


C(:,:,3) =

     0     0     0
     0     0     0
     0     0     0
```

Display an image of the data in `C`.

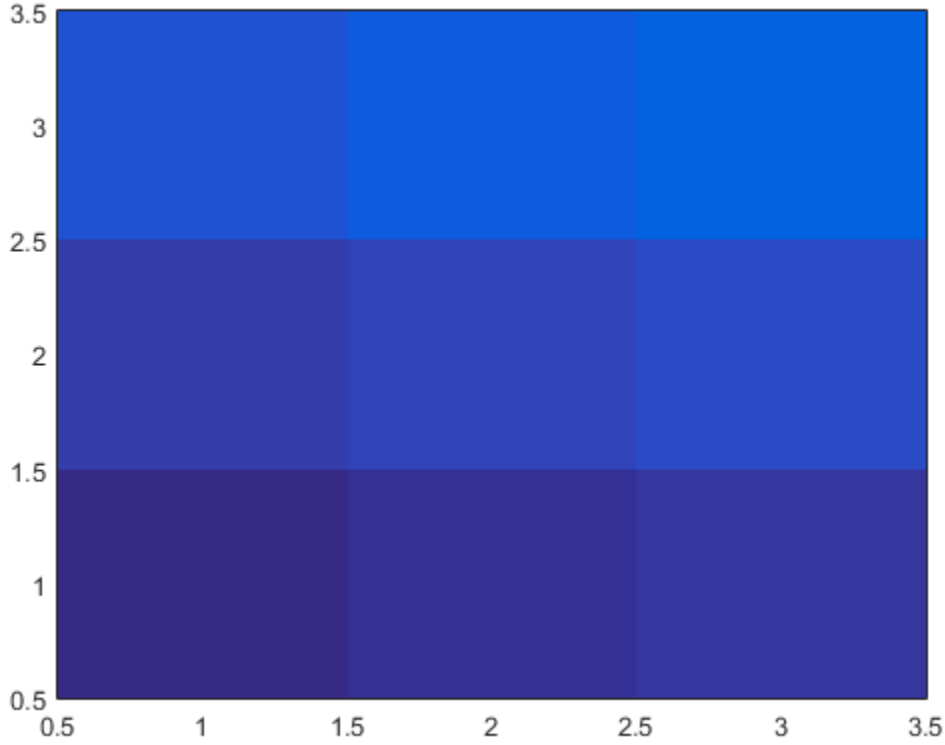
```
image(C)
```



Modify Image After Creation

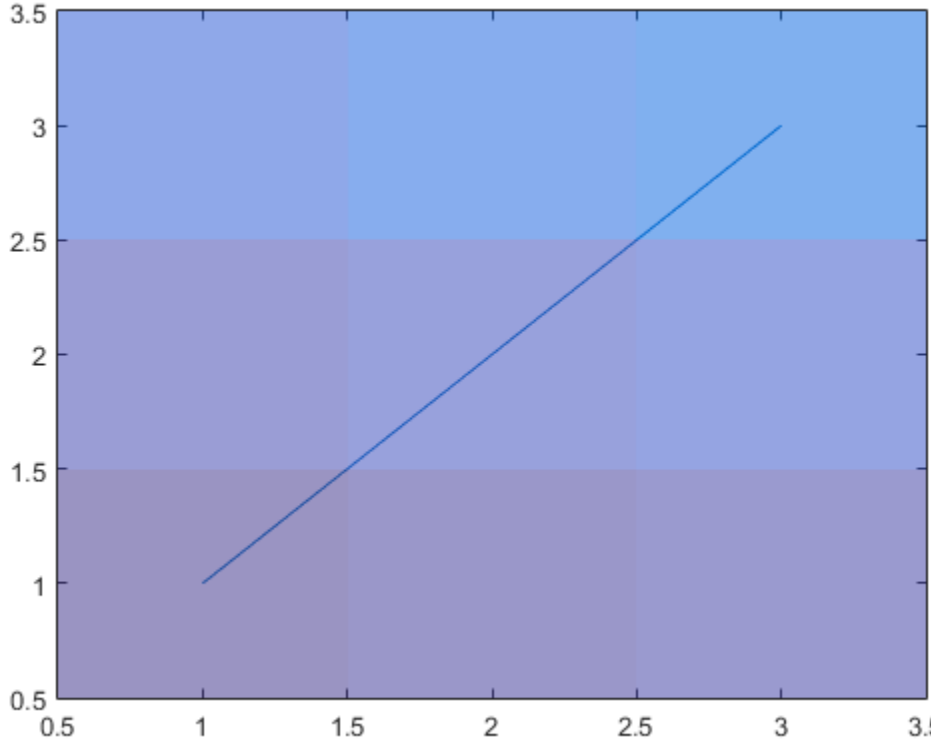
Plot a line, and then create an image on top of the line. Return the image object.

```
plot(1:3)
hold on
C = [1 2 3; 4 5 6; 7 8 9];
im = image(C);
```



Make the image semitransparent so that the line shows through the image.

```
im.AlphaData = 0.5;
```



Read and Display JPEG Image File

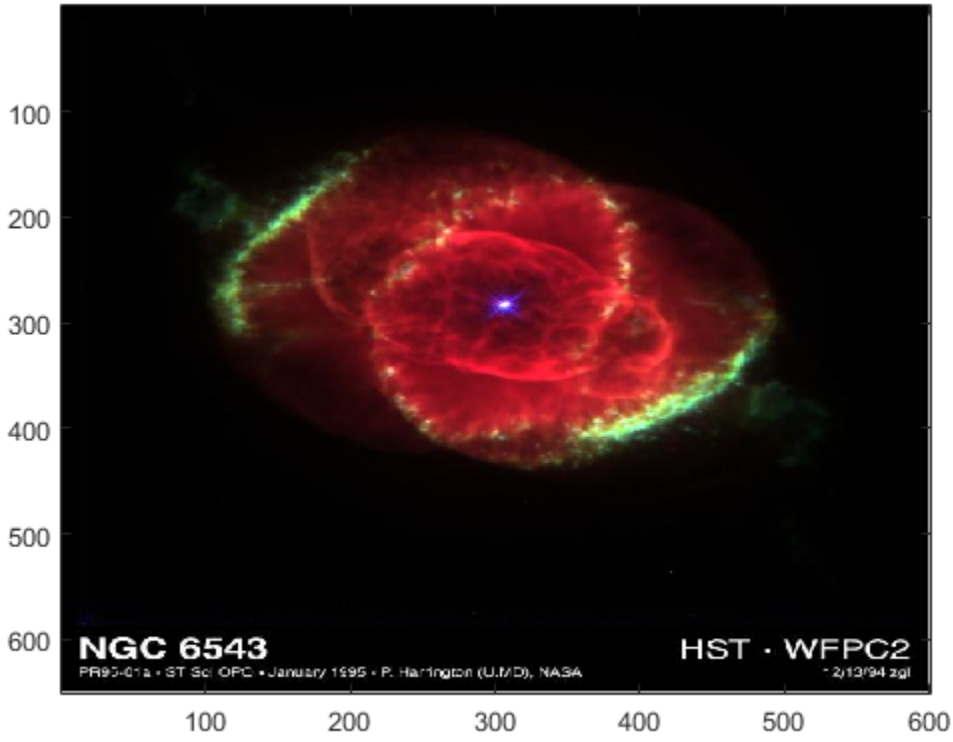
Read a JPEG image file.

```
C = imread('ngc6543a.jpg');
```

`imread` returns a 650-by-600-by-3 array, `C`.

Display the image.

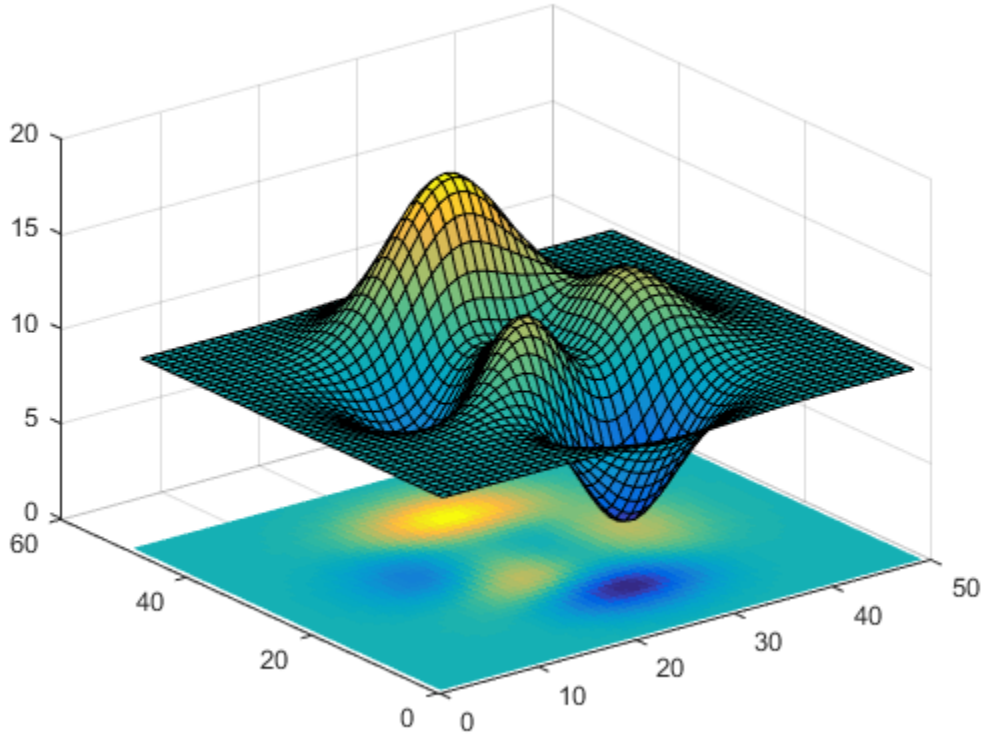
```
image(C)
```



Add Image to Axes in 3-D View

Create a surface plot. Then, add an image under the surface. The image displays in the *xy*-plane.

```
Z = 10 + peaks;
surf(Z)
hold on
image(Z,'CDataMapping','scaled')
```



Input Arguments

`C` — Image color data

vector or matrix | 3-D array of RGB triplets

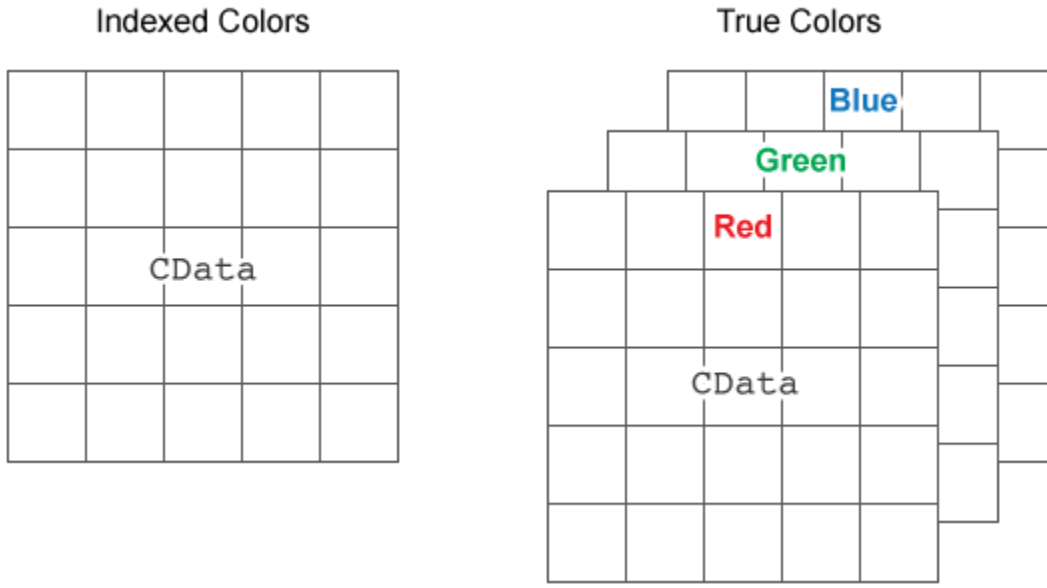
Image color data, specified in one of these forms:

- Vector or matrix — This format defines indexed image data. Each element of `C` defines a color for 1 pixel of the image. For example, `C = [1 2 3; 4 5 6; 7 8 9];`. The elements of `C` map to colors in the colormap. The CDataMapping property controls the mapping method.

- 3-D array of RGB triplets — This format defines true color image data using RGB triplet values. Each RGB triplet defines a color for 1 pixel of the image. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of the color. The first page of the 3-D array contains the red components, the second page contains the green components, and the third page contains the blue components. Since the image uses true colors instead of colormap colors, the CDataMapping property has no effect.

  - If `C` is of type `double`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.
  - If `C` is an integer type, then the image uses the full range of data to determine the color. For example, if `C` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.
  - If `C` is of type `logical`, then `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.

This illustration shows the relative dimensions of C for the two color models.



The behavior of `NaN` elements is not defined.

To use the low-level version of the `image` function instead, set the `CData` property as a name-value pair. For example, `image('CData',C)`.

Converting Between Data Types

To convert indexed image data from an integer type to type `double`, add 1. For example, if `X8` is indexed image data of type `uint8`, convert it to type `double` using:

```
X64 = double(X8) + 1;
```

To convert indexed image data from type `double` to an integer type, subtract 1 and use `round` to ensure that all the values are integers. For example, if `X64` is indexed image data of type `double`, convert it to `uint8` using:

```
X8 = uint8(round(X64 - 1));
```

To convert true color image data from an integer type to type `double`, rescale the data. For example, if `RGB8` is true color image data of type `uint8`, convert it to `double` using:

```
RGB64 = double(RGB8)/255;
```

To convert true color image data from type `double` to an integer type, rescale the data and use `round` to ensure that all the values are integers. For example, if `RGB64` is image data of type `double`, convert it to `uint8` using:

```
RGB8 = uint8(round(RGB64*255));
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

x — Placement along x-axis

`[1 size(C,2)]` (default) | two-element vector | scalar

Placement along the *x*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `C(1,1)` and the second element as the location for the center of `C(m,n)`, where `[m,n] = size(C)`. If `C` is a 3-D array, then `m` and `n` are the first two dimensions. Evenly distribute the centers of the remaining elements of C between those two points.

  The width of each pixel is determined by the expression:

```
(x(2)-x(1))/(size(C,2)-1)
```

If `x(1) > x(2)`, then the image is flipped left-right.

- Scalar — Center `C(1,1)` at this location and each following element one unit apart.

To use the low-level version of the `image` function instead, set the `XData` property as a name-value pair. For example, `image('XData',x,'YData',y,'CData',C)`.

You cannot interactively pan or zoom outside the x-axis limits or y-axis limits of an image, unless the limits are already set outside the bounds of the image. If the limits are already outside the bounds, there is no such restriction. If other objects (such as a line) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

y — Placement along y-axis

`[1 size(C,1)]` (default) | two-element vector | scalar

Placement along *y*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `C(1,1)` and the second element as the location for the center of `C(m,n)`, where `[m,n] = size(C)`. If `C` is a 3-D array, then `m` and `n` are the first two dimensions. Evenly distribute the centers of the remaining elements of C between those two points.

  The height of each pixel is determined by the expression:

  ```
  (y(2)-y(1))/(size(C,1)-1)
  ```

  If `y(1) > y(2)`, then the image is flipped up-down.
- Scalar — Center `C(1,1)` at this location and each following element one unit apart.

To use the low-level version of the `image` function instead, set the `YData` property as a name-value pair. For example, `image('XData',x,'YData',y,'CData',C)`.

You cannot interactively pan or zoom outside the x-axis limits or y-axis limits of an image, unless the limits are already set outside the bounds of the image. If the limits are already outside the bounds, there is no such restriction. If other objects (such as a line) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `image([1 2 3],'AlphaData',0.5)` displays a semitransparent image.

The properties listed here are a subset of image properties. For a complete list, see Image Properties.

`'CDataMapping'` — Color data mapping method

`'direct'` (default) | `'scaled'`

Color data mapping method, specified as `'direct'` or `'scaled'`. Use this property to control the mapping of color data values in CData into the colormap. `CData` must be a vector or a matrix defining indexed colors. This property has no effect if CData is a 3-D array defining true colors.

The methods have these effects:

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
  - If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
  - If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.
- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

'AlphaData' — Transparency data

1 (default) | scalar | array the same size as CData

Transparency data, specified in one of these forms:

- Scalar — Use a consistent transparency across the entire image.
- Array the same size as CData — Use a different transparency value for each image element.

The AlphaDataMapping property controls how the transparency data values are interpreted.

Example: 0.5

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

'AlphaDataMapping' — Transparency data mapping method

'none' (default) | 'scaled' | 'direct'

Transparency data mapping method, specified as 'none', 'scaled', or 'direct'. Use this property to control how the transparency values in AlphaData are interpreted.

The methods have these effects:

- 'none' — Clamp the values to the region between 0 and 1. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.
- 'scaled' — Scale the values to range between the minimum and maximum alpha limits of the axes. The ALim property of the axes contains the alpha limits.
- 'direct' — Interpret the values as indices into the figure alphamap. The Alphamap property of the figure contains the alphamap. Values with a decimal portion are fixed to the nearest lower integer.

  - If the values are of type double or single, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap are mapped to the last element in the alphamap.
  - If the values are of type uint8, uint16, uint32, uint64 , int8, int16, int32, or int64 then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap are mapped to the last element in the alphamap (or up to the range limits of the type).

- If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

Output Arguments

`im` — Image object

scalar

Image object, returned as a scalar. Use `im` to set properties of the image after it is created. For a list, see Image Properties.

More About

High-Level Versus Low-Level Version of Image

The `image` function has two versions, the high-level version and the low-level version. If you use `image` with `'CData'` as an input argument, then you are using the low-level version. Otherwise, you are using the high-level version.

The high-level version of `image` calls `newplot` before plotting and sets these axes properties:

- `Layer` to `'top'`. The image is shown in front of any tick marks or grid lines.
- `YDir` to `'reverse'`. Values along the *y*-axis decrease from top to bottom. To increase the values from top to bottom, set `YDir` to `'normal'`. This setting reverses both the *y*-axis and the image.
- `View` to `[0 90]`.

The low-level version of the `image` function does not call `newplot` and does not set these axes properties.

Tips

- To read image data into MATLAB from graphics files in various standard formats, such as TIFF, use `imread`. To write MATLAB image data to graphics files, use `imwrite`. The `imread` and `imwrite` functions support a variety of graphics file formats and compression schemes.

See Also

Functions
`colormap` | `imagesc` | `imfinfo` | `imread` | `imshow` | `imwrite`

Properties
Image Properties

Introduced before R2006a

Image Properties

Control image appearance and behavior

Image properties control the appearance and behavior of image objects. By changing property values, you can modify certain aspects of the image.

Starting in R2014b, you can use dot notation to query and set properties.

```
im = image(rand(20));
C = im.CData;
im.CDataMapping = 'scaled';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Image Data

`CData` — Image color data

64-by-64 array (default) | vector or matrix | 3-D array of RGB triplets

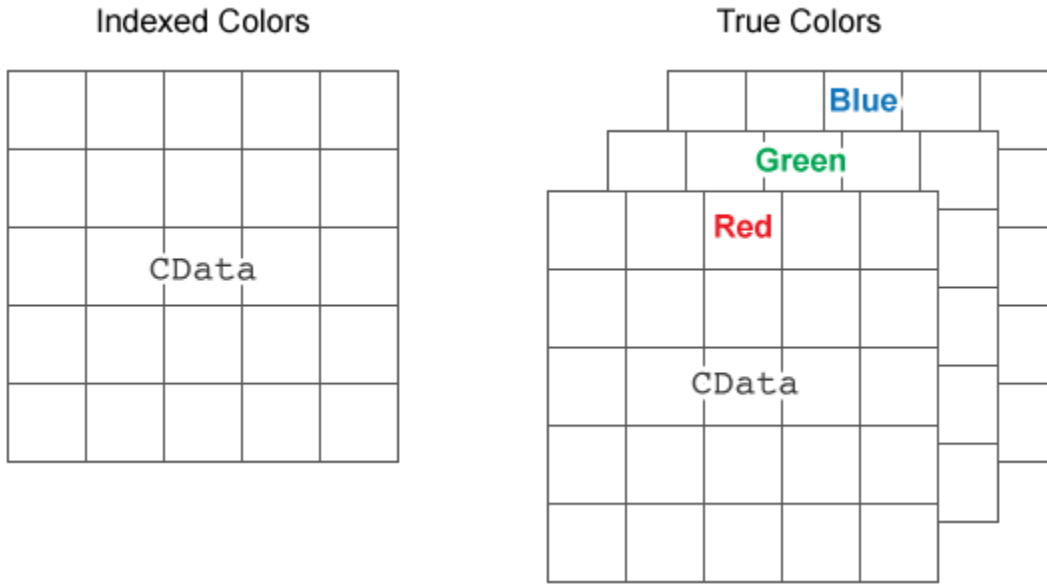
Image color data, specified in one of these forms:

- Vector or matrix — This format defines indexed image data. Each element defines a color for one pixel of the image. The elements map to colors in the colormap. The CDataMapping property controls the mapping method.
- 3-D array of RGB triplets — This format defines true color image data using RGB triplet values. Each RGB triplet defines a color for one pixel of the image. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of the color. The first page of the 3-D array contains the red components, the second page contains the green components, and the third page contains the blue components. Since the image uses true colors instead of colormap colors, the CDataMapping property has no effect.

  - If `CData` is of type `double`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.
  - If `CData` is an integer type, then the image uses the full range of data to determine the color. For example, if `CData` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.

- If `CData` is of type `logical`, then `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.

This illustration shows the relative dimensions of CData for the two color models.



The behavior of `NaN` elements is not defined.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`CDataMapping` — Color data mapping method

`'direct'` (default) | `'scaled'`

Color data mapping method, specified as `'direct'` or `'scaled'`. Use this property to control the mapping of color data values in CData into the colormap. `CData` must be a vector or a matrix defining indexed colors. This property has no effect if CData is a 3-D array defining true colors.

The methods have these effects:

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.

- If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
- If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
- If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.

- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

Image Position

`XData` — Placement along x-axis

`[1 size(CData,2)]` (default) | two-element vector | scalar

Placement along the *x*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `CData(1,1)` and the second element as the location for the center of `CData(m,n)`, where `[m,n] = size(CData)`. Evenly distribute the centers of the remaining CData elements between those two points.

  The width of each pixel is determined by the expression:

  ```
  (XData(2)-XData(1))/(size(CData,2)-1)
  ```

  If `XData(1)` > `XData(2)`, then the image is flipped left-right.
- Scalar — Center `CData(1,1)` at this location and each following element one unit apart.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`YData` — Placement along y-axis

`[1 size(CData,1)]` (default) | two-element vector | scalar

Placement along *y*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `CData(1,1)` and the second element as the location for the center of `CData(m,n)`, where `[m,n] = size(CData)`. Evenly distribute the centers of the remaining CData elements between those two points.

  The height of each pixel is determined by the expression:

  ```
  (YData(2)-YData(1))/(size(CData,1)-1)
  ```

  If `YData(1)` > `YData(2)`, then the image is flipped up-down.
- Scalar — Center `CData(1,1)` at this location and each following element one unit apart.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Image Transparency

`AlphaData` — Transparency data

1 (default) | scalar | array the same size as `CData`

Transparency data, specified in one of these forms:

- Scalar — Use a consistent transparency across the entire image.
- Array the same size as CData — Use a different transparency value for each image element.

The AlphaDataMapping property controls how the transparency data values are interpreted.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Transparency data mapping method

`'none'` (default) | `'scaled'` | `'direct'`

Transparency data mapping method, specified as `'none'`, `'scaled'`, or `'direct'`. Use this property to control how the transparency values in AlphaData are interpreted.

The methods have these effects:

- `'none'` — Clamp the values to the region between 0 and 1. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.
- `'scaled'` — Scale the values to range between the minimum and maximum alpha limits of the axes. The `ALim` property of the axes contains the alpha limits.
- `'direct'` — Interpret the values as indices into the figure alphamap. The `Alphamap` property of the figure contains the alphamap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap are mapped to the last element in the alphamap.
  - If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64` then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap are mapped to the last element in the alphamap (or up to the range limits of the type).
  - If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

Visibility

`Visible` — Visibility of image

`'on'` (default) | `'off'`

Visibility of image, specified as one of these values:

- `'on'` — Display the image.
- `'off'` — Hide the image without deleting it. You still can access the properties of an invisible image object.

`Clipping` — Clipping of image to axes limits

`'on'` (default) | `'off'`

Clipping of image to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the image that are outside the axes limits.

- `'off'` — Display the entire image, even if parts of it appear outside the axes limits. Parts of the image might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the image that is larger than the original plot.

Identifiers

`Type` — Type of graphics object

`'image'`

Type of graphics object, returned as the string `'image'`.

`Tag` — Tag to associate with image

`''` (default) | string

Tag to associate with the image, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with image

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the image object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent of image

axes object | group object | transform object

Parent of image, specified as an axes, group, or transform object.

`Children` — Children of image

empty `GraphicsPlaceholder` array

The image has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of image object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The image object handle is always visible.
- `'off'` — The image object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The image object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the image at the command-line, but allows callback functions to access it.

If the image object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle

- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the image. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The image object — You can access properties of the image object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the image. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the image when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the image.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The Visible property must be set to `'on'`. The HitTest property determines if the image responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the image passes the click to the object below it in the current view of the figure window. The HitTest property of the image has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the image. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the image that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the image object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the image is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the image tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the image. Setting the `CreateFcn` property on an existing image has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during image creation. MATLAB executes the callback after creating the image and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The image object — You can access properties of the image object from within the callback function. You also can access the image object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the image. MATLAB executes the callback before destroying the image so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The image object — You can access properties of the image object from within the callback function. You also can access the image object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of image

`'off'` (default) | `'on'`

Deletion status of image, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the image begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the image no longer exists.

Check the value of the BeingDeleted property to verify that the image is not about to be deleted before querying or modifying it.

See Also

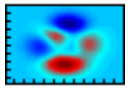
`image` | `imagesc`

More About

- "Access Property Values"
- "Graphics Object Properties"

imagesc

Scale data and display image object



Syntax

```
imagesc(C)
imagesc(x,y,C)
imagesc(...,clims)
imagesc('PropertyName',PropertyValue,...)
h = imagesc(...)
```

Description

The `imagesc` function scales image data to the full range of the current colormap and displays the image. (See “Examples” on page 1-3827 for an illustration.)

`imagesc(C)` displays `C` as an image. Each element of `C` corresponds to a rectangular area in the image. The values of the elements of `C` are indices into the current colormap that determine the color of each patch.

`imagesc(x,y,C)` displays `C` as an image and specifies the bounds of the *x*- and *y*-axis with vectors `x` and `y`. If `x(1) > x(2)` or `y(1) > y(2)`, the image is flipped left-right or up-down, respectively. If `x` and `y` are scalars, the image is translated to the specified location `(x,y)` such that the upper left corner of the image starts at `(x,y)`.

`imagesc(...,clims)` normalizes the values in `C` to the range specified by `clims` and displays `C` as an image. `clims` is a two-element vector that limits the range of data values in `C`. These values map to the full range of values in the current colormap.

`imagesc('PropertyName',PropertyValue,...)` is the low-level syntax of the `imagesc` function. It specifies only property name/property value pairs as input arguments. See Image Properties for a list of the property names and their values.

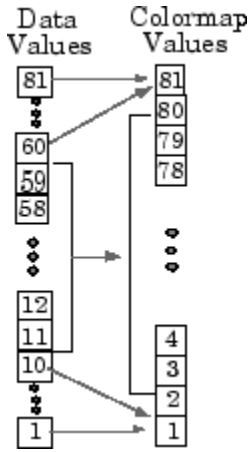
`h = imagesc(...)` returns the handle for an image graphics object.

Examples

You can expand midrange color resolution by mapping low values to the first color and high values to the last color in the colormap by specifying color value limits (`clims`). If the size of the current colormap is 81-by-3, the statements

```
clims = [ 10 60 ]
imagesc(C,clims)
```

map the data values in C to the colormap as shown in this illustration and the code that follows:



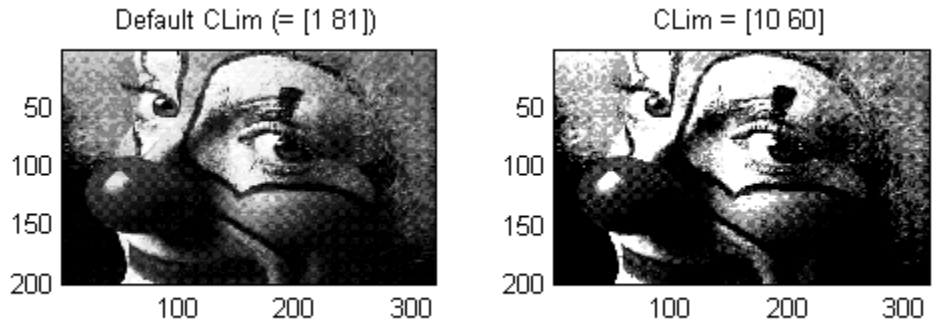
In this example, the left image maps to the `gray` colormap using the statements

```
load clown
figure
subplot(1,2,1)
imagesc(X)
```

```
colormap(gray)
axis image
title('Default CLim (= [1 81])')
```

The right image has values between 10 and 60 scaled to the full range of the gray colormap using the statements

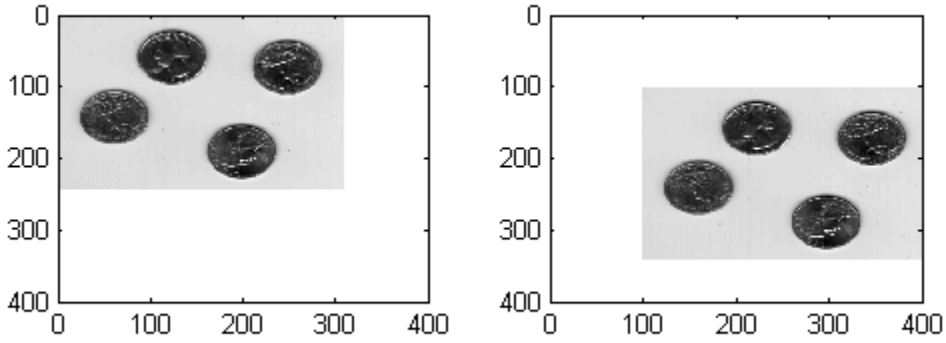
```
subplot(1,2,2)
clims = [10 60];
imagesc(X,clims)
colormap(gray)
axis image
title('CLim = [10 60]')
```



This example shows how to shift the image starting from origin to a position (100, 100),

```
i = imread('eight.tif');
figure; subplot(2,2,1); imagesc(i);
axis([0 400 0 400]);
colormap(gray);
subplot(2,2,2); imagesc(100,100,i);
axis([0 400 0 400]);
colormap(gray);
```

The figure output is as:



The top right corner of the image is now starting from (100,100) instead of the origin.

Passing vector values with the image scales the image to the specified size of 400–by–400.

```
figure; imagesc(1:400,1:400,i);
colormap(gray);
```

More About

Tips

`x` and `y` do not affect the elements in `C`; they only affect the annotation of the axes. If `length(x) > 2` or `length(y) > 2`, `imagesc` ignores all except the first and last elements of the respective vector.

`imagesc` creates an image with CDataMapping set to scaled, and sets the axes CLim property to the value passed in `clims`.

You cannot interactively pan or zoom outside the *x*-limits or *y*-limits of an image.

By default, `imagesc` plots the *y*-axis from lowest to highest value, top to bottom. To reverse this, type `set(gca,'YDir','normal')`. This will reverse both the *y*-axis and the image.

See Also

Functions

`colorbar` | `colormap` | `image` | `imfinfo` | `imread` | `imshow` | `imwrite` | `pcolor` | `surf` | `surface`

Properties

Image Properties

Introduced before R2006a

imapprox

Approximate indexed image by reducing number of colors

Syntax

```
[Y,newmap] = imapprox(X,map,n)
[Y,newmap] = imapprox(X,map,tol)
Y = imapprox(X,map,newmap)
Y = imapprox(...,dither_option)
```

Description

`[Y,newmap] = imapprox(X,map,n)` approximates the colors in the indexed image `X` and associated colormap `map` by using minimum variance quantization. `imapprox` returns the indexed image `Y` with colormap `newmap`, which has at most `n` colors.

`[Y,newmap] = imapprox(X,map,tol)` approximates the colors in `X` and `map` through uniform quantization. `newmap` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0 and 1.0.

`Y = imapprox(X,map,newmap)` approximates the colors in `map` by using colormap mapping to find the colors in `newmap` that best match the colors in `map`.

`Y = imapprox(...,dither_option)` enables or disables dithering. `dither_option` is a string that can have one of these values.

Value	Description
`{'dither'}`(default)	Dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.
`'nodither'`	Maps each color in the original image to the closest color in the new map. No dithering is performed.

Class Support

The input image `X` can be of class `uint8`, `uint16`, or `double`. The output image `Y` is of class `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `Y` is of class `double`.

Examples

Load an indexed image of a mandrill's face. Display image `X` using its associated colormap, `map`, which has 220 colors.

```
load mandrill
figure('color','k')
image(X)
colormap(map)
size(map)         % See that the color map has 220 entries

ans =
   220     3

axis off          % Remove axis ticks and numbers
axis image        % Set aspect ratio to obtain square pixels
```

Reduce the number of colors in the indexed image from 220 to only 16 colors by producing a new image, `Y`, and its associated colormap, `newmap`:

```
figure('color','k')
[Y, newmap] = imapprox(X, map, 16);
size(newmap)   % See that the new color map has 16 entries

ans =
    16     3

image(Y)
colormap(newmap)
axis off       % Remove axis ticks and numbers
axis image     % Set aspect ratio to obtain square pixels
```

More About

Algorithms

`imapprox` uses `rgb2ind` to create a new colormap that uses fewer colors.

See Also

`cmunique` | `dither` | `rgb2ind`

imfinfo

Information about graphics file

Syntax

```
info = imfinfo(filename)
info = imfinfo(filename,fmt)
info = imfinfo(URL)
```

Description

`info = imfinfo(filename)` returns a structure whose fields contain information about an image in a graphics file, `filename`.

The format of the file is inferred from its contents.

- If `filename` is a TIFF, HDF, ICO, GIF, or CUR file containing more than one image, then `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file.

`info = imfinfo(filename,fmt)` additionally looks for a file named `filename.fmt`, if MATLAB cannot find a file named `filename`.

`info = imfinfo(URL)` returns information about the image at the specified internet resource, `URL`.

Examples

Return Information About Graphics File

Find information about the example image, `ngc6543a.jpg`.

```
info = imfinfo('ngc6543a.jpg')
```

```
info =
```

```
           Filename: 'matlabroot\toolbox\matlab\demos\ngc6543a.jpg'
        FileModDate: '01-Oct-1996 16:19:44'
           FileSize: 27387
             Format: 'jpg'
      FormatVersion: ''
              Width: 600
             Height: 650
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {'CREATOR: XV Version 3.00b  Rev: 6/15/94  Quality = 75, Smoothing
'}
```

Input Arguments

`filename` — Name of graphics file

string

Name of graphics file, specified as a string.

Example: `'myImage.jpg'`

Example: `'C:\myFolder\myImage.jpg'`

Data Types: `char`

`fmt` — Image format

string

Image format, specified as a string. The possible values for `fmt` are contained in the MATLAB file format registry. To view of list of these formats, run the `imformats` command.

Example: `'gif'`

Data Types: `char`

`URL` — Image location

string

Image location, specified as a string. `URL` must include the protocol type (e.g., `http://`).

Data Types: `char`

Output Arguments

`info` — Information about graphics file

structure array

Information about the graphics file, returned as a structure array. The set of fields in `info` depends on the individual file and its format. This table lists the nine fields that always appear, and describes their values.

Field Name	Description	Value
`Filename`	Name of the file or the internet URL specified. If the file is not in the current folder, the string contains the full path name of the file.	string
`FileModDate`	Date when the file was last modified.	string
`FileSize`	Size of the file, in bytes.	integer
`Format`	File format, as specified by *`fmt`*. For formats with more than one possible extension (for example, JPEG and TIFF files), `imfinfo` returns the first variant in the file format registry.	string
`FormatVersion`	File format version.	string or number
`Width`	Image width, in pixels.	integer
`Height`	Image height, in pixels.	integer
`BitDepth`	Number of bits per pixel.	integer
`ColorType`	Image type. `ColorType` includes, but is not limited to, `'truecolor'` for a truecolor (RGB) image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image.	string

Additional fields returned by some file formats:

- **JPEG and TIFF only** — If `filename` contains Exchangeable Image File Format (EXIF) tags, then `info` might also contain `'DigitalCamera'` or `'GPSInfo'` (global positioning system information) fields.

- **GIF only** — `imfinfo` returns the value of the `'DelayTime'` field in hundredths of seconds.
- **JPEG2000 only** — The `info` structure contains an m-by-3 cell array, `'ChannelDefinition'`. The first column of `'ChannelDefinition'` reports a channel position as it exists in the file. The second column reports the type of channel, and the third column reports the channel mapping.

See Also

`imformats` | `imread` | `imwrite`

Introduced before R2006a

imformats

Manage image file format registry

Syntax

```
imformats

formatStruct = imformats(fmt)

registry = imformats
registry = imformats(formatStruct)
registry = imformats('add',formatStruct)
registry = imformats('remove',fmt)
registry = imformats('update',fmt,formatStruct)
registry = imformats('factory')
```

Description

`imformats` displays a table of information listing all the values in the MATLAB file format registry. This registry determines which file formats the `imfinfo`, `imread`, and `imwrite` functions support.

`formatStruct = imformats(fmt)` searches the known formats in the MATLAB file format registry for the format associated with the file name extension specified by `fmt`. If found, `formatStruct` is a structure containing the characteristics and function names associated with the format. Otherwise, `formatStruct` is an empty structure.

`registry = imformats` returns a structure array, `registry`, containing all the values in the MATLAB file format registry.

`registry = imformats(formatStruct)` sets the MATLAB file format registry for the current MATLAB session to the values in `formatStruct`. The output structure, `registry`, contains the new registry settings. Use this syntax to replace image file format support.

Incorrect use of `imformats` to specify values in the MATLAB file format registry can result in the inability to load any image files. To return the file format registry to a working state, use `imformats` with the `'factory'` input.

`registry = imformats('add',formatStruct)` adds the values in `formatStruct` to the file format registry. Use this syntax to add image file format support.

`registry = imformats('remove',fmt)` removes the format with the extension specified by `fmt` from the file format registry. Use this syntax to remove image file format support.

`registry = imformats('update',fmt,formatStruct)` changes the format registry values for the format with extension `fmt` to have the values specified by `formatStruct`.

`registry = imformats('factory')` resets the MATLAB file format registry to the default format registry values. This removes any user-specified settings.

Examples

Determine if File Format Exists in Registry

Determine if the file format associated with the `.bmp` file extension is in the image file format registry.

```
formatStruct = imformats('bmp')
```

```
formatStruct =

            ext: {'bmp'}
            isa: @isbmp
           info: @imbmpinfo
           read: @readbmp
          write: @writebmp
          alpha: 0
    description: 'Windows Bitmap'
```

`formatStruct` is a non-empty structure, so the BMP file format is in the registry.

Add, Update, or Remove File Format from Registry

Add a hypothetical file format, ABC, to the image file format registry. Update, and then remove the format.

Create a structure with seven fields, defining values for the new format.

```
formatStruct = struct('ext','abc','isa',@isabc,...
    'info',@abcinfo,'read',@readabc,'write','',...
    'alpha',0,'description','My ABC Format')
```

```
formatStruct =

            ext: 'abc'
            isa: @isabc
           info: @abcinfo
           read: @readabc
          write: ''
          alpha: 0
    description: 'My ABC Format'
```

`formatStruct` is a 1-by-1 structure with seven fields. In this example, the `write` field is empty.

Add the new format to the file format registry.

```
registry = imformats('add',formatStruct);
```

Redefine the format associated with the extension, `abc`, by adding a value for the `write` field. Then, update the registry value for the format.

```
formatStruct2 = struct('ext','abc','isa',@isabc,...
    'info',@abcinfo,'read',@readabc,'write',@writeabc,...
    'alpha',0,'description','My ABC Format');

registry = imformats('update','abc',formatStruct2);
```

Remove the format with the extension, `abc`, from the file format registry.

```
registry = imformats('remove','abc');
```

Input Arguments

`formatStruct` — File format registry values

structure array

File format registry values, specified as a structure array with the following 7 fields.

Field	Description	Value
`ext`	File name extensions that are valid for this format.	Cell array of strings
`isa`	Name of the function that determines if a file is of a certain format.	String or function handle
`info`	Name of the function that reads information about a file.	String or function handle
`read`	Name of the function that reads image data in a file.	String or function handle
`write`	Name of the function that writes MATLAB data to a file.	String or function handle
`alpha`	Presence or absence of an alpha channel.	`1` if the format has an alpha channel; otherwise it is `0`.
`description`	Text description of the file format.	String

The values for the `isa`, `info`, `read`, and `write` fields must be either functions on the MATLAB search path or function handles.

Data Types: `struct`

`fmt` — File format extension

string

File format extension, specified as a string.

Example: `'jpg'`

Data Types: `char`

Output Arguments

`registry` — File format registry

structure array

File format registry, returned as a structure array with the following fields.

Field	Description	Value
`ext`	File name extensions that are valid for this format.	Cell array of strings
`isa`	Name of the function that determines if a file is of a certain format.	String or function handle
`info`	Name of the function that reads information about a file.	String or function handle
`read`	Name of the function that reads image data in a file.	String or function handle
`write`	Name of the function that writes MATLAB data to a file.	String or function handle
`alpha`	Presence or absence of an alpha channel.	`1` if the format has an alpha channel; otherwise it is `0`.
`description`	Text description of the file format.	String

Note: Use the `imread`, `imwrite`, and `imfinfo` functions to read, write, or get information about an image file when the file format is in the format registry. Do not directly invoke the functions returned in the fields of the `registry` structure array.

More About

Tips

- Changes to the format registry do not persist between MATLAB sessions. To have a format always available when you start MATLAB, add the appropriate `imformats` command to the MATLAB startup file, `startup.m`, located in `$MATLAB/toolbox/local` on UNIX systems, or `$MATLAB\toolbox\local` on Windows systems.
- "What Is the MATLAB Search Path?"

See Also

`imfinfo` | `imread` | `imwrite` | `path`

Introduced before R2006a

import

Add package or class to current import list

Syntax

```
import package_name.class_name
import package_name.function_name
import package_name.*
import package_name.class_name1 package_name.class_name2...
import package_name1.* package_name2.*...
L = import
import
```

Description

`import package_name.class_name` adds the fully qualified class name to the current import list.

`import package_name.function_name` adds the specified package-based function.

`import package_name.*` adds the specified package name. Note that *`package_name`* must be followed by `.*`.

`import package_name.class_name1 package_name.class_name2...` adds multiple fully qualified class names.

`import package_name1.* package_name2.*...` adds multiple package names.

`L = import` with no input arguments returns a cell array of strings containing the current import list, without adding to it.

`import` with no input arguments displays the current import list, without adding to it.

The `import` function allows your code to refer to an imported class or function using fewer or no package prefixes.

The `import` function only affects the import list of the function within which it is used. When invoked at the command prompt, `import` uses the import list for the MATLAB

command environment. If `import` is used in a script invoked from a function, it affects the import list of the function. If `import` is used in a script that is invoked from the command prompt, it affects the import list for the command environment.

The import list of a function is persistent across calls to that function and is only cleared when the function is cleared.

To clear the current import list, use the following command.

```
clear import
```

This command may only be invoked at the command prompt. Attempting to use `clear import` within a function results in an error.

Importing MATLAB Packages and Classes

You can `import` packages and classes into a MATLAB workspace (from the command line or in a function definition). For example:

```
import packagename.*
```

imports all classes and package functions so that you can reference those classes and functions by their simple names, without the package qualifier.

You can import just a single class from a package:

```
import packagename.ClassName
import Classname
```

You must still use the class name to call static methods:

```
ClassName.staticMethod()
```

For more information on how `import` works with MATLAB classes and packages, see "Importing Classes".

Limitations

- `import` cannot load a Java JAR package created by the MATLAB Compiler SDK™ product.

Examples

To add the `containers.Map` class to the current import list:

```
import containers.Map
myMap = Map('KeyType', 'char', 'ValueType', 'double');
```

To import two Java packages:

```
import java.util.Enumeration java.lang.String
s = String('hello'); % Create java.lang.String object
methods Enumeration % List java.util.Enumeration methods
```

To add the `java.awt` package:

```
import java.awt.*
f = Frame; % Create java.awt.Frame object
```

This example uses `import` in a function to call members of a .NET class in the `System.Drawing` namespace. Create the `getPrinterInfo` function:

```
function ptr = getPrinterInfo
import System.Drawing.Printing.*;
ptr = PrinterSettings;
end
```

To call the function, type:

```
NET.addAssembly('System.Drawing');
printer = getPrinterInfo;
```

For more information about using `import` with .NET assemblies, see "Use import in MATLAB Functions".

More About

Tips

The `import` function allows your code to refer to an imported class by class name only, rather than with the fully qualified class name. `import` is particularly useful in streamlining calls to constructors, where most references to Java classes occur.

If you use the `import` function in a control statement, for example, `if` or `switch`, or in a function, MATLAB limits the scope of the variables to that block of code. If you use the variables outside the function or control block, MATLAB displays an error message.

See Also

`clear` | `load` | `importdata`

Introduced before R2006a

importdata

Load data from file

Syntax

```
A = importdata(filename)
A = importdata('-pastespecial')
A = importdata(___,delimiterIn)
A = importdata(___,delimiterIn,headerlinesIn)
[A,delimiterOut,headerlinesOut] = importdata(___)
```

Description

`A = importdata(filename)` loads data into array `A`.

`A = importdata('-pastespecial')` loads data from the system clipboard rather than from a file.

`A = importdata(___,delimiterIn)` interprets `delimiterIn` as the column separator in ASCII file, `filename`, or the clipboard data. You can use `delimiterIn` with any of the input arguments in the above syntaxes.

`A = importdata(___,delimiterIn,headerlinesIn)` loads data from ASCII file, `filename`, or the clipboard, reading numeric data starting from line `headerlinesIn+1`.

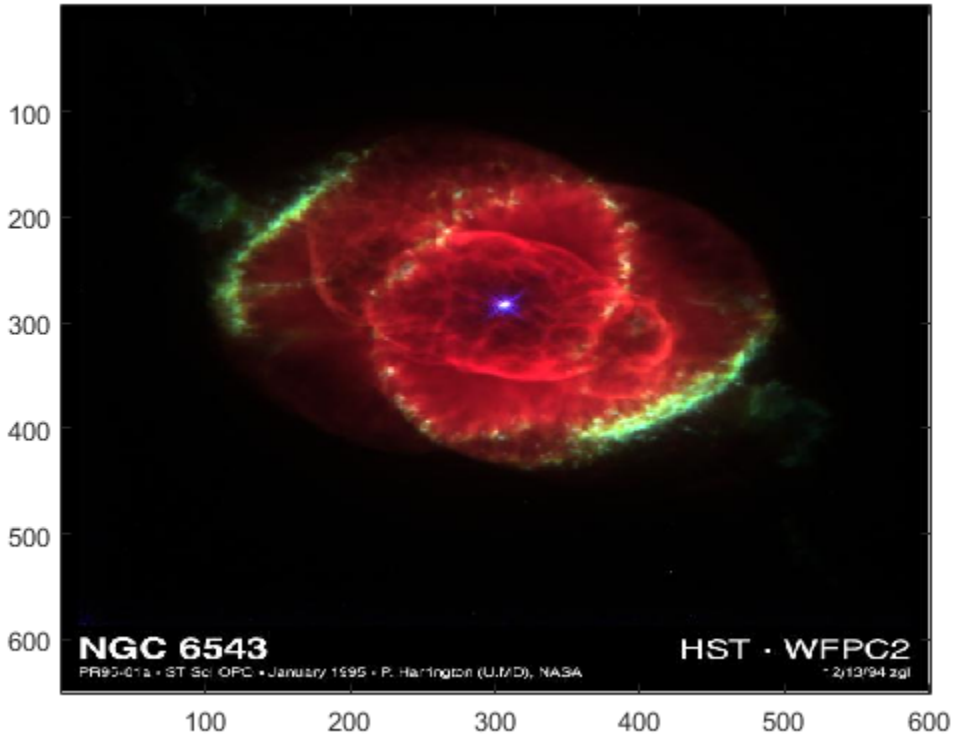
`[A,delimiterOut,headerlinesOut] = importdata(___)` additionally returns the detected delimiter character for the input ASCII file in `delimiterOut` and the detected number of header lines in `headerlinesOut`, using any of the input arguments in the previous syntaxes.

Examples

Import and Display an Image

Import and display the sample image, `ngc6543a.jpg`.

```
A = importdata('ngc6543a.jpg');
image(A)
```



The output, `A`, is class `uint8` because the helper function, `imread`, returns empty results for `colormap` and `alpha`.

Import a Text File and Specify Delimiter and Column Header

Using a text editor, create a space-delimited ASCII file with column headers called `myfile01.txt`.

```
Day1  Day2  Day3  Day4  Day5  Day6  Day7
95.01 76.21 61.54 40.57  5.79 20.28  1.53
23.11 45.65 79.19 93.55 35.29 19.87 74.68
```

```
60.68  1.85 92.18 91.69 81.32 60.38 44.51
48.60 82.14 73.82 41.03  0.99 27.22 93.18
89.13 44.47 17.63 89.36 13.89 19.88 46.60
```

Import the file, specifying the space delimiter and the single column header.

```
filename = 'myfile01.txt';
delimiterIn = ' ';
headerlinesIn = 1;
A = importdata(filename,delimiterIn,headerlinesIn);
```

View columns 3 and 5.

```
for k = [3, 5]
   disp(A.colheaders{1, k})
   disp(A.data(:, k))
   disp(' ')
end
```

```
Day3
   61.5400
   79.1900
   92.1800
   73.8200
   17.6300


Day5
    5.7900
   35.2900
   81.3200
    0.9900
   13.8900
```

Import a Text File and Return Detected Delimiter

Using a text editor, create a comma-delimited ASCII file called `myfile02.txt`.

```
1,2,3
4,5,6
7,8,9
```

Import the file, and display the output data and detected delimiter character.

```
filename = 'myfile02.txt';
[A,delimiterOut]=importdata(filename)
```

```
A =

     1     2     3
     4     5     6
     7     8     9


delimiterOut =

,
```

Import Data from Clipboard

Copy the following lines to the clipboard. Select the text, right-click, and then select **Copy**.

```
1,2,3
4,5,6
7,8,9
```

Import the clipboard data into MATLAB by typing the following.

```
A = importdata('-pastespecial')
```

```
A =

     1     2     3
     4     5     6
     7     8     9
```

Input Arguments

`filename` — Name and extension of file to import

string

Name and extension of the file to import, specified as a string. If `importdata` recognizes the file extension, it calls the MATLAB helper function designed to import the associated file format (such as `load` for MAT-files or `xlsread` for spreadsheets). Otherwise, `importdata` interprets the file as a delimited ASCII file.

For ASCII files and spreadsheets, `importdata` expects to find numeric data in a rectangular form (that is, like a matrix). Text headers can appear above or to the left of the numeric data, as follows:

- Column headers or file description text at the top of the file, above the numeric data.
- Row headers to the left of the numeric data.

Example: `'myFile.jpg'`

Data Types: `char`

`delimiterIn` — Column separator character
string

Column separator character, specified as a string. The default character is interpreted from the file. Use `'\t'` for tab.

Example: `','`

Example: `' '`

Data Types: `char`

`headerlinesIn` — Number of text header lines in ASCII file
nonnegative scalar integer

Number of text header lines in the ASCII file, specified as a nonnegative scalar integer. If you do not specify `headerlinesIn`, the `importdata` function detects this value in the file.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Output Arguments

`A` — Data from the file
matrix | multidimensional array | scalar structure array

Data from the file, returned as a matrix, multidimensional array, or scalar structure array, depending on the characteristics of the file. Based on the file format of the input file, `importdata` calls a helper function to read the data. When the helper function returns more than one nonempty output, `importdata` combines the outputs into a `struct` array.

This table lists the file formats associated with helper functions that can return more than one output, and the possible fields in the structure array, `A`.

File Format	Possible Fields	Class
MAT-files	One field for each variable	Associated with each variable.

ASCII files and Spreadsheets	`data`
	`textdata`
	`colheaders`
	`rowheaders`
		For ASCII files, `data` contains a `double` array. Other fields contain `cell` arrays of strings. `textdata` includes row and column headers.
		For spreadsheets, each field contains a `struct`, with one field for each worksheet.

Images	`cdata`
	`colormap`
	`alpha`
		See `imread`.

Audio files	`data` `fs`	See `audioread`.

The MATLAB helper functions for most other supported file formats return one output. For more information about the class of each output, see the functions listed in "Supported File Formats for Import and Export".

If the ASCII file or spreadsheet contains either column or row headers, but not both, `importdata` returns a `colheaders` or `rowheaders` field in the output structure, where:

- `colheaders` contains only the last line of column header text. `importdata` stores all text in the `textdata` field.
- `rowheaders` is created only when the file or worksheet contains a single column of row headers.

`delimiterOut` — Detected column separator in the input ASCII file
string

Detected column separator in the input ASCII file, returned as a string.

`headerlinesOut` — Detected number of text header lines in the input ASCII file
integer

Detected number of text header lines in the input ASCII file, returned as an integer.

More About

Tips

- To import ASCII files with nonnumeric characters outside of column or row headers, including columns of character data or formatted dates or times, use `textscan` instead of `importdata`.

- "Supported File Formats for Import and Export"
- "Ways to Import Text Files"
- "Ways to Import Spreadsheets"
- "Import or Export a Sequence of Files"

See Also

`imread` | `load` | `readtable` | `save` | `textscan` | `uiimport` | `xlsread`

Introduced before R2006a

imread

Read image from graphics file

Syntax

```
A = imread(filename)
A = imread(filename,fmt)
A = imread( ___ ,idx)
A = imread( ___ ,Name,Value)
[A,map] = imread( ___ )
[A,map,transparency] = imread( ___ )
```

Description

`A = imread(filename)` reads the image from the file specified by `filename`, inferring the format of the file from its contents. If `filename` is a multi-image file, then `imread` reads the first image in the file.

`A = imread(filename,fmt)` additionally specifies the format of the file with the standard file extension indicated by `fmt`. If `imread` cannot find a file with the name specified by `filename`, it looks for a file named *`filename.fmt`*.

`A = imread( ___ ,idx)` reads the specified image or images from a multi-image file. This syntax applies only to GIF, CUR, ICO, and HDF4 files. You must specify a filename input, and you can optionally specify fmt.

`A = imread( ___ ,Name,Value)` specifies format-specific options using one or more name-value pair arguments, in addition to any of the input arguments in the previous syntaxes.

`[A,map] = imread( ___ )` reads the indexed image in `filename` into `A` and reads its associated colormap into `map`. Colormap values in the image file are automatically rescaled into the range `[0,1]`.

`[A,map,transparency] = imread( ___ )` additionally returns the image transparency. This syntax applies only to PNG, CUR, and ICO files. For PNG files,

`transparency` is the alpha channel, if one is present. For CUR and ICO files, it is the AND (opacity) mask.

Examples

Read and Display Image

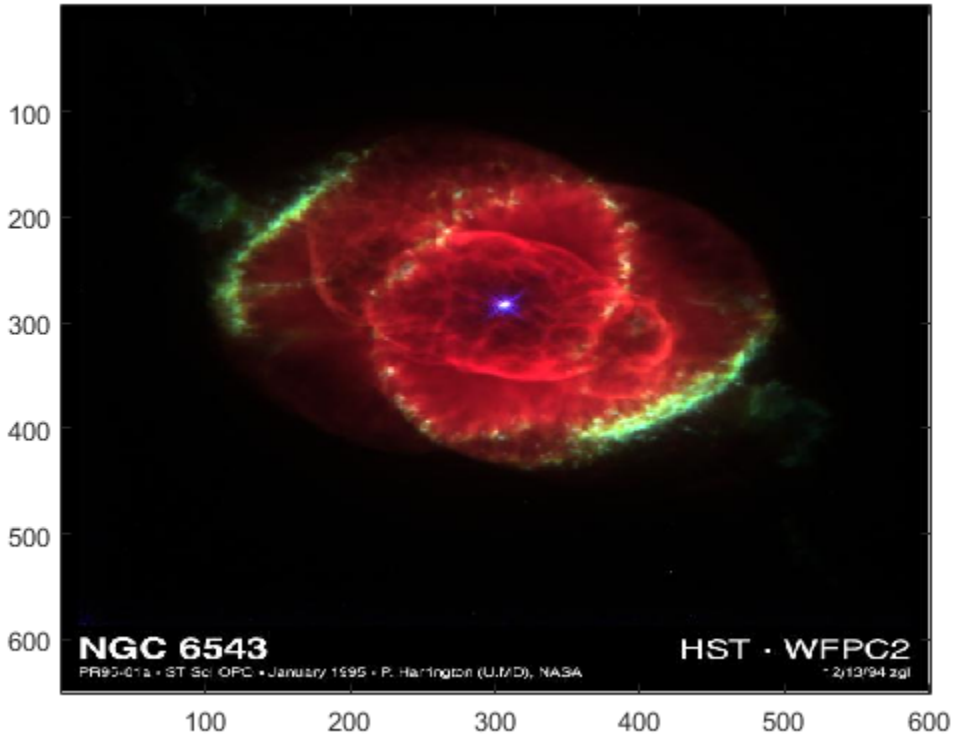
Read a sample image.

```
A = imread('ngc6543a.jpg');
```

`imread` returns a 650-by-600-by-3 array, `A`.

Display the image.

```
image(A)
```



Convert Indexed Image to RGB

Read an image and convert it to an RGB image.

Read the first image in the sample indexed image file, `corn.tif`.

```
[X,map] = imread('corn.tif');
```

`X` is a 415-by-312 array of type `uint8`.

Verify that the colormap, `map`, is not empty, and convert the data in `X` to RGB.

```
if ~isempty(map)
    Im = ind2rgb(X,map);
```

```
end
```

View the size and class of X.

```
whos Im
```

```
  Name        Size                 Bytes  Class     Attributes

  Im        415x312x3            3107520  double
```

X is now a 415-by-312-by-3 array of type `double`.

Read Specific Image in Multipage TIFF File

Read the third image in the sample file, `corn.tif`.

```
[X,map] = imread('corn.tif',3);
```

Return Alpha Channel of PNG Image

Return the alpha channel of the sample image, `peppers.png`.

```
[X,map,alpha] = imread('peppers.png');
alpha
```

```
alpha =

     []
```

No alpha channel is present, so `alpha` is empty.

Read Specified Region of TIFF Image

Read a specific region of pixels of the sample image, `corn.tif`.

Specify the `'PixelRegion'` parameter with a cell array of vectors indicating the boundaries of the region to read. The first vector specifies the range of rows to read, and the second vector specifies the range of columns to read.

```
A = imread('corn.tif','PixelRegion',{[1,2],[2,5]});
```

`imread` reads the image data in rows 1-2 and columns 2-5 from `corn.tif` and returns the 2-by-4 array, `A`.

Input Arguments

`filename` — File name

string

File name, specified as a string. If the file is not in the current folder or in a folder on the MATLAB path, specify the full path name.

`filename` also can be an internet URL specifying an image location. The URL must include the protocol type (for example, `http://`).

For information on the bit depths, compression schemes, and color spaces supported for each file type, see "Algorithms" on page 1-3866.

Example: `'myFile.jpg'`

Example: `'http://www.mathworks.com/myImage.gif'`

Data Types: `char`

`fmt` — Image format

string

Image format, specified as a string indicating the standard file extension. Call `imformats` to see a list of supported formats and their file extensions.

Example: `'png'`

Data Types: `char`

`idx` — Image to read

integer scalar | vector of integers

Image to read, specified as an integer scalar or, for GIF files, a vector of integers. For example, if `idx` is 3, then `imread` returns the third image in the file. For a GIF file, if `idx` is `1:5`, then `imread` returns only the first five frames. The `idx` argument is supported only for multi-image GIF, CUR, ICO, and HDF4 files.

When reading multiple frames from the same GIF file, specify `idx` as a vector of frames or use the `'Frames','all'` name-value pair argument. Because of the way that GIF

files are structured, these syntaxes provide faster performance compared to calling `imread` in a loop.

For HDF4 files, `idx` corresponds to the reference number of the image to read. Reference numbers do not necessarily correspond to the order of the images in the file. You can use `imfinfo` to match image order with reference number.

Example: `3`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Index',5` reads the fifth image of a TIFF file.

GIF Files

`'Frames'` — Frame to read

1 (default) | positive integer | vector of integers | `'all'`

Frames to read, specified as the comma-separated pair consisting of `'Frames'` and a positive integer, a vector of integers, or the string `'all'`. For example, if you specify the value 3, `imread` reads the third frame in the file. If you specify `'all'`, then `imread` reads all frames and returns them in the order in which they appear in the file.

Example: `'frames',5`

JPEG 2000 Files

`'PixelRegion'` — Subimage to read

cell array in the form `{rows,cols}`

Subimage to read, specified as the comma-separated pair consisting of `'PixelRegion'` and a cell array of the form `{rows,cols}`. The `rows` input specifies the range of rows to read. The `cols` input specifies the range of columns to read. Both `rows` and `cols` must

be two-element vectors containing 1-based indices. For example, `'PixelRegion',{[1 2],[3 4]}` reads the subimage bounded by rows 1 and 2 and columns 3 and 4 in the image data. If the `'ReductionLevel'` value is greater than 0, then `rows` and `cols` are coordinates of the subimage.

Example: `'PixelRegion',{[1 100],[4 500]}`

`'ReductionLevel'` — Reduction of image resolution

0 (default) | nonnegative integer

Reduction of the image resolution, specified as the comma-separated pair consisting of `'ReductionLevel'` and a nonnegative integer. For reduction level L, the image resolution is reduced by a factor of 2^L. The reduction level is limited by the total number of decomposition levels as specified by the `'WaveletDecompositionLevels'` field in the output of the `imfinfo` function.

Example: `'ReductionLevel',5`

Data Types: `single` | `double`

`'V79Compatible'` — Compatibility with MATLAB 7.9 (R2009b) and earlier

`false` (default) | `true`

Compatibility with MATLAB 7.9 (R2009b) and earlier, specified as the comma-separated pair consisting of `'V79Compatible'` and either `true` or `false`. If you specify `true`, then the returned grayscale or RGB image is consistent with previous versions of `imread` (MATLAB 7.9 (R2009b) and earlier).

Example: `'V79Compatible',true`

Data Types: `logical`

PNG Files

`'BackgroundColor'` — Background color

`'none'` | integer | 3-element vector of integers

Background color, specified as `'none'`, an integer, or a three-element vector of integers. If `BackgroundColor` is `'none'`, then `imread` does not perform any compositing. Otherwise, `imread` blends transparent pixels with the background color.

- If the input image is indexed, then the value of `BackgroundColor` must be an integer in the range `[1,P]`, where `P` is the colormap length.

- If the input image is grayscale, then the value of `BackgroundColor` must be an integer in the range `[0,1]`.
- If the input image is RGB, then the value of `BackgroundColor` must be a three-element vector with values in the range `[0,1]`.

The default value for `BackgroundColor` depends on the presence of the `transparency` output argument and the image type:

- If you request the `transparency` output argument, then the default value of `BackgroundColor` is `'none'`.
- If you do not request the `transparency` output and the PNG file contains a background color chunk, then that color is the default value for `BackgroundColor`.
- If you do not request the `transparency` output and the file does not contain a background color chunk, then the default value for `BackgroundColor` is `1` for indexed images, `0` for grayscale images, and `[0 0 0]` for truecolor (RGB) images.

TIFF Files

`'Index'` — Image to read

1 (default) | positive integer

Image to read, specified as the comma-separated pair consisting of `'Index'` and a positive integer. For example, if the value of `Index` is 3, then `imread` reads the third image in the file.

Data Types: `single` | `double`

`'Info'` — Information about image

structure array

Information about the image, specified as the comma-separated pair consisting of `'Info'` and a structure array returned by the `imfinfo` function. Use the `Info` name-value pair argument to help `imread` locate the images in a multi-image TIFF file more quickly.

Data Types: `struct`

`'PixelRegion'` — Region boundary

cell array

Region boundary, specified as the comma-separated pair consisting of `'PixelRegion'` and a cell array of the form `{rows,cols}`. The `rows` input specifies the range of rows to read. The `cols` input specifies the range of columns to read. `rows` and `cols` must be either two-element or three-element vectors of 1-based indices. A two-element vector specifies the first and last rows or columns to read. For example, `'PixelRegion',{[1 2],[3 4]}` reads the region bounded by rows 1 and 2 and columns 3 and 4 in the image data.

A three-element vector must be in the form `[start increment stop]`, where `start` is the first row or column to read, `increment` is an incremental value, and `stop` is the last row or column to read. This syntax allows image downsampling. For example, `'PixelRegion',{[1 2 10],[4 3 12]}` reads the region bounded by rows 1 and 10 and columns 4 and 12, and samples data from every 2 pixels in the vertical direction, and every 3 pixels in the horizontal direction.

Example: `'PixelRegion',{[1 100],[4 500]}`

Data Types: `cell`

Output Arguments

A — Image data

array

Image data, returned as an array.

- If the file contains a grayscale image, then A is an `m`-by-`n` array.
- If the file contains an indexed image, then A is an `m`-by-`n` array of index values corresponding to the color at that index in `map`.
- If the file contains a truecolor image, then A is an `m`-by-`n`-by-3 array.
- If the file is a TIFF file containing color images that use the CMYK color space, then A is an `m`-by-`n`-by-4 array.

The class of `A` depends on the image format and the bit depth of the image data. For more information, see “Algorithms” on page 1-3866

map — Colormap

`m`-by-3 matrix

Colormap associated with the indexed image data in `A`, returned as an `m`-by-3 matrix of class `double`.

`transparency` — Transparency information

matrix

Transparency information, returned as a matrix. For PNG files, `transparency` is the alpha channel, if present. If no alpha channel is present, or if you specify the `'BackgroundColor'` name-value pair argument, then `transparency` is `[]`. For CUR and ICO files, `transparency` is the AND mask. For cursor files, this mask sometimes contains the only useful data.

More About

Bit Depth

Bit depth is the number of bits used to represent each image pixel.

Bit depth is calculated by multiplying the bits-per-sample with the samples-per-pixel. Thus, a format that uses 8 bits for each color component (or sample) and three samples per pixel has a bit depth of 24. Sometimes the sample size associated with a bit depth can be ambiguous. For example, does a 48-bit bit depth represent six 8-bit samples, four 12-bit samples, or three 16-bit samples? See “Algorithms” on page 1-3866 for sample size information to avoid this ambiguity.

Algorithms

For most image file formats, `imread` uses 8 or fewer bits per color plane to store image pixels. This table lists the class of the returned image array, `A`, for the bit depths used by the file formats.

Bit Depth in File	Class of Array Returned by `imread`
1 bit per pixel	`logical`
2 to 8 bits per color plane	`uint8`
9 to 16 bits per pixel	`uint16` (`BMP`, `JPEG`, `PNG`, and `TIFF`) For the 16-bit BMP packed format (5-6-5), MATLAB returns `uint8`

The following sections provide information about the support for specific formats, listed in alphabetical order by format name.

BMP — Windows Bitmap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	No Compression	RLE Compression	Output Class	Notes
1 bit	✓	–	`logical`	
4 bit	✓	✓	`uint8`	
8 bit	✓	✓	`uint8`	
16 bit	✓	–	`uint8`	1 sample/pixel
24 bit	✓	–	`uint8`	3 samples/pixel
32 bit	✓	–	`uint8`	3 samples/pixel (1 byte padding)

CUR — Cursor File

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	No Compression	Compression	Output Class
1 bit	✓	–	`logical`

Supported Bit Depths	No Compression	Compression	Output Class
4 bit	✓	–	`uint8`
8 bit	✓	–	`uint8`

Note By default, Microsoft Windows cursors are 32-by-32 pixels. Since MATLAB pointers must be 16-by-16, you might need to scale your image. If you have Image Processing Toolbox™, you can use the `imresize` function.

GIF — Graphics Interchange Format

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	No Compression	Compression	Output Class
1 bit	✓	–	`logical`
2 bit to 8 bit	✓	–	`uint8`

HDF4 — Hierarchical Data Format

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raster Image with colormap	Raster image without colormap	Output Class	Notes
8 bit	✓	✓	`uint8`	
24 bit	–	✓	`uint8`	3 samples/pixel

ICO — Icon File

See “CUR — Cursor File” on page 1-3867

JPEG — Joint Photographic Experts Group

`imread` reads any baseline JPEG image, as well as JPEG images with some commonly used extensions. For information on JPEG 2000 file support, see JPEG 2000.

Supported Bits per Sample	Lossy Compression	Lossless Compression	Output Class	Notes
8 bit	✓	✓	`uint8`	Grayscale or RGB
12 bit	✓	✓	`uint16`	Grayscale or RGB
16 bit	–	✓	`uint16`	Grayscale

JPEG 2000 — Joint Photographic Experts Group 2000

For information about JPEG files, see JPEG.

Note: Indexed JPEG 2000 images are not supported. Only JP2 compatible color spaces are supported for JP2/JPX files. By default, all image channels are returned in the order they are stored in the file.

Supported Bits per Sample	Lossy Compression	Lossless Compression	Output Class	Notes
1 bit	✓	✓	`logical`	Grayscale only
2 bit to 8 bit	✓	✓	`uint8` or `int8`	Grayscale or RGB
9 bit to 16 bit	✓	✓	`uint16` or `int16`	Grayscale or RGB

PBM — Portable Bitmap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raw Binary	ASCII (Plain) Encoded	Output Class
1 bit	✓	✓	`logical`

PCX — Windows Paintbrush

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Output Class	Notes
1 bit	`logical`	Grayscale only

Supported Bit Depths	Output Class	Notes
8 bit	`uint8`	Grayscale or indexed
24 bit	`uint8`	RGB Three 8-bit samples/pixel

PGM — Portable Graymap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raw Binary	ASCII (Plain) Encoded	Output Class	Notes
8 bit	✓	–	`uint8`	
16 bit	✓	–	`uint16`	
Arbitrary	–	✓	1-bit to 8-bit: `uint8` 9-bit to 16-bit: `uint16`	Values are scaled

PNG — Portable Network Graphics

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Output Class	Notes
1 bit	`logical`	Grayscale
2 bit	`uint8`	Grayscale
4 bit	`uint8`	Grayscale
8 bit	`uint8`	Grayscale or Indexed
16 bit	`uint16`	Grayscale or Indexed
24 bit	`uint8`	RGB Three 8-bit samples/pixel.
48 bit	`uint16`	RGB Three 16-bit samples/pixel.

PPM — Portable Pixmap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16 bit	✓	–	`uint8`
Arbitrary	–	✓	

RAS — Sun Raster

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Output Class	Notes
1 bit	`logical`	Bitmap
8 bit	`uint8`	Indexed
24 bit	`uint8`	RGB Three 8-bit samples/pixel
32 bit	`uint8`	RGB with Alpha Four 8-bit samples/pixel

TIFF — Tagged Image File Format

`imread` reads most images supported by the TIFF specification or LibTIFF. The `imread` function supports these TIFF capabilities:

- Any number of samples per pixel
- CCITT group 3 and 4 FAX, Packbits, JPEG, LZW, Deflate, ThunderScan compression, and uncompressed images
- Logical, grayscale, indexed color, truecolor and hyperspectral images
- RGB, CMYK, CIELAB, ICCLAB color spaces. If the color image uses the CMYK color space, `A` is an `m`-by-`n`-by-4 array. To determine which color space is used, use `imfinfo` to get information about the graphics file and look at the value of the `PhotometricInterpretation` field. If a file contains CIELAB color data, `imread` converts it to ICCLAB before bringing it into the MATLAB workspace. This conversion is necessary because 8-bit or 16-bit TIFF CIELAB-encoded values use a mixture of signed and unsigned data types that cannot be represented as a single MATLAB array.
- Data organized into tiles or scanlines

`imread` reads and converts TIFF images as follows:

- YCbCr images are converted into the RGB colorspace.
- All grayscale images are read as if black = `0`, white = largest value.
- 1-bit images are returned as class `logical`.
- CIELab images are converted into ICCLab colorspace.

For copyright information, open the `libtiffcopyright.txt` file.

XWD — X Window Dump

This table lists the supported bit depths, compression, and output classes for XWD files.

Supported Bit Depths	ZPixmaps	XYBitmaps	XYPixmaps	Output Class
1 bit	✓	–	✓	`logical`
8 bit	✓	–	–	`uint8`

- "Importing Images"

See Also

`fread` | `image` | `imfinfo` | `imformats` | `imwrite` | `ind2rgb`

Introduced before R2006a

imshow

Display image

Syntax

```
imshow(I)
imshow(X,map)
imshow(filename)
imshow(I,[low high])
imshow(___,Name,Value)

himage = imshow( ___ )
```

Description

`imshow(I)` displays image `I` in a Handle Graphics figure, where `I` is a grayscale, RGB (truecolor), or binary image. For binary images, `imshow` displays pixels with the value `0` (zero) as black and `1` as white. `imshow` optimizes figure, axes, and image object properties for image display.

`imshow(X,map)` displays the indexed image `X` with the colormap `map`. A colormap matrix can have any number of rows, but it must have exactly 3 columns. Each row is interpreted as a color, with the first element specifying the intensity of red light, the second green, and the third blue. Color intensity can be specified on the interval 0.0 to 1.0.

`imshow(filename)` displays the image stored in the graphics file specified by `filename`.

`imshow(I,[low high])` displays grayscale image `I`, specifying the display range as a two-element vector, `[low high]`. For more information, see the DisplayRange parameter.

`imshow(___,Name,Value)` displays an image, using name-value pairs to control aspects of the operation.

`himage = imshow( ___ )` returns the handle to the image object created by `imshow`.

Examples

Display Grayscale Image

Display a grayscale image by reading an RGB image into the workspace and converting it to a grayscale image.

Read RGB image into the workspace.

```
RGB = imread('peppers.png');
```

Convert the image to grayscale.

```
I = rgb2gray(RGB);
```

Display the grayscale image.

```
imshow(I)
```

Display Image from File

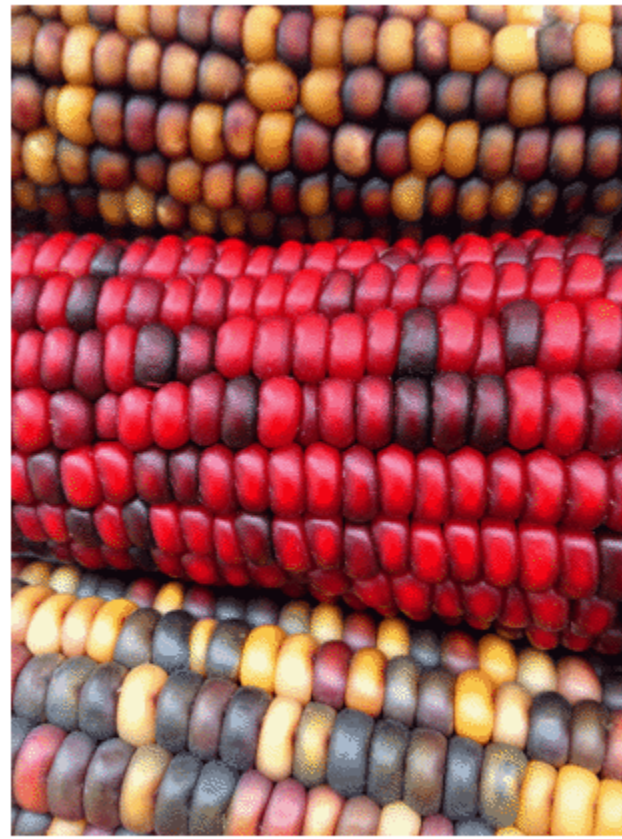
Display an image stored in a file.

```
imshow('peppers.png');
```

Display Indexed Image

Read a sample indexed image, `corn.tif`, into the workspace, and then display it.

```
[X,map] = imread('corn.tif');
imshow(X,map)
```



Input Arguments

`I` — Input image
scalar | vector | matrix | `m`-by-`n`-by-3 array

Input image, specified as a scalar, vector, or matrix representing a grayscale, RGB, or binary image. Multi-plane image inputs must be RGB images of size `m`-by-`n`-by-3. An RGB image can be `uint8`, `uint16`, `single`, or `double`. A grayscale image can be any numeric data type. A binary image is of class `logical`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`X` — Indexed image

2-D array of real numeric values

Indexed image, specified as a 2-D array of real numeric values. The values in X are indices into the colormap specified by map.

Data Types: `single` | `double` | `uint8` | `logical`

`map` — Colormap

m-by-3 array

Colormap, specified as an *m*-by-3 array of type `single` or `double` in the range `[0 1]`, or an *m*-by-3 array of type `uint8`. Each row specifies an RGB color value.

Data Types: `single` | `double` | `uint8`

`filename` — File name

string

File name, specified as a string. The image must be readable by `imread`. The `imshow` function displays the image, but does not store the image data in the MATLAB workspace. If the file contains multiple images, `imshow` displays the first image in the file.

Example: `imshow('peppers.png')`

Data Types: `char`

`[low high]` — Grayscale image display range

two-element vector

Grayscale image display range, specified as a two-element vector. For more information, see the `'DisplayRange'` name-value pair argument.

Example: `[50 250]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `imshow('board.tif','Border','tight')`

`'Border'` — Figure window border space

`'loose'` (default) | `'tight'`

Figure window border space, specified as the comma-separated pair consisting of `'Border'` and either `'tight'` or `'loose'`. When set to `'loose'`, the figure window includes space around the image in the figure. When set to `'tight'`, the figure window does not include any space around the image in the figure.

If the image is very small or if the figure contains other objects besides an image and its axes, `imshow` might use a border regardless of how this parameter is set.

Example: `imshow('board.tif','Border','tight')`

Data Types: `char`

`'Colormap'` — Colormap

m-by-3 matrix

Colormap, specified as the comma-separated pair consisting of `'Colormap'` and an `m`-by-3 matrix. `imshow` uses this to set the figure's `colormap` property. Use this parameter to view grayscale images in false color. If you specify an empty colormap (`[]`), then `imshow` ignores this parameter.

Example: `newmap = copper; imshow('board.tif','Colormap',newmap)`

Data Types: `double`

`'DisplayRange'` — Grayscale image display range

two-element vector

Grayscale image display range, specified as the comma-separated pair consisting of `'DisplayRange'` and a two-element vector of the form `[low high]`. The `imshow` function displays the value `low` (and any value less than `low`) as black, and it displays the value `high` (and any value greater than `high`) as white. Values between `low` and

high are displayed as intermediate shades of gray, using the default number of gray levels.

For uint8, 'DisplayRange' defaults to [0 255]. For all other data types it defaults to [min(I(:)) max(I(:))]. If you specify an empty matrix ([]), imshow uses the default.

Note: Including the parameter name is optional, except when the image is specified by a file name. The syntax imshow(I,[low high]) is equivalent to imshow(I,'DisplayRange',[low high]). If you call imshow with a file name, then you must specify the 'DisplayRange' parameter.

Example: h = imshow(I,'DisplayRange',[0 80]);

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

'InitialMagnification' — Initial magnification of image display

100 (default) | numeric scalar | 'fit'

Initial magnification of image display, specified as the comma-separated pair consisting of 'InitialMagnification' and a numeric scalar or the string, 'fit'. If set to 100, then imshow displays the image at 100% magnification (one screen pixel for each image pixel). If set to 'fit', then imshow scales the entire image to fit in the window.

Initially, imshow always displays the entire image. If the magnification value is so large that the image is too big to display on the screen, imshow warns and displays the image at the largest magnification that fits on the screen.

If the image is displayed in a figure with its 'WindowStyle' property set to 'docked', then imshow warns and displays the image at the largest magnification that fits in the figure.

Note: If you specify the axes position (using subplot or axes), imshow ignores any initial magnification you might have specified and defaults to the 'fit' behavior.

When you use imshow with the 'Reduce' parameter, the initial magnification must be 'fit'.

Example: h = imshow(I,'InitialMagnification','fit');

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

`'Parent'` — Parent axes of image object

handle

Parent axes of image object, specified as the comma-separated pair consisting of `'Parent'` and a handle. Use the `'Parent'` name-value argument to build a UI that gives you control of the figure and axes properties.

Data Types: `function_handle`

`'Reduce'` — Indicator for subsampling

`true` | `false` | `1` | `0`

Indicator for subsampling image, specified as the comma-separated pair consisting of `'Reduce'` and either `true`, `false`, `1`, or `0`. This argument is valid only when you use it with the name of a TIFF file. Use the `Reduce` argument to display overviews of very large images.

Data Types: `logical`

`'Xdata'` — X-axis limits of nondefault coordinate system

two-element vector

X-axis limits of nondefault coordinate system, specified as the comma-separated pair consisting of `'Xdata'` and a two-element vector. This argument establishes a nondefault spatial coordinate system by specifying the image `XData`. The value can have more than two elements, but `imshow` uses only the first and last elements.

Example: `'Xdata',[100 200]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'YData'` — Y-axis limits of nondefault coordinate system

two-element vector

Y-axis limits of nondefault coordinate system, specified as the comma-separated pair consisting of `'Ydata'` and a two-element vector. The value can have more than two elements, but `imshow` uses only the first and last elements.

Example: `'YData',[100 200]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`himage` — Image object

handle

Image object, specified as a handle.

More About

Tips

- If you have Image Processing Toolbox, you can use the Image Viewer app as an integrated environment for displaying images and performing common image processing tasks.
- If you have Image Processing Toolbox, you can use the `iptsetpref` function to set toolbox preferences that modify the behavior of `imshow`.
- The `imshow` function is not supported when you start MATLAB with the `-nojvm` option.

See Also

`image` | `imagesc` | `imfinfo` | `imread` | `imwrite` | `iptsetpref`

imwrite

Write image to graphics file

Syntax

```
imwrite(A,filename)
imwrite(A,map,filename)

imwrite( ___ ,fmt)

imwrite( ___ ,Name,Value)
```

Description

`imwrite(A,filename)` writes image data `A` to the file specified by `filename`, inferring the file format from the extension. `imwrite` creates the new file in your current folder. The bit depth of the output image depends on the data type of `A` and the file format. For most formats:

- If `A` is of data type `uint8`, then `imwrite` outputs 8-bit values.
- If `A` is of data type `uint16` and the output file format supports 16-bit data (JPEG, PNG, and TIFF), then `imwrite` outputs 16-bit values. If the output file format does not support 16-bit data, then `imwrite` returns an error.
- If `A` is a grayscale or RGB color image of data type `double` or `single`, then `imwrite` assumes that the dynamic range is [0,1] and automatically scales the data by 255 before writing it to the file as 8-bit values. If the data in `A` is `single`, convert `A` to `double` before writing to a GIF or TIFF file.
- If `A` is of data type `logical`, then `imwrite` assumes that the data is a binary image and writes it to the file with a bit depth of 1, if the format allows it. BMP, PNG, or TIFF formats accept binary images as input arrays.

If `A` contains indexed image data, you should additionally specify the `map` input argument.

`imwrite(A,map,filename)` writes the indexed image in `A` and its associated colormap, `map`, to the file specified by `filename`.

- If A is an indexed image of data type `double` or `single`, then `imwrite` converts the indices to zero-based indices by subtracting 1 from each element, and then writes the data as `uint8`. If the data in A is `single`, convert A to `double` before writing to a GIF or TIFF file.

`imwrite(___,fmt)` writes the image in the format specified by `fmt`, regardless of the file extension in `filename`. You can specify `fmt` after the input arguments in any of the previous syntaxes.

`imwrite(___,Name,Value)` specifies additional parameters for output GIF, HDF, JPEG, PBM, PGM, PNG, PPM, and TIFF files, using one or more name-value pair arguments. You can specify `Name,Value` after the input arguments in any of the previous syntaxes.

Examples

Write Grayscale Image to PNG

Write a 50-by-50 array of grayscale values to a PNG file in the current folder.

```
A = rand(50);
imwrite(A,'myGray.png')
```

Write Indexed Image Data to PNG

Write an indexed image array and its associated colormap to a PNG file.

Load sample image data from the file, `clown.mat`.

```
load clown.mat
```

The image array X and its associated colormap, `map`, are loaded into the MATLAB workspace.

Write the data to a new PNG file.

```
imwrite(X,map,'myclown.png')
```

`imwrite` creates the file, `myclown.png`, in your current folder.

View the new file by opening it outside of MATLAB.

Write Indexed Image with MATLAB Colormap

Write image data to a new PNG file with the built-in MATLAB colormap, `copper`.

Load sample image data from the file `clown.mat`.

```
load clown.mat
```

The image array `X` and its associated colormap, `map`, are loaded into the MATLAB workspace. `map` is a matrix of 81 RGB vectors.

Define a copper-tone colormap with 81 RGB vectors. Then, write the image data to a PNG file using the new colormap.

```
newmap = copper(81);
imwrite(X,newmap,'copperclown.png');
```

`imwrite` creates the file, `copperclown.png`, in your current folder.

View the new file by opening it outside of MATLAB.

Write Truecolor Image to JPEG

Create and write truecolor image data to a JPEG file.

Create a 49-by-49-by-3 array of random RGB values.

```
A = rand(49,49);
A(:,:,2) = rand(49,49);
A(:,:,3) = rand(49,49);
```

Write the image data to a JPEG file, specifying the output format using the string, `'jpg'`. Add a comment to the file using the `'Comment'` name-value pair argument.

```
imwrite(A,'newImage.jpg','jpg','Comment','My JPEG file')
```

View information about the new file.

```
imfinfo('newImage.jpg')
```

```
ans =

           Filename: 'S:\newImage.jpg'
        FileModDate: '25-Jan-2013 16:18:41'
           FileSize: 2339
             Format: 'jpg'
```

```
      FormatVersion: ''
              Width: 49
             Height: 49
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {'My JPEG file'}
```

Write Multiple Images to TIFF File

Write multiple images to a single multipage TIFF file.

Create two sets of random image data, `im1` and `im2`.

```
im1 = rand(50,40,3);
im2 = rand(50,50,3);
```

Write the first image to a new TIFF file. Then, append the second image to the same file.

```
imwrite(im1,'myMultipageFile.tif')
imwrite(im2,'myMultipageFile.tif','WriteMode','append')
```

Write Animated GIF

Animate a series of plots and write the result to a GIF file.

Define the x-axis limits for the plot and specify the output file name.

```
x = 0:0.01:1;
figure
filename = 'testAnimated.gif';
```

Call `frame2im` to get image data from a single movie frame. Because three-dimensional data is not supported for GIF files, call `rgb2ind` to convert the RGB data in the image data, `im`, to an indexed image, `A`, with a colormap, `map`. Call `imwrite` with the name-value pair argument, `'WriteMode','append'`, to append multiple images to the first image.

```
for n = 1:0.5:5
y = x.^n;
plot(x,y)
drawnow
frame = getframe(1);
```

```
im = frame2im(frame);
[A,map] = rgb2ind(im,256);
 if n == 1;
  imwrite(A,map,filename,'gif','LoopCount',Inf,'DelayTime',1);
 else
  imwrite(A,map,filename,'gif','WriteMode','append','DelayTime',1);
 end
end
```

`imwrite` writes the GIF file to your current folder. Name-value pair `'LoopCount',Inf` causes the animation to continuously loop. `'DelayTime',1` specifies a 1-second delay between the display of each image in the animation.

Input Arguments

A — Image data to write

matrix

Image data to write, specified as a full (nonsparse) matrix.

- For grayscale images, `A` can be `m`-by-`n`.
- For indexed images, `A` can be `m`-by-`n`. Specify the associated colormap in the `map` input argument.
- For truecolor images, `A` must be `m`-by-`n`-by-3. `imwrite` does not support writing RGB images to GIF files.

For TIFF files, `A` can be an `m`-by-`n`-by-4 array containing color data that uses the CMYK color space.

For multiframe GIF files, `A` can be an `m`-by-`n`-by-1-by-`p` array containing grayscale or indexed images, where `p` is the number of frames to write. RGB images are not supported in this case.

Data Types: `double` | `single` | `uint8` | `uint16` | `logical`

filename — Name of output file

string

Name of the output file including the file extension, specified as a string. For a list of the image types that `imwrite` can write, see the description for the fmt input argument.

Example: `'myFile.gif'`

Data Types: `char`

`map` — Colormap of indexed image

`m`-by-3 array

Colormap associated with indexed image data in `A`, specified as an `m`-by-3 array. `map` must be a valid MATLAB colormap. See `colormap` for a list of built-in MATLAB colormaps. Most image file formats do not support colormaps with more than 256 entries.

Example: `[0,0,0;0.5,0.5,0.5;1,1,1]`

Example: `jet(60)`

Data Types: `double`

`fmt` — Format of output file

string

Format of the output file, specified as one of the following strings.

This table also summarizes the types of images that `imwrite` can write. The MATLAB file format registry determines which file formats are supported. See `imformats` for more information about this registry.

For certain formats, `imwrite` can accept additional name-value pair arguments. To view these arguments, click the linked format names below.

Value of `fmt`	Format of Output File	Description
`'bmp'`	Windows Bitmap (BMP)	1-bit, 8-bit, and 24-bit uncompressed images
`'gif'`	Graphics Interchange Format (GIF)	8-bit images
`'hdf'`	Hierarchical Data Format (HDF4)	8-bit raster image data sets with or without associated colormap, 24-bit raster image data sets
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)	8-bit, 12-bit, and 16-bit Baseline JPEG images

Value of `fmt`	Format of Output File	Description
		Note: `imwrite` converts indexed images to RGB before writing data to JPEG files, because the JPEG format does not support indexed images.
`'jp2'` or `'jpx'`	JPEG 2000 — Joint Photographic Experts Group 2000	1-bit, 8-bit, and 16-bit JPEG 2000 images
`'pbm'`	Portable Bitmap (PBM)	Any 1-bit PBM image, ASCII (plain) or raw (binary) encoding
`'pcx'`	Windows Paintbrush (PCX)	8-bit images
`'pgm'`	Portable Graymap (PGM)	Any standard PGM image; ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per gray value
`'png'`	Portable Network Graphics (PNG)	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit truecolor images; 24-bit and 48-bit truecolor images with alpha channels
`'pnm'`	Portable Anymap (PNM)	Any of the PPM/PGM/PBM formats, chosen automatically
`'ppm'`	Portable Pixmap (PPM)	Any standard PPM image: ASCII (plain) encoded with arbitrary color depth or raw (binary) encoded with up to 16 bits per color component
`'ras'`	Sun™ Raster (RAS)	Any RAS image, including 1-bit bitmap, 8-bit indexed, 24-bit truecolor, and 32-bit truecolor with alpha

Value of `fmt`	Format of Output File	Description
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)	Baseline TIFF images, including: • 1-bit, 8-bit, 16-bit, 24-bit, and 48-bit uncompressed images and images with packbits, LZW, or Deflate compression • 1-bit images with CCITT 1D, Group 3, and Group 4 compression • CIELAB, ICCLAB, and CMYK images
`'xwd'`	X Windows Dump (XWD)	8-bit ZPixmaps

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `imwrite(A,'myFile.png','BitDepth',8)` writes the data in `A` using 8 bits to represent each pixel.

GIF — Graphics Interchange Format

`'BackgroundColor'` — Color to use as background color
scalar integer

Color to use as background color for the indexed image, specified as the comma-separated pair consisting of `'BackgroundColor'` and a scalar integer corresponding to the colormap index.

The background color is used for some disposal methods in animated GIFs.

- If image data `A` is `uint8` or `logical`, then the colormap index is zero-based.
- If image data `A` is `double`, then the colormap index is one-based.

The default background color corresponds to the first color in the colormap.

Example: `'BackgroundColor',15`

'Comment' — Comment to add to image

string | cell array of strings

Comment to add to the image, specified as the comma-separated pair consisting of 'Comment' and a string or a 1-by-n cell array of strings. For a cell array of strings, imwrite adds a carriage return after each string.

Example: 'Comment',{'Sample #314','January 5, 2013'}

Data Types: char | cell

'DelayTime' — Delay before displaying next image

0.5 (default) | scalar value in the range [0,655]

Delay before displaying next image, in seconds, specified as the comma-separated pair consisting of 'DelayTime' and a scalar value in the range [0,655]. A value of 0 displays images as fast as your hardware allows.

Example: 'DelayTime',60

'DisposalMethod' — Disposal method of animated GIF

'doNotSpecify' (default) | 'leaveInPlace' | 'restoreBG' | 'restorePrevious'

Disposal method of an animated GIF, specified as the comma-separated pair consisting of 'DisposalMethod' and one of the following strings.

Value of DisposalMethod	Result
'doNotSpecify' (default)	Replace one full-size, nontransparent frame with another.
'leaveInPlace'	Any pixels not covered up by the next frame continue to display.
'restoreBG'	The background color or background tile shows through transparent pixels.
'restorePrevious'	Restore to the state of a previous, undisposed frame.

Example: 'DisposalMethod','restoreBG'

'Location' — Offset of screen relative to image

[0,0] (default) | two-element vector

Offset of the screen relative to the image, measured from the top left corner of each, specified as the comma-separated pair consisting of `'Location'` and a two-element vector. The first vector element specifies the offset from the top, and the second element specifies the offset from the left, in pixels.

Example: `'Location',[10,15]`

Data Types: `double`

`'LoopCount'` — Number of times to repeat animation

`Inf` (default) | integer in the range `[0,65535]`

Number of times to repeat the animation, specified as the comma-separated pair consisting of `'LoopCount'` and either an integer in the range `[0,65535]`, or the value `Inf`. If you specify 0, the animation plays once. If you specify the value 1, the animation plays twice, and so on. A `LoopCount` value of `Inf` causes the animation to continuously loop.

To enable animation within Microsoft PowerPoint®, specify a value for `'LoopCount'` within the range [1,65535]. Some Microsoft applications interpret the value 0 to mean do not loop at all.

Example: `'LoopCount',3`

`'ScreenSize'` — Height and width of frame

height and width of input image (default) | two-element vector

Height and width of the frame, specified as the comma-separated pair consisting of `'ScreenSize'` and a two-element vector. When you use the `ScreenSize` argument with `'Location'`, it provides a way to write frames to the image that are smaller than the whole frame. `'DisposalMethod'` determines the fill value for pixels outside the frame.

Example: `'ScreenSize',[1000 1060]`

Data Types: `double`

`'TransparentColor'` — Color to use as transparent color

scalar integer

Color to use as transparent color for the image, specified as the comma-separated pair consisting of `'TransparentColor'` and a scalar integer corresponding to the colormap index.

- If image data A is `uint8` or `logical`, then indexing begins at `0`.
- If image data A is `double`, then indexing begins at `1`.

Example: `'TransparentColor',20`

`'WriteMode'` — Writing mode

`'overwrite'` (default) | `'append'`

Writing mode, specified as the comma-separated pair consisting of `'WriteMode'` and either `'overwrite'` or `'append'`. In `overwrite` mode, `imwrite` overwrites an existing file,`filename`. In `append` mode, `imwrite` adds a single frame to the existing file.

Example: `'WriteMode','append'`

HDF4 — Hierarchical Data Format

`'Compression'` — Compression scheme

`'none'` (default) | `'jpeg'` | `'rle'`

Compression scheme, specified as the comma-separated pair consisting of `'Compression'` and one of the following strings.

Value of `Compression`	Result
`'none'` (default)	No compression
`'jpeg'`	JPEG compression. Valid only for grayscale and RGB images.
`'rle'`	Run-length encoding. Valid only for grayscale and indexed images.

Example: `'Compression','jpeg'`

`'Quality'` — Quality of JPEG-compressed file

75 (default) | scalar in the range [0,100]

Quality of the JPEG-compressed file, specified as the comma-separated pair consisting of `'Quality'` and a scalar in the range [0,100], where 0 is lower quality and higher compression, and 100 is higher quality and lower compression. This parameter applies only if `'Compression'` is `'jpeg'`.

Example: `'Quality',25`

'WriteMode' — Writing mode

'overwrite' (default) | 'append'

Writing mode, specified as the comma-separated pair consisting of 'WriteMode' and either 'overwrite' or 'append'. In overwrite mode, imwrite overwrites an existing file,filename. In append mode, imwrite adds a single frame to the existing file.

Example: 'WriteMode','append'

JPEG — Joint Photographic Experts Group

'BitDepth' — Number of bits per pixel

8 (default) | scalar

Number of bits per pixel, specified as the comma-separated pair consisting of 'BitDepth' and a scalar.

- For grayscale images, the BitDepth value can be 8, 12, or 16. The default value is 8. For 16-bit images, the 'Mode' name-value pair argument must be 'lossless'.
- For color images, the BitDepth value is the number of bits per plane, and can be 8 or 12. The default is 8 bits per plane.

Example: 'BitDepth',12

'Comment' — Comment to add to image

string | character array | n-by-1 cell array of strings

Comment to add to the image, specified as the comma-separated pair consisting of 'Comment' and a single string, a character array, or an n-by-1 cell array of strings. imwrite writes each row of input as a comment in the JPEG file.

Example: 'Comment',{'First line';'second line';'third line'}

Data Types: char | cell

'Mode' — Type of compression

'lossy' (default) | 'lossless'

Type of compression, specified as the comma-separated pair consisting of 'Mode' and one of the following strings:

- 'lossy'
- 'lossless'

Example: `'Mode','lossless'`

`'Quality'` — Quality of output file

75 (default) | scalar in the range [0,100]

Quality of the output file, specified as the comma-separated pair consisting of `'Quality'` and a scalar in the range [0,100], where 0 is lower quality and higher compression, and 100 is higher quality and lower compression. A `Quality` value of 100 does not write a lossless JPEG image. Instead, use the `'Mode','lossless'` name-value pair argument.

Example: `'Quality',100`

JPEG 2000— Joint Photographic Experts Group 2000

`'Comment'` — Comment to add to image

string | character array | cell array of strings

Comment to add to the image, specified as the comma-separated pair consisting of `'Comment'` and a single string, a character array, or a cell array of strings. `imwrite` writes each row of input as a comment in the JPEG 2000 file.

Example: `'Comment',{'First line';'second line';'third line'}`

Example: `'Comment',{'First line','second line','third line'}`

Data Types: `cell` | `char`

`'CompressionRatio'` — Target compression ratio

1 (default) | scalar

Target compression ratio, specified as the comma-separated pair consisting of `'CompressionRatio'` and a real scalar greater than or equal to 1. The compression ratio is the ratio of the input image size to the output compressed size. For example, a value of 2.0 implies that the output image size is half of the input image size or less. A higher value implies a smaller file size and reduced image quality. The compression ratio does not take into account the header size.

Specifying `CompressionRatio` is valid only when `'Mode'` is `'lossy'`.

Example: `'CompressionRatio',3`

`'Mode'` — Type of compression

`'lossy'` (default) | `'lossless'`

Type of compression, specified as the comma-separated pair consisting of `'Mode'` and one of the following strings:

- `'lossy'`
- `'lossless'`

Example: `'Mode','lossless'`

`'ProgressionOrder'` — Order of packets in code stream

`'LRCP'` (default) | `'RLCP'` | `'RPCL'` | `'PCRL'` | `'CPRL'`

Order of packets in the code stream, specified as the comma-separated pair consisting of `'ProgressionOrder'` and one of the following strings:

- `'LRCP'`
- `'RLCP'`
- `'RPCL'`
- `'PCRL'`
- `'CPRL'`

The characters in the text strings represent the following: `L` = layer, `R` = resolution, `C` = component and `P` = position.

Example: `'ProgressionOrder','RLCP'`

`'QualityLayers'` — Number of quality layers

1 (default) | integer in the range [1,20]

Number of quality layers, specified as the comma-separated pair consisting of `'QualityLayers'` and an integer in the range [1,20].

Example: `'QualityLayers',8`

`'ReductionLevels'` — Number of reduction levels

4 (default) | integer in the range [1,8]

Number of reduction levels, or wavelet decomposition levels, specified as the comma-separated pair consisting of `'ReductionLevels'` and an integer in the range [1,8].

Example: `'ReductionLevels',6`

`'TileSize'` — Tile height and width

image size (default) | two-element vector

Tile height and width, specified as the comma-separated pair consisting of `'TileSize'` and a two-element vector. The minimum size you can specify is `[128 128]`.

Example: `'TileSize',[130 130]`

PBM-, PGM-, and PPM — Portable Bitmap, Graymap, Pixmap

`'Encoding'` — Encoding
`'rawbits'` (default) | `'ASCII'`

Encoding, specified as the comma-separated pair consisting of `'Encoding'` and either `'rawbits'` for binary encoding, or `'ASCII'` for plain encoding.

Example: `'Encoding','ASCII'`

`'MaxValue'` — Maximum gray or color value
scalar

Maximum gray or color value, specified as the comma-separated pair consisting of `'MaxValue'` and a scalar.

Available only for PGM and PPM files. For PBM files, this value is always 1.

If the image array is `uint16`, then the default value for `MaxValue` is `65535`. Otherwise, the default value is `255`.

Example: `'MaxValue',510`

PNG — Portable Network Graphics

In addition to the following name-value pair arguments, you can use any parameter name that satisfies the PNG specification for keywords. That is, the name uses only printable characters, contains 80 or fewer characters, and does not contain leading or trailing spaces. The value corresponding to these user-specified names must be a string that contains no control characters other than linefeed.

`'Alpha'` — Transparency of each pixel
matrix of values in the range [0,1]

Transparency of each pixel, specified as the comma-separated pair consisting of `'Alpha'` and a matrix of values in the range [0,1]. The row and column dimensions of the `Alpha` matrix must be the same as those of the image data array. You can specify `Alpha` only for grayscale (`m`-by-`n`) and truecolor (`m`-by-`n`-by-3) image data.

Note: You cannot specify both `'Alpha'` and `'Transparency'` at the same time.

Data Types: `double` | `uint8` | `uint16`

`'Author'` — Author information

string

Author information, specified as the comma-separated pair consisting of `'Author'` and a string.

Example: `"Author','Ann Smith'`

Data Types: `char`

`'Background'` — Background color when compositing transparent pixels

scalar in the range `[0,1]` | integer in the range `[1,P]` | 3-element vector in the range `[0,1]`

Background color when compositing transparent pixels, specified as the comma-separated pair consisting of `'Background'` and a value dependent on the image data, as follows.

Image Type	Form of `Background` Value
Grayscale images	Scalar in the range `[0,1]`.
Indexed images	Integer in the range `[1,P]`, where `P` is the colormap length. For example, `'Background',50` sets the background color to the color specified by the 50th index in the colormap.
Truecolor images	Three-element vector of RGB intensities in the range `[0,1]`. For example, `'Background',[0 1 1]` sets the background color to cyan.

Data Types: `double`

`'BitDepth'` — Number of bits per pixel

scalar

Number of bits per pixel, specified as the comma-separated pair consisting of `'BitDepth'` and a scalar. Depending on the output image, the scalar can be one of the following values.

Image Type	Allowed Values for `BitDepth`
Grayscale images	1, 2, 4, 8, or 16
Grayscale images with an alpha channel	8 or 16
Indexed images	1, 2, 4, or 8
Truecolor images	8 or 16

- If the image is of class `double` or `uint8`, then the default bit depth is 8 bits per pixel.
- If the image is `uint16`, then the default is 16 bits per pixel.
- If the image is `logical`, then the default is 1 bit per pixel.

Example: `'BitDepth',4`

`'Chromaticities'` — Reference white point and primary chromaticities

8-element vector

Reference white point and primary chromaticities, specified as the comma-separated pair consisting of `'Chromaticities'` and an 8-element vector, `[wx wy rx ry gx gy bx by]`. The elements `wx` and `wy` are the chromaticity coordinates of the white point, and the elements `rx`, `ry`, `gx`, `gy`, `bx`, and `by` are the chromaticity coordinates of the three primary colors.

If you specify `Chromaticities`, you should also specify the `Gamma` name-value pair argument.

Example: `'Chromaticities', [0.312,0.329,0.002,0.002,0.001,0.001,0.115,0.312]`

Data Types: `double`

`'Comment'` — Comment to add to image

string

Comment to add to the image, specified as the comma-separated pair consisting of `'Comment'` and a string.

`'Copyright'` — Copyright notice

string

Copyright notice, specified as the comma-separated pair consisting of `'Copyright'` and a string.

`'CreationTime'` — Time of original image creation

string

Time of original image creation, specified as a string.

`'Description'` — Description of image

string

Description of the image, specified as the comma-separated pair consisting of `'Description'` and a string.

`'Disclaimer'` — Legal disclaimer

string

Legal disclaimer, specified as the comma-separated pair consisting of `'Disclaimer'` and a string.

`'Gamma'` — File gamma

scalar

File gamma, specified as the comma-separated pair consisting of `'Gamma'` and a scalar.

Example: `'Gamma',2.2`

`'ImageModTime'` — Time of last image modification

serial date number | date string

Time of the last image modification, specified as the comma-separated pair consisting of `'ImageModTime'` and a MATLAB serial date number or a date string that can be converted to a date vector using the `datevec` function. Values should be in Coordinated Universal Time (UTC).

The default `ImageModTime` value is the time when you call `imwrite`.

Example: `'ImageModTime','17-Jan-2013 11:23:10'`

Data Types: `double | char`

'InterlaceType' — Interlacing scheme

'none' (default) | 'adam7'

Interlacing scheme, specified as the comma-separated pair consisting of 'InterlaceType' and either 'none' for no interlacing, or 'adam7' to use the Adam7 algorithm.

Example: 'InterlaceType','adam7'

'ResolutionUnit' — Unit for image resolution

'unknown' (default) | 'meter'

Unit for image resolution, specified as the comma-separated pair consisting of 'ResolutionUnit' and either 'unknown' or 'meter'. If you specify ResolutionUnit, you must include at least one of the XResolution and YResolution name-value pair arguments. When the value of ResolutionUnit is 'meter', the XResolution and YResolution values are interpreted in pixels per meter.

Example: 'ResolutionUnit','meter','XResolution',1000

'SignificantBits' — Number of bits to regard as significant

[ ] (default) | scalar | vector

Number of bits in the data array to regard as significant, specified as the comma-separated pair consisting of 'SignificantBits' and a scalar or a vector in the range [1,BitDepth]. Depending on the output image type, the value must be in the following form.

Image Type	Form of SignificantBits Value
Grayscale image without an alpha channel	Scalar
Grayscale image with an alpha channel	2-element vector
Indexed image	3-element vector
Truecolor image without an alpha channel	3-element vector
Truecolor image with an alpha channel	4-element vector

Example: 'SignificantBits',[2,3]

'Software' — Software used to create the image

string

Software used to create the image, specified as the comma-separated pair consisting of `'Software'` and a string.

`'Source'` — Device used to create the image

string

Device used to create the image, specified as the comma-separated pair consisting of `'Source'` and a string.

`'Transparency'` — Pixels to consider transparent

[ ] (default) | scalar in the range [0,1] | vector

Pixels to consider transparent when no alpha channel is used, specified as the comma-separated pair consisting of `'Transparency'` and a scalar or a vector. Depending on the output image, the value must be in the following form.

Image Type	Form of `Transparency` Value
Grayscale images	Scalar in the range [0,1], indicating the grayscale color to be considered transparent.
Indexed images	`Q`-element vector of values in the range [0,1], where `Q` is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, `Q = 1`.
Truecolor images	3-element vector of RGB intensities in the range [0,1], indicating the truecolor color to consider transparent.

Note: You cannot specify both `'Transparency'` and `'Alpha'` at the same time.

Example: `'Transparency',[1 1 1]`

Data Types: `double`

`'Warning'` — Warning of nature of content

string

Warning of nature of content, specified as the comma-separated pair consisting of `'Warning'` and a string.

`'XResolution'` — Image resolution in horizontal direction

scalar

Image resolution in the horizontal direction, in pixels/unit, specified as the comma-separated pair consisting of `'XResolution'` and a scalar. Define the unit by specifying the `ResolutionUnit` name-value pair argument.

If you do not also specify `YResolution`, then the `XResolution` value applies to both the horizontal and vertical directions.

Example: `'XResolution',900`

`'YResolution'` — Image resolution in vertical direction

scalar

Image resolution in the vertical direction, in pixels/unit, specified as the comma-separated pair consisting of `'XResolution'` and a scalar. Define the unit by specifying the `ResolutionUnit` name-value pair argument.

If you do not also specify `XResolution`, then the `YResolution` value applies to both the horizontal and vertical directions.

Example: `'YResolution',900`

RAS — Sun Raster Graphic

`'Alpha'` — Transparency of each pixel

[ ] (default) | matrix

Transparency of each pixel, specified as the comma-separated pair consisting of `'Alpha'` and a matrix with row and column dimensions the same as those of the image data array.

Valid only for truecolor (`m`-by-`n`-by-3) image data.

Data Types: `double` | `single` | `uint8` | `uint16`

`'Type'` — Image type

`'standard'` (default) | `'rgb'` | `'rle'`

Image type, specified as the comma-separated pair consisting of `'Type'` and one of the following strings.

Value of `Type`	Description
`'standard'` (default)	Uncompressed, B-G-R color order for truecolor images
`'rgb'`	Uncompressed, R-G-B color order for truecolor images
`'rle`	Run-length encoding of 1-bit and 8-bit images

Example: `'Type','rgb'`

TIFF — Tagged Image File Format

`'ColorSpace'` — Color space representing color data
`'rgb'` (default) | `'cielab'` | `'icclab'`

Color space representing the color data, specified as the comma-separated pair consisting of `'ColorSpace'` and one of the following strings:

- `'rgb'`
- `'cielab'`
- `'icclab'`

Valid only when the image data array, `A`, is truecolor (`m`-by-`n`-by-3). To use the CMYK color space in a TIFF file, do not use the `'ColorSpace'` name-value pair argument. Instead, specify an `m`-by-`n`-by-4 image data array.

`imwrite` can write color image data that uses the *L*a*b** color space to TIFF files. The 1976 CIE *L*a*b** specification defines numeric values that represent luminance (*L**) and chrominance (*a** and *b**) information. To store *L*a*b** color data in a TIFF file, the values must be encoded to fit into either 8-bit or 16-bit storage. `imwrite` can store *L*a*b** color data in a TIFF file using the following encodings:

- CIELAB encodings — 8-bit and 16-bit encodings defined by the TIFF specification
- ICCLAB encodings — 8-bit and 16-bit encodings defined by the International Color Consortium

The output class and encoding used by `imwrite` depends on the class of the input image data array and the `ColorSpace` value, as shown in the following table. (The 8-bit and 16-bit CIELAB encodings cannot be input arrays because they use a mixture of signed and unsigned values and cannot be represented as a single MATLAB array.)

Input Class and Encoding	Value of `ColorSpace`	Output Class and Encoding
8-bit ICCLAB Values are integers in the range [0 255]. L^* values are multiplied by `255/100`. `128` is added to both the a^* and b^* values.	`'icclab'`	8-bit ICCLAB
	`'cielab'`	8-bit CIELAB
16-bit ICCLAB Values are integers in the range [0, 65280]. L^* values are multiplied by `65280/100`. `32768` is added to both the a^* and b^* values, which are represented as integers in the range [0,65535].	`'icclab'`	16-bit ICCLAB
	`'cielab'`	16-bit CIELAB
Double-precision 1976 CIE $L^*a^*b^*$ values L^* is in the dynamic range [0, 100]. a^* and b^* can take any value. Setting a^* and b^* to 0 (zero) produces a neutral color (gray).	`'icclab'`	8-bit ICCLAB
	`'cielab'`	8-bit CIELAB

Example: `'ColorSpace','cielab'`

`'Compression'` — Compression scheme

`'packbits' | 'none' | 'lzw' | 'deflate' | 'jpeg' | 'ccitt' | 'fax3' | 'fax4'`

Compression scheme, specified as the comma-separated pair consisting of `'Compression'` and one of the following strings:

- `'packbits'` (default for nonbinary images)
- `'none'`
- `'lzw'`
- `'deflate'`
- `'jpeg'`
- `'ccitt'` (binary images only, and the default for such images)
- `'fax3'` (binary images only)
- `'fax4'` (binary images only)

`'jpeg'` is a lossy compression scheme; other compression modes are lossless. Also, if you specify `'jpeg'` compression, you must specify the `'RowsPerStrip'` parameter and the value must be a multiple of 8.

Example: `'Compression','none'`

`'Description'` — Image description

string

Image description, specified by the comma-separated pair consisting of `'Description'` and a string. This is the text that `imfinfo` returns in the `ImageDescription` field for the output image.

Example: `'Description','Sample 2A301'`

`'Resolution'` — X- and Y-resolution

72 (default) | scalar | two-element vector

X- and Y-resolution, specified as the comma-separated pair consisting of `'Resolution'` and a scalar indicating both resolution, or a two-element vector containing the X-Resolution and Y-Resolution.

Example: `'Resolution',80`

Example: `'Resolution',[320,72]`

Data Types: `double`

`'RowsPerStrip'` — Number of rows to include in each strip

scalar

Number of rows to include in each strip, specified as the comma-separated pair consisting of `'RowsPerStrip'` and a scalar. The default value is such that each strip is about 8 kilobytes.

You must specify `RowsPerStrip` if you specify `'jpeg'` compression. The value must be a multiple of 8.

Example: `'RowsPerStrip',16`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'WriteMode'` — Writing mode

`'overwrite'` (default) | `'append'`

Writing mode, specified as the comma-separated pair consisting of `'WriteMode'` and either `'overwrite'` or `'append'`. In `overwrite` mode, `imwrite` overwrites an existing file. In `append` mode, `imwrite` adds a page to the existing file.

Example: `'WriteMode','append'`

More About

Tips

- For copyright information, see the `libtiffcopyright.txt` file.

See Also

`fwrite` | `getframe` | `imfinfo` | `imformats` | `imread` | `Tiff`

Introduced before R2006a

incenters

Class: TriRep

(Will be removed) Incenters of specified simplices

Note: `incenters(TriRep)` will be removed in a future release. Use `incenter(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
IC = incenters(TR,SI)
[IC RIC] = incenters(TR, SI)
```

Description

`IC = incenters(TR,SI)` returns the coordinates of the incenter of each specified simplex `SI`.

`[IC RIC] = incenters(TR, SI)` returns the incenters and the corresponding radius of the inscribed circle/sphere.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the incenter information for the entire triangulation is returned, where the incenter associated with simplex `i` is the `i`'th row of `IC`.

Output Arguments

`IC`	`m`-by-`n` matrix, where `m = length(SI)`, the number of specified simplices, and `n` is the dimension of the space where the triangulation resides. Each row `IC(i,:)` represents the coordinates of the incenter of simplex `SI(i)`.
`RIC`	Vector of length `length(SI)`, the number of specified simplices.

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Examples

Example 1

Load a 3-D triangulation:

```
load tetmesh
```

Use `TriRep` to compute the incenters of the first five tetrahedra.

```
trep = TriRep(tet, X)
ic = incenters(trep, [1:5]')
```

Example 2

Query a 2-D triangulation created with `DelaunayTri`.

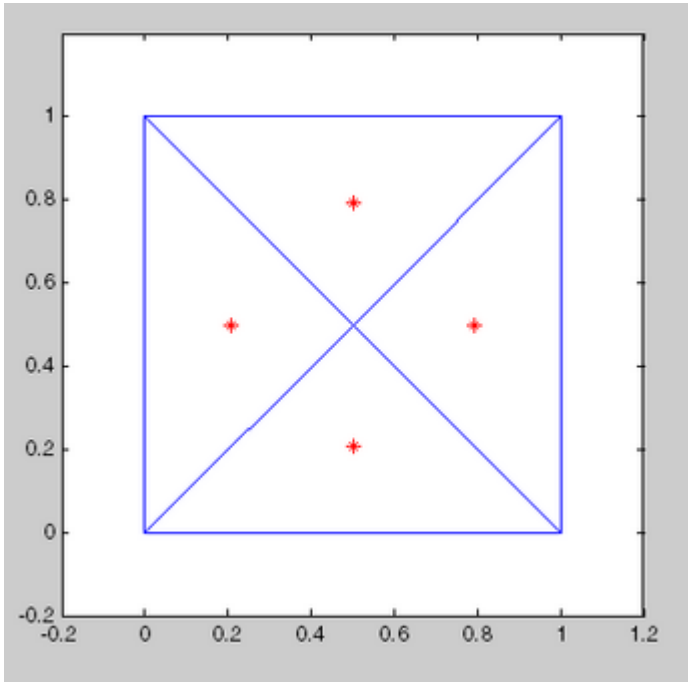
```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

Compute incenters of the triangles:

```
ic = incenters(dt);
```

Plot the triangles and incenters:

```
triplot(dt);
axis equal;
axis([-0.2 1.2 -0.2 1.2]);
hold on;
plot(ic(:,1),ic(:,2),'*r');
hold off;
```



See Also

`triangulation` | `delaunayTriangulation` | `circumcenter`

inOutStatus

Class: DelaunayTri

(Will be removed) Status of triangles in 2-D constrained Delaunay triangulation

---

Note: `inOutStatus(DelaunayTri)` will be removed in a future release. Use `isInterior(delaunayTriangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
IN = inOutStatus(DT)
```

Description

`IN = inOutStatus(DT)` returns the in/out status of the triangles in a 2-D constrained Delaunay triangulation of a geometric domain. Given a Delaunay triangulation that has a set of constrained edges that define a bounded geometric domain. The `i`'th triangle in the triangulation is classified as inside the domain if `IN(i) = 1` and outside otherwise.

Note: `inOutStatus` is only relevant for 2-D constrained Delaunay triangulations where the imposed edge constraints bound a closed geometric domain.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

`IN`	Logical array of length equal to the number of triangles in the triangulation. The constrained edges in the triangulation define the boundaries of a valid geometric domain.

Examples

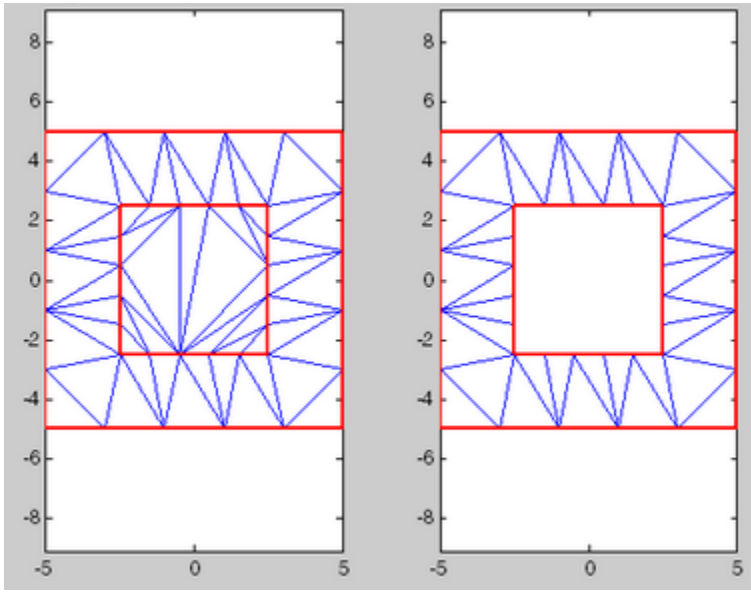
Create a geometric domain that consists of a square with a square hole:

```
outerprofile = [-5 -5; -3 -5; -1 -5; 1 -5; 3 -5; ...
 5 -5; 5 -3; 5 -1; 5 1; 5 3;...
 5 5; 3 5; 1 5; -1 5; -3 5; ...
 -5 5; -5 3; -5 1; -5 -1; -5 -3; ];
innerprofile = outerprofile.*0.5;
profile = [outerprofile; innerprofile];
outercons = [(1:19)' (2:20)'; 20 1;];
innercons = [(21:39)' (22:40)'; 40 21];
edgeconstraints = [outercons; innercons];
```

Create a constrained Delaunay triangulation of the domain:

```
dt = DelaunayTri(profile, edgeconstraints)
subplot(1,2,1);
triplot(dt);
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
axis equal;
% Plot showing interior and exterior
% triangles with respect to the domain.
hold off;
subplot(1,2,2);
inside = inOutStatus(dt);
triplot(dt(inside, :), dt.X(:,1), dt.X(:,2));
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
axis equal;
```

```
% Plot showing interior triangles only
hold off;
```



See Also

`triangulation` | `delaunayTriangulation` | `isInterior`

ind2rgb

Convert indexed image to RGB image

Syntax

```
RGB = ind2rgb(X,map)
```

Description

`RGB = ind2rgb(X,map)` converts the indexed image, `X`, and the corresponding colormap, `map`, to the truecolor image, `RGB`. The indexed image, `X`, is an m-by-n array of integers. The colormap, `map`, is a three-column array of values in the range [0,1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

- If you specify `X` as an array of class `uint8` or `uint16`, then the value 0 corresponds to the first color in the colormap.
- If you specify `X` as an array of class `single` or `double`, then the value 1 corresponds to the first color in the colormap.

The truecolor image output, `RGB`, is an m-by-n-by-3 array. For more information on image types, see “Image Types”.

Class Support

`X` can be of class `uint8`, `uint16`, `single`, or `double`. `RGB` is an m-by-n-by-3 array of class `double`.

See Also

`image` | `imread`

Introduced before R2006a

ind2sub

Subscripts from linear index

Syntax

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

Description

The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

`[I,J] = ind2sub(siz,IND)` returns the matrices `I` and `J` containing the equivalent row and column subscripts corresponding to each linear index in the matrix `IND` for a matrix of size `siz`. `siz` is a vector with `ndim(A)` elements (in this case, 2), where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

Note For matrices, `[I,J] = ind2sub(size(A),find(A>5))` returns the same values as `[I,J] = find(A>5)`.

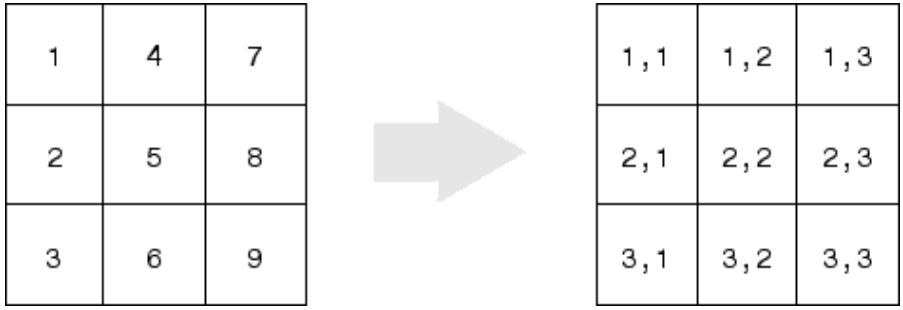
`[I1,I2,I3,...,In] = ind2sub(siz,IND)` returns `n` subscript arrays `I1,I2,...,In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

The `IND` input can be `single`, `double`, or any integer type. The outputs are always of class `double`.

Examples

Example 1 — Two-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 3-by-3 matrix is

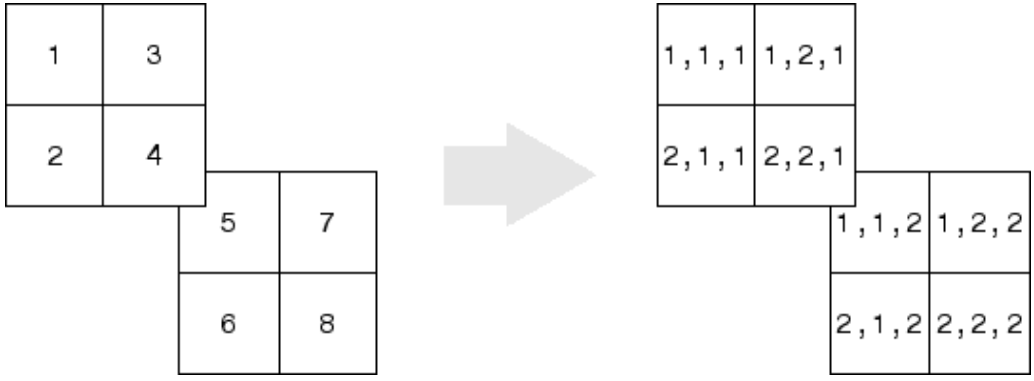


This code determines the row and column subscripts in a 3-by-3 matrix, of elements with linear indices 3, 4, 5, 6.

```
IND = [3 4 5 6]
s = [3,3];
[I,J] = ind2sub(s,IND)

I =
     3     1     2     3

J =
     1     2     2     2
```

Example 2 — Three-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is



This code determines the subscript equivalents in a 2-by-2-by-2 array, of elements whose linear indices 3, 4, 5, 6 are specified in the `IND` matrix.

```
IND = [3 4;5 6];
s = [2,2,2];
[I,J,K] = ind2sub(s,IND)

I =
     1     2
     1     2

J =
     2     2
     1     1

K =
     1     1
     2     2
```

Example 3 — Effects of Returning Fewer Outputs

When calling `ind2sub` for an N-dimensional matrix, you would typically supply N output arguments in the call: one for each dimension of the matrix. This example shows what happens when you return three, two, and one output when calling `ind2sub` on a 3-dimensional matrix.

The matrix is 2-by-2-by-2 and the linear indices are 1 through 8:

```
dims = [2 2 2];
indices = [1 2 3 4 5 6 7 8];
```

The 3-output call to `ind2sub` returns the expected subscripts for the 2-by-2-by-2 matrix:

```
[rowsub colsub pagsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     1     1     2     2
pagsub =
     1     1     1     1     2     2     2     2
```

If you specify only two outputs (row and column), `ind2sub` still returns a subscript for each specified index, but drops the third dimension from the matrix, returning subscripts for a 2-dimensional, 2-by-4 matrix instead:

```
[rowsub colsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     3     3     4     4
```

If you specify one output (row), `ind2sub` drops both the second and third dimensions from the matrix, and returns subscripts for a 1-dimensional, 1-by-8 matrix instead:

```
[rowsub] = ind2sub(dims, indices)
rowsub =
     1     2     3     4     5     6     7     8
```

See Also

`find` | `size` | `sub2ind`

Introduced before R2006a

Inf

Infinity

Syntax

```
Inf
I = Inf(n)
I = Inf(sz1,...,szN)
I = Inf(sz)

I = Inf(classname)
I = Inf(n,classname)
I = Inf(sz1,...,szN,classname)
I = Inf(sz,classname)

I = Inf('like',p)
I = Inf(n,'like',p)
I = Inf(sz1,...szN,'like',p)
I = Inf(sz,'like',p)
```

Description

`Inf` returns the IEEE arithmetic representation for positive infinity. Infinity values result from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

`I = Inf(n)` is an `n`-by-`n` matrix of `Inf` values.

`I = Inf(sz1,...,szN)` is a `sz1`-by-...-by-`szN` array of `Inf` values where `sz1,...,szN` indicates the size of each dimension. For example, `Inf(3,4)` returns a 3-by-4 array of `Inf` values.

`I = Inf(sz)` is an array of `Inf` values where the size vector, `sz`, defines `size(I)`. For example, `Inf([3,4])` returns a 3-by-4 array of `Inf` values.

Note The size inputs `sz1,...,szN`, as well as the elements of the size vector `sz`, should be nonnegative integers. Negative integers are treated as 0.

`I = Inf(classname)` returns an `Inf` value where the string, `classname`, specifies the data type. `classname` can be either `'single'` or `'double'`.

`I = Inf(n,classname)` returns an `n`-by-`n` array of `Inf` values of data type `classname`.

`I = Inf(sz1,...,szN,classname)` returns a `sz1`-by-...-by-`szN` array of `Inf` values of data type `classname`.

`I = Inf(sz,classname)` returns an array of `Inf` values where the size vector, `sz`, defines `size(I)` and `classname` defines `class(I)`.

`I = Inf('like',p)` returns an array of `Inf`s of the same data type, sparsity, and complexity (real or complex) as the numeric variable, `p`.

`I = Inf(n,'like',p)` returns an `n`-by-`n` array of `Inf` values like `p`.

`I = Inf(sz1,...szN,'like',p)` returns a `sz1`-by-...-by-`szN` array of `Inf` values like `p`.

`I = Inf(sz,'like',p)` returns an array of `Inf` values like `p` where the size vector, `sz`, defines `size(I)`.

Examples

`1/0`, `1.e1000`, `2^2000`, and `exp(1000)` all produce `Inf`.

`log(0)` produces `-Inf`.

`Inf-Inf` and `Inf/Inf` both produce `NaN` (Not-a-Number).

More About

- "Class Support for Array-Creation Functions"

See Also

`isfinite` | `isfloat` | `isinf` | `nan`

Introduced before R2006a

inferiorto

Specify inferior class relationship

Syntax

```
inferiorto('class1','class2',...)
```

Description

`inferiorto('class1','class2',...)` establishes that the class invoking this function in its constructor has lower precedence than the classes in the argument list. MATLAB uses this precedence to determines which method or function MATLAB calls in any given situation.

Use this function only from a constructor that calls the `class` function to create objects (classes defined before MATLAB 7.6).

Examples

Specify class precedence.

Suppose `a` is an object of class `class_a`, `b` is an object of class `class_b`, and `c` is an object of class `class_c`. Suppose the constructor method of `class_c` contains the statement:

```
inferiorto('class_a')
```

This function call establishes `class_a` as taking precedence over `class_c` for function dispatching. Therefore, either of the following two statements:

```
e = fun(a,c);
e = fun(c,a);
```

Invoke `class_a/fun`.

If you call a function with two objects having an unspecified relationship, the two objects have equal precedence. In this case, MATLAB calls the method of the left-most object. So `fun(b, c)` calls `class_b/fun`, while `fun(c, b)` calls `class_c/fun`.

See Also

`superiorto`

Introduced before R2006a

info

Information about contacting MathWorks

Note: `info` will be removed in a future release.

Syntax

```
info
```

Description

`info` displays in the Command Window, information about contacting MathWorks.

See Also

`help` | `version`

Introduced before R2006a

inline

Construct inline object

Compatibility

`inline` will be removed in a future release. Use “Anonymous Functions” instead.

Syntax

```
inline(expr)
inline(expr,arg1,arg2,...)
inline(expr,n)
```

Description

`inline(expr)` constructs an inline function object from the MATLAB expression contained in the string `expr`. The input argument to the inline function is automatically determined by searching `expr` for an isolated lower case alphabetic character, other than `i` or `j`, that is not part of a word formed from several alphabetic characters. If no such character exists, `x` is used. If the character is not unique, the one closest to `x` is used. If two characters are found, the one later in the alphabet is chosen.

`inline(expr,arg1,arg2,...)` constructs an inline function whose input arguments are specified by the strings `arg1`, `arg2`,`...`. Multicharacter symbol names may be used.

`inline(expr,n)` where `n` is a scalar, constructs an inline function whose input arguments are `x`, `P1`, `P2`, ...

Examples

Example 1

This example creates a simple inline function to square a number.

```
g = inline('t^2')
g =

     Inline function:
     g(t) = t^2
```

You can convert the result to a string using the `char` function.

```
char(g)

ans =

t^2
```

Example 2

This example creates an inline function to represent the formula $f = 3\sin(2x^2)$. The resulting inline function can be evaluated with the `argnames` and `formula` functions.

```
f = inline('3*sin(2*x.^2)')

f =
     Inline function:
     f(x) = 3*sin(2*x.^2)

argnames(f)

ans =
    'x'

formula(f)
ans =

3*sin(2*x.^2)
```

Example 3

This call to `inline` defines the function `f` to be dependent on two variables, `alpha` and `x`:

```
f = inline('sin(alpha*x)')

f =
```

```
     Inline function:
     f(alpha,x) = sin(alpha*x)
```

If `inline` does not return the desired function variables or if the function variables are in the wrong order, you can specify the desired variables explicitly with the `inline` argument list.

```
g = inline('sin(alpha*x)','x','alpha')

g =

     Inline function:
     g(x,alpha) = sin(alpha*x)
```

More About

Tips

Three commands related to `inline` allow you to examine an inline function object and determine how it was created.

`char(fun)` converts the inline function into a character array. This is identical to `formula(fun)`.

`argnames(fun)` returns the names of the input arguments of the inline object `fun` as a cell array of strings.

`formula(fun)` returns the formula for the inline object `fun`.

A fourth command `vectorize(fun)` inserts a `.` before any `^`, `*` or `/` in the formula for `fun`. The result is a vectorized version of the inline function.

Introduced before R2006a

inmem

Names of functions, MEX-files, classes in memory

Syntax

```
M = inmem
[M,X] = inmem
[M,X,C] = inmem
[...] = inmem('-completenames')
```

Description

`M = inmem` returns a cell array of strings containing the names of the functions that are currently loaded.

`[M,X] = inmem` returns an additional cell array `X` containing the names of the MEX-files that are currently loaded.

`[M,X,C] = inmem` also returns a cell array `C` containing the names of the classes that are currently loaded.

`[...] = inmem('-completenames')` returns not only the names of the currently loaded function and MEX-files, but the path and filename extension for each as well. No additional information is returned for loaded classes.

Examples

Functions in Memory

List the functions that remain in memory after calling the `magic` function.

```
clear all
magic(10);

M = inmem
```

```
M =
```

```
    'workspacefunc'
    'magic'
```

The function list includes `magic` and additional functions that are in memory in your current session.

MEX-Files in Memory

Call a sample MEX-function named `arrayProduct`, and then verify that the MEX-function is in memory. You must have a supported C compiler installed on your system to run this example.

```
clear all
sampleFolder = fullfile(matlabroot,'extern','examples','mex');
addpath(sampleFolder)
mex arrayProduct.c

s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A);

[M,X] = inmem('-completenames');
X
```

```
X =
    'matlabroot\extern\examples\mex\arrayProduct.mexw64'
```

See Also

`clear`

Introduced before R2006a

innerjoin

Inner join between two tables

Syntax

```
C = innerjoin(A,B)
C = innerjoin(A,B,Name,Value)
[C,ia,ib] = innerjoin( ___ )
```

Description

`C = innerjoin(A,B)` creates the table, `C`, as the inner join between the tables `A` and `B` by matching up rows using all the variables with the same name as key variables.

The inner join retains only the rows that match between `A` and `B` with respect to the key variables. `C` contains all nonkey variables from `A` and `B`.

`C = innerjoin(A,B,Name,Value)` performs the inner-join operation with additional options specified by one or more Name,Value pair arguments.

For example, you can specify the variables to use as key variables.

`[C,ia,ib] = innerjoin( ___ )` also returns index vectors, ia and ib indicating the correspondence between rows in `C` and those in `A` and `B` respectively. You can use this syntax with any of the input arguments in the previous syntaxes.

Examples

Inner-Join Operation of Tables with One Variable in Common

Create a table, `A`.

```
A = table([5;12;23;2;6],...
    {'cereal';'pizza';'salmon';'cookies';'pizza'},...
    'VariableNames',{'Age','FavoriteFood'})
```

```
A =

    Age    FavoriteFood
```

```
    ___    ____________

     5     'cereal'
    12     'pizza'
    23     'salmon'
     2     'cookies'
     6     'pizza'
```

Create a table, `B`, with one variable in common with `A`.

```
B = table({'cereal';'cookies';'pizza';'salmon';'cake'},...
    [110;160;140;367;243],...
    {'A-';'D';'B';'B';'C-'},...
    'VariableNames',{'FavoriteFood','Calories','NutritionGrade'})
```

```
B =

    FavoriteFood    Calories    NutritionGrade
    ____________    ________    ______________

    'cereal'        110         'A-'
    'cookies'       160         'D'
    'pizza'         140         'B'
    'salmon'        367         'B'
    'cake'          243         'C-'
```

Use the`innerjoin` function to create a new table, `C`, with data from tables `A` and `B`.

```
C = innerjoin(A,B)
```

```
C =

    Age    FavoriteFood    Calories    NutritionGrade
    ___    ____________    ________    ______________

     5     'cereal'        110         'A-'
     2     'cookies'       160         'D'
    12     'pizza'         140         'B'
     6     'pizza'         140         'B'
    23     'salmon'        367         'B'
```

Table `C` is sorted by the key variable, `FavoriteFood`.

Inner-Join Operation of Tables and Indices to Values

Create a table, `A`.

```
A = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
    'VariableNames',{'Key1' 'Var1'})
```

```
A =

    Key1    Var1
    ____    ____

    'a'      1
    'b'      2
    'c'      3
    'e'     11
    'h'     17
```

Create a table, `B`, with common values in the variable `Key1` between tables `A` and `B`, but also containing rows with values of `Key1` not present in `A`.

```
B = table({'a' 'b' 'd' 'e'}',[4 5 6 7]',...
    'VariableNames',{'Key1' 'Var2'})
```

```
B =

    Key1    Var2
    ____    ____

    'a'     4
    'b'     5
    'd'     6
    'e'     7
```

Use the `innerjoin` function to create a new table, `C`, with data from tables `A` and `B`. Retain only rows whose values in the variable `Key1` match.

Also, return index vectors, `ia` and `ib` indicating the correspondence between rows in `C` and rows in `A` and `B` respectively.

```
[C,ia,ib] = innerjoin(A,B)
```

```
 C =

    Key1    Var1    Var2
    ____    ____    ____

    'a'      1      4
    'b'      2      5
    'e'     11      7
```

```
ia =

     1
     2
     4


ib =

     1
     2
     4
```

Table `C` is sorted by the values in the key variable, `Key1`, and contains the horizontal concatenation of `A(ia,:)` and `B(ib,'Var2')` .

Inner-Join Operation of Tables Using Left and Right Keys

Create a table, `A`.

```
A = table([10;4;2;3;7],[5;4;9;6;1],[10;3;8;8;4])
```

```
A =

    Var1    Var2    Var3
    ____    ____    ____

    10      5       10
     4      4        3
     2      9        8
     3      6        8
     7      1        4
```

Create a table, `B`, with common values in the second variable as the first variable of table `A`.

```
B = table([6;1;1;6;8],[2;3;4;5;6])
```

```
B =

    Var1    Var2
    ____    ____
```

```
    6       2
    1       3
    1       4
    6       5
    8       6
```

Use the `innerjoin` function to create a new table, `C`, with data from tables `A` and `B`. Use the first variable of `A` and the second variable of `B` as key variables.

```
[C,ia,ib] = innerjoin(A,B,'LeftKeys',1,'RightKeys',2)
```

```
C =

    Var1_A    Var2    Var3    Var1_B
    ______    ____    ____    ______

    2         9       8       6
    3         6       8       1
    4         4       3       1


ia =

     3
     4
     2


ib =

     1
     2
     3
```

Table `C` retains only the rows that match between `A` and `B` with respect to the key variables.

Table `C` contains the horizontal concatenation of `A(ia,:)` and `B(ib,'Var1')`.

Input Arguments

`A,B` — Input tables

tables

Input tables, specified as tables.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Keys',2` uses the second variable in `A` and the second variable in `B` as key variables.

`'Keys'` — Variables to use as keys

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys, specified as the comma-separated pair consisting of `'Keys'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You cannot use the `'Keys'` name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

Example: `'Keys',[1 3]` uses the first and third variables in `A` and `B` as a key variables.

`'LeftKeys'` — Variables to use as keys in `A`

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys in `A`, specified as the comma-separated pair consisting of `'LeftKeys'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You must use the `'LeftKeys'` name-value pair argument in conjunction with the `'RightKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `innnerjoin` pairs key values based on their order.

Example: `'LeftKeys',1` uses only the first variable in `A` as a key variable.

`'RightKeys'` — Variables to use as keys in `B`

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys in B, specified as the comma-separated pair consisting of 'RightKeys' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You must use the 'RightKeys' name-value pair argument in conjunction with the 'LeftKeys' name-value pair argument. 'LeftKeys' and 'RightKeys' both must specify the same number of key variables. innerjoin pairs key values based on their order.

Example: 'RightKeys',3 uses only the third variable in B as a key variable.

'LeftVariables' — Variables from A to include in C

positive integer | vector of positive integers | variable name | cell array containing one or more variable names | logical vector

Variables from A to include in C, specified as the comma-separated pair consisting of 'LeftVariables' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You can use 'LeftVariables' to include or exclude key variables, as well as nonkey variables from the output, C.

By default, innerjoin includes all variables from A.

'RightVariables' — Variables from B to include in C

positive integer | vector of positive integers | variable name | cell array containing one or more variable names | logical vector

Variables from B to include in C, specified as the comma-separated pair consisting of 'RightVariables' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You can use 'RightVariables' to include or exclude key variables, as well as nonkey variables from the output, C.

By default, innerjoin includes all the variables from B except the key variables.

Output Arguments

C — Inner join from A and B

table

Inner join from A and B, returned as a table. The output table, C, contains one row for each pair of rows in tables A and B that share the same combination of values in the key variables. If A and B contain variables with the same name, innerjoin adds a unique suffix to the corresponding variable names in C.

In general, if there are m rows in table A and n rows in table B that all contain the same combination of values in the key variables, table C contains m*n rows for that combination.

C is sorted by the values in the key variables and contains the horizontal concatenation of A(ia,LeftVars) and B(ib,RightVars). By default, LeftVars consists of all the variables of A, and RightVars consists of all the nonkey variables from B. Otherwise, LefttVars consists of the variables specified by the 'LeftVariables' name-value pair argument, and RightVars is the variables specified by the 'RightVariables' name-value pair argument.

You can store additional metadata such as descriptions, variable units, variable names, and row names in the output table, C. For more information, see Table Properties.

ia — Index to A
column vector

Index to A, returned as a column vector. Each element of ia identifies the row in table A that corresponds to that row in the output table, C.

ib — Index to B
column vector

Index to B, returned as a column vector. Each element of ib identifies the row in table B that corresponds to that row in the output table, C.

More About

Key Variable

Variable used to match and combine data between the input tables, A and B.

See Also
join | outerjoin

inpolygon

Points located inside or on edge of polygonal region

Syntax

```
in = inpolygon(xq,yq,xv,yv)
[in,on] = inpolygon(xq,yq,xv,yv)
```

Description

`in = inpolygon(xq,yq,xv,yv)` returns `in` indicating if the query points specified by `xq` and `yq` are inside or on the edge of the polygon area defined by `xv` and `yv`.

`[in,on] = inpolygon(xq,yq,xv,yv)` also returns `on` indicating if the query points are on the edge of the polygon area.

Examples

Points Inside Convex Polygon

Define a pentagon and a set of points. Then, determine which points lie inside (or on the edge) of the pentagon.

Define the x and y coordinates of polygon vertices to create a pentagon.

```
L = linspace(0,2.*pi,6);
xv = cos(L)';
yv = sin(L)';
```

Define x and y coordinates of 250 random query points. Initialize the random-number generator to make the output of `randn` repeatable.

```
rng default
xq = randn(250,1);
```

```
yq = randn(250,1);
```

Determine whether each point lies inside or on the edge of the polygon area. Also determine whether any of the points lie on the edge of the polygon area.

```
[in,on] = inpolygon(xq,yq,xv,yv);
```

Determine the number of points lying inside or on the edge of the polygon area.

```
numel(xq(in))
```

```
ans =

    80
```

Determine the number of points lying on the edge of the polygon area.

```
numel(xq(on))
```

```
ans =

     0
```

Since there are no points lying on the edge of the polygon area, all 80 points identified by `xq(in)`, `yq(in)` are strictly inside the polygon area.

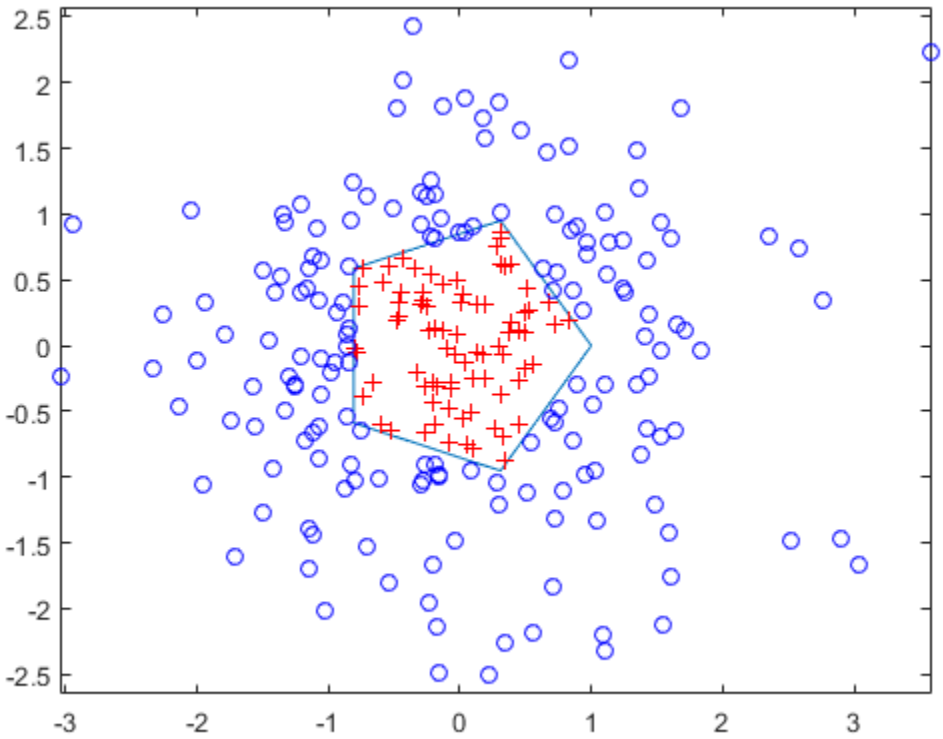
Determine the number of points lying outside the polygon area (not inside or on the edge).

```
numel(xq(~in))
```

```
ans =

   170
```

Plot the polygon and the query points. Display the points inside the polygon with a red plus. Display the points outside the polygon with a blue circle.

```
figure

plot(xv,yv) % polygon
axis equal

hold on
plot(xq(in),yq(in),'r+') % points inside
plot(xq(~in),yq(~in),'bo') % points outside
hold off
```



Points Inside Multiply Connected Polygon

Find the points inside a square with a square hole.

Define a square region with a square hole. Specify the vertices of the outer loop in a counterclockwise direction, and specify the vertices for the inner loop in a clockwise direction. Use `NaN` to separate the coordinates for the outer and inner loops.

```
xv = [1 4 4 1 1 NaN 2 2 3 3 2];
yv = [1 1 4 4 1 NaN 2 3 3 2 2];
```

Define x and y coordinates of 500 random points. Initialize the random-number generator to make the output of `randn` repeatable.

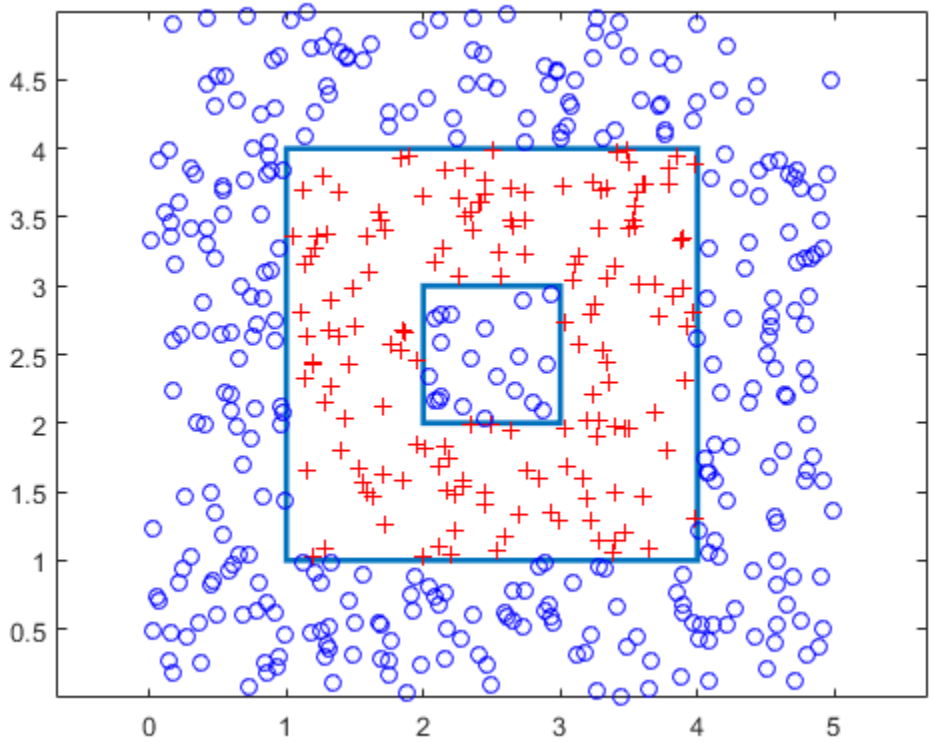
```
rng default
xq = rand(500,1)*5;
yq = rand(500,1)*5;
```

Determine whether each point lies inside or on the edge of the polygon area.

```
in = inpolygon(xq,yq,xv,yv);
```

Plot the polygon and the query points. Display the points inside the polygon with a red plus. Display the points outside the polygon with a blue circle.

```
figure

plot(xv,yv,'LineWidth',2) % polygon
axis equal

hold on
plot(xq(in),yq(in),'r+') % points inside
plot(xq(~in),yq(~in),'bo') % points outside
hold off
```



Query points in the square hole are outside the polygon.

Points Inside Self-Intersecting Polygon

Define the x and y coordinates for a pentagram.

```
xv = [0.5;0.2;1.0;0;0.8;0.5];
yv = [1.0;0.1;0.7;0.7;0.1;1];
```

Define the x and y coordinates of 12 query points.

```
xq = [0.1;0.5;0.9;0.2;0.4;0.5;0.5;0.9;0.6;0.8;0.7;0.2];
yq = [0.4;0.6;0.9;0.7;0.3;0.8;0.2;0.4;0.4;0.6;0.2;0.6];
```

Determine whether each point lies inside or on the edge of the polygon area. Also determine whether any of the points lie on the edge of the polygon area.

```
[in,on] = inpolygon(xq,yq,xv,yv);
```

Determine the number of points lying inside or on the edge of the polygon area.

```
numel(xq(in))
```

```
ans =

     8
```

Determine the number of points lying on the edge of the polygon area.

```
numel(xq(on))
```

```
ans =

     2
```

Determine the number of points lying outside the polygon area (not inside or on the edge).

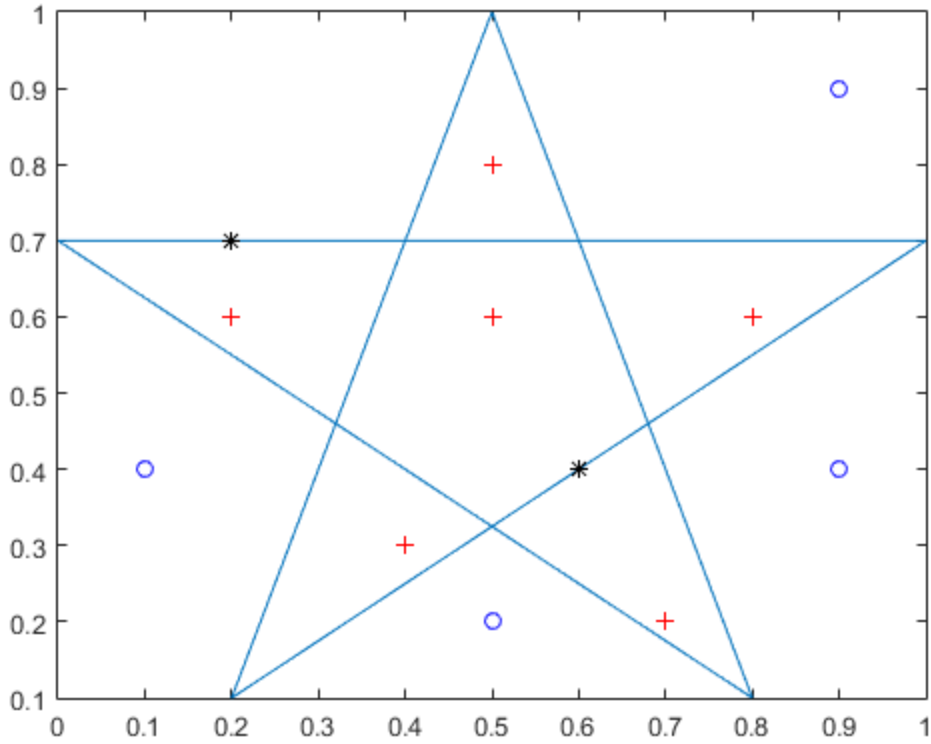
```
numel(xq(~in))
```

```
ans =

     4
```

Plot the polygon and the points. Display the points strictly inside the polygon with a red plus. Display the points on the edge with a black asterisk. Display the points outside the polygon with a blue circle.

```
figure

plot(xv,yv) % polygon

hold on
```

```
plot(xq(in&~on),yq(in&~on),'r+') % points strictly inside
plot(xq(on),yq(on),'k*') % points on edge
plot(xq(~in),yq(~in),'bo') % points outside
hold off
```



Six points lie inside the polygon. Two points lie on the edge of the polygon. Four points lie outside the polygon.

Input Arguments

xq — x-coordinates of query points

scalar | vector | matrix | multidimensional array

x-coordinates of query points, specified as a scalar, vector, matrix, or multidimensional array.

The size of `xq` must match the size of `yq`.

Data Types: `double | single`

yq — y-coordinates of query points

scalar | vector | matrix | multidimensional array

y-coordinates of query points, specified as a scalar, vector, matrix, or multidimensional array.

The size of `yq` must match the size of `xq`.

Data Types: `double | single`

xv — x-coordinates of polygon vertices

vector

x-coordinates of polygon vertices, specified as a vector.

The size of `xv` must match the size of `yv`.

To specify vertices of multiply connected or disjoint polygons, separate the coordinates for distinct loops with `NaN`. Additionally for multiply connected polygons, you must orient the vertices for external and internal loops in opposite directions.

The polygon cannot be self-intersecting and multiply connected due to the ambiguity associated with self-intersections and loop orientations.

Data Types: `double | single`

yv — y-coordinates of polygon vertices

vector

y-coordinates of polygon vertices, specified as a vector.

The size of `yv` must match the size of `xv`.

To specify vertices of multiply connected or disjoint polygons, separate the coordinates for distinct loops with `NaN`. Additionally for multiply connected polygons, you must orient the vertices for external and internal loops in opposite directions.

The polygon cannot be self-intersecting and multiply connected due to the ambiguity associated with self-intersections and loop orientations.

Data Types: `double | single`

Output Arguments

`in` — Indicator for points inside or on edge of polygon area

logical array

Indicator for the points inside or on the edge of the polygon area, returned as a logical array. `in` is the same size as `xq` and `yq`.

- A logical `1` (`true`) indicates that the corresponding query point is inside the polygonal region or on the edge of the polygon boundary.
- A logical `0` (`false`) indicates that the corresponding query point is outside the polygonal region.

Therefore, you can use `in` to index into `xq` and `yq` to identify query points of interest.

`xq(in)`, `yq(in)`	Query points inside or on the edge of the polygon area
`xq(~in)`, `yq(~in)`	Query points outside the polygonal region

`on` — Indicator for points on edge of polygon area

logical array

Indicator for the points on the edge of the polygon area, returned as a logical array. `on` is the same size as `xq` and `yq`.

- A logical `1` (`true`) indicates that the corresponding query point is on the polygon boundary.
- A logical `0` (`false`) indicates that the corresponding query point is inside or outside the polygon boundary.

Therefore, you can use `on` and `in` to index into `xq` and `yq` identify query points of interest.

`xq(on)`, `yq(on)`	Query points on the polygon boundary

`xq(~on)`, `yq(~on)`	Query points inside or outside the polygon boundary
`xq(in&~on)`, `yq(in&~on)`	Query points strictly inside the polygonal region

See Also

`delaunay`

Introduced before R2006a

input

Request user input

Syntax

```
x = input(prompt)
str = input(prompt,'s')
```

Description

`x = input(prompt)` displays the text in prompt and waits for the user to input a value and press the **Return** key. The user can enter expressions, like `pi/4` or `rand(3)`, and can use variables in the workspace.

- If the user presses the **Return** key without entering anything, then `input` returns an empty matrix.
- If the user enters an invalid expression at the prompt, then MATLAB displays the relevant error message, and then redisplays the prompt.

`str = input(prompt,'s')` returns the entered text as a string, without evaluating the input as an expression.

Examples

Request Numeric Input or Expression

Request a numeric input, and then multiply the input by 10.

```
prompt = 'What is the original value? ';
x = input(prompt)
y = x*10
```

At the prompt, enter a numeric value or array, such as `42`.

```
x =
```

```
    42

y =
   420
```

The `input` function also accepts expressions. For example, rerun the code.

```
prompt = 'What is the original value? ';
x = input(prompt)
y = x*10
```

At the prompt, enter `magic(3)`.

```
x =
     8     1     6
     3     5     7
     4     9     2

y =
    80    10    60
    30    50    70
    40    90    20
```

The expression does not need to return a numeric result. For example:

```
prompt = 'What color is the sun? ';
s = input(prompt)
```

At the prompt, type `upper('yellow')`.

```
s =
YELLOW
```

Request Unprocessed Text Input

Request a simple text response that requires no evaluation.

```
prompt = 'Do you want more? Y/N [Y]: ';
str = input(prompt,'s');
if isempty(str)
    str = 'Y';
end
```

The `input` function returns the text exactly as typed. If the input is empty, this code assigns a default value, `'Y'`, to the output string, `str`.

Input Arguments

`prompt` — Text displayed to the user

string

Text displayed to the user, specified as a string.

To create a prompt that spans several lines, use `'\n'` to indicate each new line. To include a backslash (`'\'`) in the prompt, use `'\\'`.

Output Arguments

`x` — Result calculated from input

array

Result calculated from input, returned as an array. The type and dimensions of the array depend upon the response to the prompt.

`str` — Exact text of input

string

Exact text of the input, returned as a string.

See Also

`ginput` | `inputdlg` | `keyboard` | `menu` | `uicontrol`

Introduced before R2006a

inputdlg

Create dialog box that gathers user input

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,dlg_title)
answer = inputdlg(prompt,dlg_title,num_lines)
answer = inputdlg(prompt,dlg_title,num_lines,defAns)
answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)
```

Description

`answer = inputdlg(prompt)` creates a modal dialog box and returns user input for multiple prompts in the cell array. `prompt` is a cell array containing prompt strings.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in Figure Properties.

`answer = inputdlg(prompt,dlg_title)` `dlg_title` specifies a title for the dialog box.

`answer = inputdlg(prompt,dlg_title,num_lines)` `num_lines` specifies the number of lines for each user-entered value. `num_lines` can be a scalar, column vector, or a m x 2 array.

- If `num_lines` is a scalar, it applies to all prompts.
- If `num_lines` is a column vector, each element specifies the number of lines of input for a prompt.
- If `num_lines` is an array, it must be size m-by-2, where m is the number of prompts on the dialog box. Each row refers to a prompt. The first column specifies the number of lines of input for a prompt. The second column specifies the width of the field in characters.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns)` `defAns` specifies the default value to display for each prompt. `defAns` must contain the same number of elements as `prompt` and all elements must be strings.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)` If `options` is the string `'on'`, the dialog is made resizable in the horizontal direction. If options is a structure, the fields shown in the following table are recognized:

Field	Description
`Resize`	Can be `'on'` or `'off'` (default). If `'on'`, the window is resizable horizontally.
`WindowStyle`	Can be either `'normal'` or `'modal'` (default).
`Interpreter`	Can be either `'none'` (default) or `'tex'`. If the value is `'tex'`, the prompt strings are rendered using LaTeX.

If the user clicks the **Cancel** button to close an `inputdlg` box, the dialog returns an empty cell array:

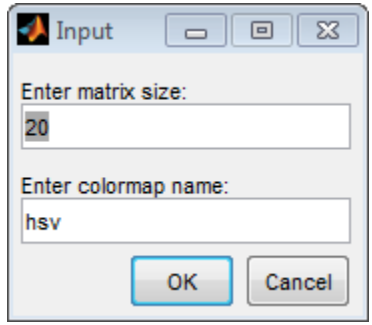
```
answer =
     {}
```

Examples

Example 1

Create a dialog box to input an integer and colormap name. Allow one line for each value.

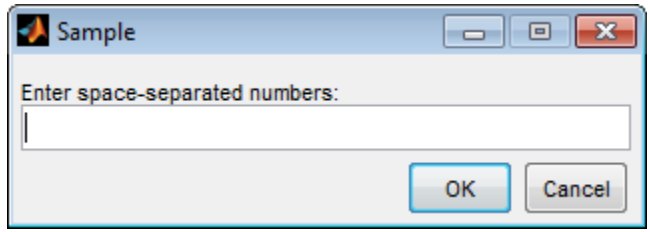
```
prompt = {'Enter matrix size:','Enter colormap name:'};
dlg_title = 'Input';
num_lines = 1;
def = {'20','hsv'};
answer = inputdlg(prompt,dlg_title,num_lines,def);
```



Example 2

Create a dialog box named to accept comma-separated numbers. MATLAB stores accepts the input as a string, so convert the string to numbers using `str2num`.

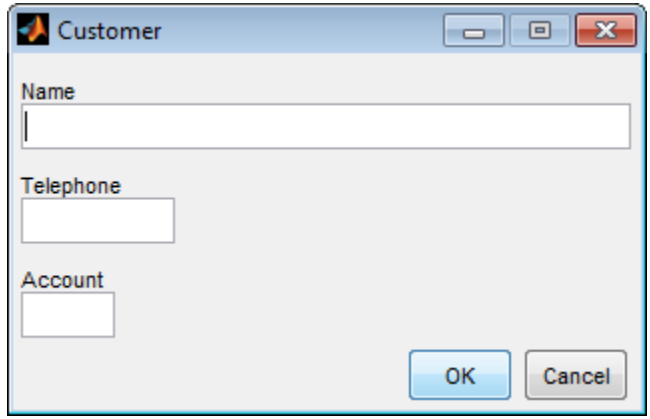
```
x = inputdlg('Enter space-separated numbers:',...
             'Sample', [1 50]);
data = str2num(x{:});
```



Example 3

Create a dialog box to display input fields of different widths.

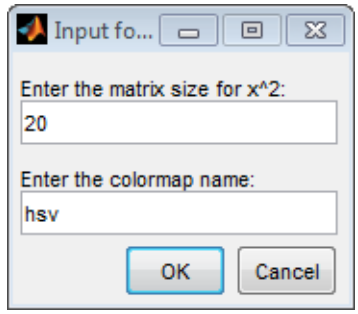
```
x = inputdlg({'Name','Telephone','Account'},...
             'Customer', [1 50; 1 12; 1 7]);
```



Example 4

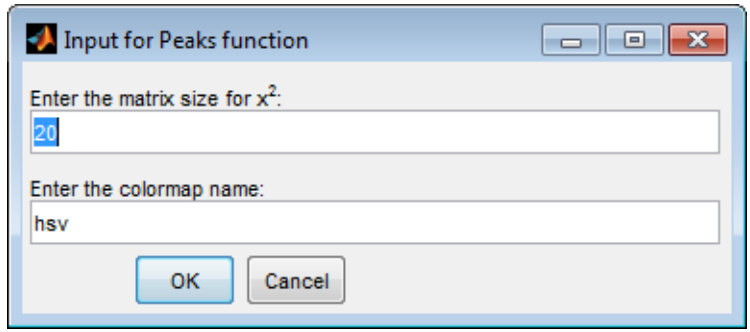
Create a dialog box using the default options. Then, use the options to make it resizable and not modal, and to interpret the text using LaTeX.

```
prompt={'Enter the matrix size for x^2:',...
        'Enter the colormap name:'};
name='Input for Peaks function';
numlines=1;
defaultanswer={'20','hsv'};
answer=inputdlg(prompt,name,numlines,defaultanswer);
```




```
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';
answer=inputdlg(prompt,name,numlines,...
```

```
                defaultanswer,options);
```



More About

Tips

`inputdlg` uses the `uiwait` function to suspend execution until the user responds.

The returned variable `answer` is a cell array containing strings, one string per text entry field, starting from the top of the dialog box.

To convert a member of the cell array to a number, use `str2num`. To do this, you can add the following code to the end of any of the examples below:

```
% Use curly bracket for subscript
[val status] = str2num(answer{1});
if ~status
    % Handle empty value returned
    % for unsuccessful conversion
    % ...
end
% val is a scalar or matrix converted from the first input
```

Users can enter scalar or vector values into `inputdlg` fields; `str2num` converts space- and comma-delimited strings into row vectors, and semicolon-delimited strings into column vectors. For example, if `answer{1}` contains `'1 2 3;4 -5 6+7i'`, the conversion produces:

```
val = str2num(answer{1})
val =
   1.0000     2.0000     3.0000
```

```
   4.0000    -5.0000     6.0000 + 7.0000i
```

See Also

dialog | errordlg | helpdlg | listdlg | msgbox | questdlg | warndlg | input | figure | str2num | uiwait | uiresume

Introduced before R2006a

inputname

Variable name of function input

Syntax

```
inputname(argnum)
```

Description

This command can be used only inside the body of a function.

`inputname(argnum)` returns the workspace variable name corresponding to the argument number *`argnum`*. If the input argument has no name (for example, if it is an expression instead of a variable), the `inputname` command returns the empty string (`''`).

Examples

Suppose the function `myfun.m` is defined as

```
function c = myfun(a,b)
fprintf('First calling variable is "%s"\n.', inputname(1))
```

Then

```
x = 5;  y = 3;  myfun(x,y)
```

produces

```
First calling variable is "x".
```

But

```
myfun(pi+1, pi-1)
```

produces

```
First calling variable is "".
```

See Also

nargin | nargout | narginchk

Introduced before R2006a

inputParser class

Parse function inputs

Description

The `inputParser` object allows you to manage inputs to a function by creating an input scheme. To check the input, you can define validation functions for required arguments, optional arguments, and name-value pair arguments. Optionally, you can set properties to adjust the parsing behavior, such as handling case sensitivity, structure array inputs, and inputs that are not in the input scheme.

After calling the `parse` method to parse the inputs, the `inputParser` saves names and values of inputs that match the input scheme (stored in `Results`), names of inputs that are not passed to the function and, therefore, are assigned default values (stored in `UsingDefaults`), and names and values of inputs that do not match the input scheme (stored in `Unmatched`).

Construction

`p = inputParser` creates `inputParser` object `p`.

Properties

`CaseSensitive`

Scalar logical value that indicates whether to match case when checking argument names.

Possible values:

`false` (0)	Names are not sensitive to case: `'a'` matches `'A'`.
`true` (1)	Names are case sensitive: `'a'` does not match `'A'`.

Default: `false`

FunctionName

String that specifies the name of the function to include in error messages. By specifying the `FunctionName`, the `parse` method of the `inputParser` will throw error messages as if it were that function. This allows the error to be attributed to the correct function and allows easy access to function documentation through the error message.

Default: The default value is an empty string, `''`.

KeepUnmatched

Scalar logical value that indicates how to handle parameter name and value inputs that are not in the input scheme.

Possible values:

`false` (0)	Throw an error whenever inputs are not in the scheme.
`true` (1)	Store the parameter names and values of unmatched inputs in the `Unmatched` property of the `inputParser` object, and suppress the error.

Default: `false`

PartialMatching

Scalar logical value that indicates whether partial matching of parameter names will be accepted. Partial parameter matching is supported by the `addParameter` method. If the value of `StructExpand` is `true`, then `PartialMatching` is not supported for structure field names corresponding to input parameter names.

Possible values:

`true` (1)	Inputs that are leading substrings of parameter names will be accepted and the value matched to that parameter. If there are multiple possible matches to the input string, MATLAB throws an error.
`false` (0)	Input names are required to match a parameter name exactly (with respect to the `CaseSensitive` property.)

Default: `true`

StructExpand

Scalar logical value that specifies whether to interpret a structure array as a single input or as a set of parameter name and value pairs.

`true` (1)	Expand structures into separate inputs. Each field name corresponds to an input parameter name.
`false` (0)	Regard a structure array as a single input argument.

Default: `true`

Read Only Properties

Parameters

Cell array of strings that contains the names of arguments currently defined in the input scheme.

Each method that adds an input argument to the scheme (`addRequired`, `addOptional`, `addParameter`) updates the `Parameters` property.

Results

Structure containing names and values of inputs that match the function input scheme, populated by the `parse` method.

Each field of the `Results` structure corresponds to the name of an input.

Unmatched

Structure array containing the names and values of inputs that do not match the function input scheme, populated by the `parse` method.

If `KeepUnmatched` is `false` (default) or all inputs match the scheme, then `Unmatched` is a 1-by-1 structure with no fields. Otherwise, each field of the structure corresponds to the name of an input that did not match the scheme.

UsingDefaults

Cell array containing the names of inputs not passed explicitly to the function and assigned default values, populated by the `parse` method.

Methods

addOptional	Add optional positional argument to input parser scheme
addParameter	Add optional parameter name-value pair argument to input parser scheme
addParamValue	(Not recommended) Add parameter name and value argument to Input Parser scheme
addRequired	Add required positional argument to input parser scheme
parse	Parse function inputs

You can define your scheme by calling `addRequired`, `addOptional`, and `addParameter` in any order, but when you call your function that uses the input parser, you should pass in required inputs first, followed by any optional positional inputs, and, finally, any name-value pairs.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

The `copy` method creates another scheme with the same properties as an existing `inputParser` object:

```
pNew = copy(pOld);
```

Examples

Input Validation

Check the validity of required and optional function inputs.

Create a custom function with required and optional inputs in the file `findArea.m`.

```
function a = findArea(width,varargin)
   p = inputParser;
   defaultHeight = 1;
   defaultUnits = 'inches';
   defaultShape = 'rectangle';
   expectedShapes = {'square','rectangle','parallelogram'};

   addRequired(p,'width',@isnumeric);
   addOptional(p,'height',defaultHeight,@isnumeric);
   addParameter(p,'units',defaultUnits);
   addParameter(p,'shape',defaultShape,...
                @(x) any(validatestring(x,expectedShapes)));

   parse(p,width,varargin{:});
   a = p.Results.width .* p.Results.height;
```

The input parser checks whether `width` and `height` are numeric, and whether the `shape` matches a string in cell array `expectedShapes`. `@` indicates a function handle, and the syntax `@(x)` creates an anonymous function with input `x`.

Call the function with inputs that do not match the scheme. For example, specify a nonnumeric value for the `width` input:

```
findArea('text')
```

```
Error using findArea (line 14)
The value of 'width' is invalid. It must satisfy the function: isnumeric.
```

Specify an unsupported value for `shape`:

```
findArea(4,'shape','circle')
```

```
Error using findArea (line 14)
The value of 'shape' is invalid. Expected input to match one of these strings:

square, rectangle, parallelogram

The input, ''circle'', did not match any of the valid strings.
```

Extra Parameter Value Inputs

Store parameter name and value inputs that are not in the input scheme instead of throwing an error.

```
default = 0;
value = 1;
```

```
p = inputParser;
p.KeepUnmatched = true;
addOptional(p,'expectedInputName',default)
parse(p,'extraInput',value);
```

View the unmatched parameter name and value:

```
p.Unmatched
```

```
ans =

    extraInput: 1
```

Case Sensitivity

Enforce case sensitivity when checking function inputs.

```
p = inputParser;
p.CaseSensitive = true;
defaultValue = 0;
addParameter(p,'InputName',defaultValue)

parse(p,'inputname',10)
```

```
'inputname' is not a recognized parameter. For a list of valid name-value pair arguments, see the documentation for this f
```

Structure Array Inputs

Parse structure array inputs with the `StructExpand` property set to `true` (default) or `false`.

Expand a structure array input into parameter name and value pairs using the default `true` value of the `StructExpand` property.

```
s.input1 = 10;
s.input2 = 20;
default = 0;

p = inputParser;
addParameter(p,'input1',default)
addParameter(p,'input2',default)
parse(p,s)
```

```
p.Results
```

```
ans =

    input1: 10
    input2: 20
```

Explicitly specifying a parameter name and value pair overrides values in the structure.

```
parse(p,s,'input2',300)
p.Results
```

```
ans =

    input1: 10
    input2: 300
```

Accept a structure array input as a single argument by setting the `StructExpand` property to `false`.

```
s2.first = 1;
s2.random = rand(3,4,2);
s2.mytext = 'some text';

p = inputParser;
p.StructExpand = false;
addRequired(p,'structInput')
parse(p,s2)

results = p.Results
fieldList = fieldnames(p.Results.structInput)
```

```
results =

    structInput: [1x1 struct]


fieldList =
```

```
    'first'
    'random'
    'mytext'
```

Parse Inputs Using `validateattributes`

Create a function that parses information about people and, if parsing passes, adds the information to a cell array.

Create a function, `addPerson`, that sets up an `inputParser` scheme using `validateAttributes`. The function should accept the list of people, modify the list if necessary and return the list. Use a persistent `inputParser` to avoid construction of a new object with every function call. If this is the first call to the function, add a titles row to the cell array.

```
function mlist = addPerson(mlist,varargin)

persistent p
if isempty(p)
    p = inputParser;
    p.FunctionName = 'addPerson';
    addRequired(p,'name',@(x)validateattributes(x,{'char'},...
        {'nonempty'}))
    addRequired(p,'id',@(x)validateattributes(x,{'numeric'},...
        {'nonempty','integer','positive'}))
    addOptional(p,'birthyear',9999,@(x)validateattributes(x,...
        {'numeric'},{'nonempty'}))
    addParameter(p,'nickname','-',@(x)validateattributes(x,...
        {'char'},{'nonempty'}))
    addParameter(p,'favColor','-',@(x)validateattributes(x,...
        {'char'},{'nonempty'}))
end

parse(p,varargin{:})

if isempty(mlist)
    mlist = fieldnames(p.Results)';
end
mlist = [mlist; struct2cell(p.Results)'];

end
```

Create an empty list, and add a person to it.

```
pList = {};
pList = addPerson(pList,78,'Joe');

Error using addPerson
The value of 'name' is invalid. Expected input to be one of these types:

char

Instead its type was double.
Error in addPerson (line 17)
```

```
parse(p,mlist,name,id,varargin{:});
```

The parsing failed because the function received the inputs in the incorrect order and tried to assign `name` a value of `78`. This entry was not added to `pList`.

Add several more people to the list.

```
pList = addPerson(pList,'Joe',78);
pList = addPerson(pList,'Mary',3,1942,'favColor','red');
pList = addPerson(pList,'James',182,1970,'nickname','Jimmy')
```

```
pList =

    'birthyear'    'favColor'    'id'     'name'     'nickname'
    [     9999]    '-'           [ 78]    'Joe'      '-'
    [     1942]    'red'         [  3]    'Mary'     '-'
    [     1970]    '-'           [182]    'James'    'Jimmy'
```

See Also

`nargin` | `narginchk` | `validateattributes` | `validatestring` | `varargin`

More About

- “Input Parser Validation Functions”

inspect

Open Property Inspector

Syntax

```
inspect
inspect(h)
inspect([h1,h2,...])
```

Description

`inspect` creates a separate Property Inspector window to enable the display and modification of the properties of any object you select in the figure window or Layout Editor. If no object is selected, the Property Inspector is blank.

`inspect(h)` creates a Property Inspector window for the object whose handle is `h`.

`inspect([h1,h2,...])` displays properties that objects `h1` and `h2` have in common, or a blank window if there are no such properties; any number of objects can be inspected and edited in this way (for example, handles returned by the `bar` command).

The Property Inspector has the following behaviors:

- Only one Property Inspector window is active at any given time; when you inspect a new object, its properties replace those of the object last inspected.
- When the Property Inspector is open and plot edit mode is on, clicking any object in the figure window displays the properties of that object (or set of objects) in the Property Inspector.
- When you select and inspect two or more objects of different types, the Property Inspector only shows the properties that all objects have in common.
- To change the value of any property, click on the property name shown at the left side of the window, and then enter the new value in the field at the right.

The Property Inspector provides two different views:

- List view — properties are ordered alphabetically (default); this is the only view available for annotation objects.
- Group view — properties are grouped under classified headings (Handle Graphics objects only)

To view alphabetically, click the "AZ" Icon  in the Property Inspector toolbar. To see properties in groups, click

the "++" icon  . When properties are grouped, the "-" and "+" icons are enabled; click  to expand all categories and click  to collapse all categories. You can also expand and collapse individual categories by clicking on the "+" next to the category name. Some properties expand and collapse

---

Notes To see a complete description of any property, right-click on its name or value and select **What's This**; a help window opens that displays the reference page entry for it.

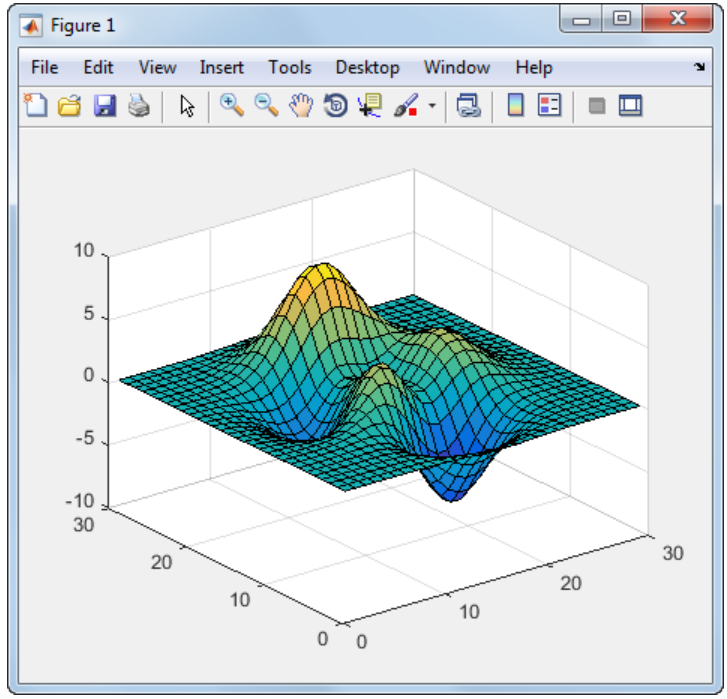
The Property Inspector displays most, but not all, properties of Handle Graphics objects. For example, the `parent` and `children` of HG objects are not shown.
`inspect h` displays a Property Inspector window that enables modification of the string `'h'`, not the object whose handle is `h`.
If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

Examples

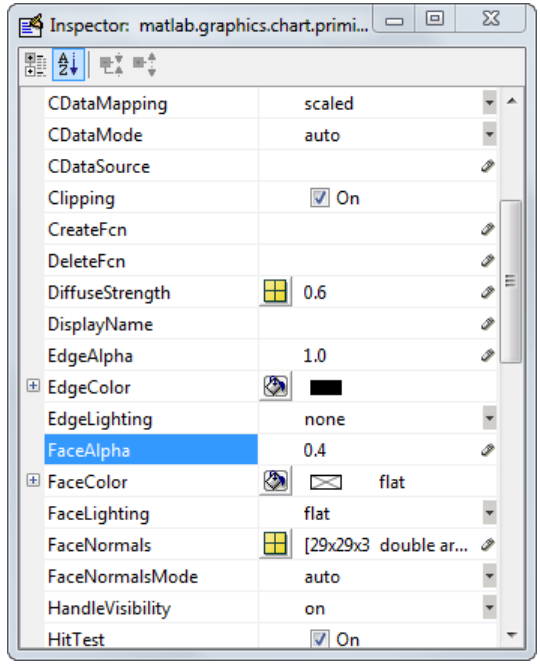
Example 1

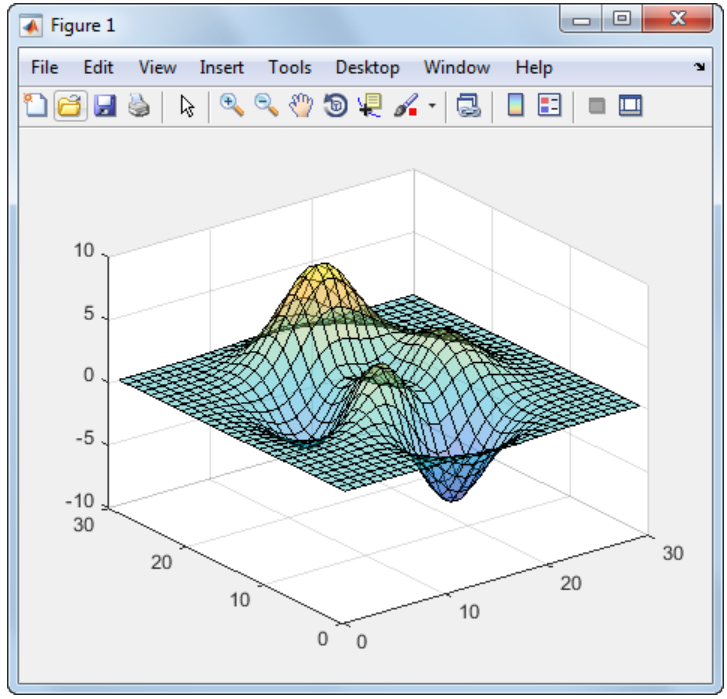
Create a surface mesh plot and view its properties with the Property Inspector.

```
Z = peaks(30);
f = surf(Z);
inspect(f);
```



In the Property Inspector, change the `FaceAlpha` property from `1.0` to `0.4`. Setting this value in the Property Inspector is equivalent to the command, `f.FaceAlpha = 0.4`, which changes the transparency of the surface faces to 40%.

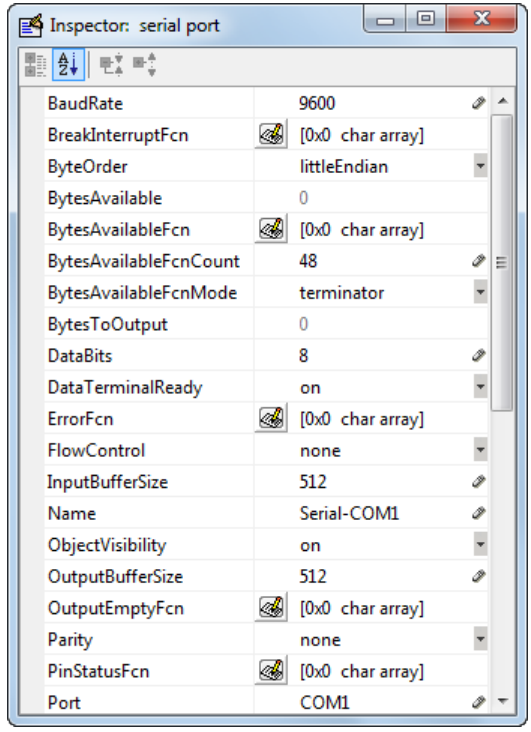

Inspector: matlab.graphics.chart.primi...
CDataMapping
scaled
CDataMode
auto
CDataSource
Clipping
On
CreateFcn
DeleteFcn
DiffuseStrength
0.6
DisplayName
EdgeAlpha
1.0
EdgeColor
EdgeLighting
none
FaceAlpha
0.4
FaceColor
flat
FaceLighting
flat
FaceNormals
[29x29x3 double ar...
FaceNormalsMode
auto
HandleVisibility
on
HitTest
On



Example 2

Create a serial port object for `COM1` on a Windows platform and use the Property Inspector to peruse its properties:

```
s = serial('COM1');
inspect(s);
```



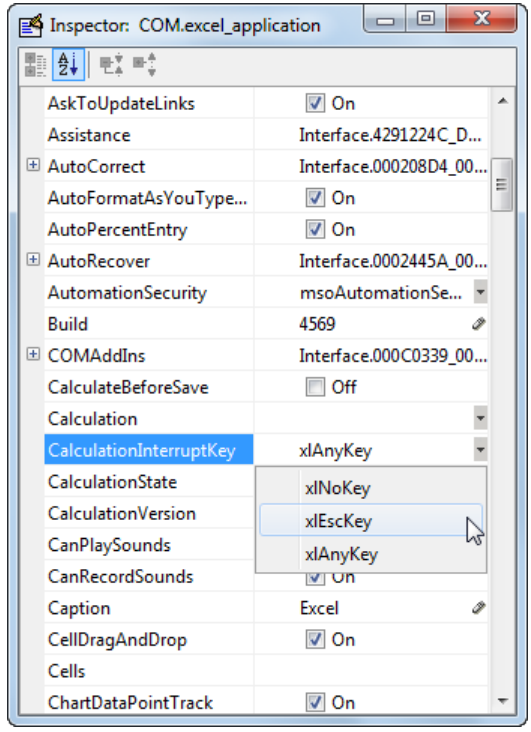
Because COM objects do not define property groupings, the Property Inspector enables only the alphabetical list view of COM object properties.

Example 3

Create a COM Excel server and open a Property Inspector window with `inspect`.

```
h = actxserver('excel.application');
inspect(h);
```

Scroll down until you see the `CalculationInterruptKey` property, which by default is `xlAnyKey`. Click on the down-arrow in the right margin of the Property Inspector, and then select `xlEscKey` from the drop-down menu.



Check this property in the MATLAB Command Window using `get` to confirm that the property value changed.

```
get(h,'CalculationInterruptKey')

ans =
xlEscKey
```

More About

- "Access Property Values"

See Also

`addproperty` | `deleteproperty` | `isprop`

Introduced before R2006a

instrcallback

Event information when event occurs

Syntax

```
instrcallback(obj,event)
```

Description

`instrcallback(obj,event)` displays a message that contains the event type, `event`, the time the event occurred, and the name of the serial port object, `obj`, that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.

Examples

The following example creates the serial port object, `s`, on a Windows platform. It configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

More About

Tips

Use `instrcallback` as a template to create callback functions that suit your specific application needs.

Introduced before R2006a

instrfind

Read serial port objects from memory to MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Description

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('PropertyName',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'PropertyName',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

Examples

Suppose you create the following two serial port objects on a Windows platform.

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'BaudRate',4800)
fopen([s1 s2])
```

You can use `instrfind` to return serial port objects based on property values.

```
out1 = instrfind('Port','COM1');
```

```
out2 = instrfind({'Port','BaudRate'},{'COM2',4800});
```

You can also use `instrfind` to return cleared serial port objects to the MATLAB workspace.

```
clear s1 s2
newobjs = instrfind
```

```
   Instrument Object Array
   Index:   Type:          Status:     Name:
   1        serial         open        Serial-COM1
   2        serial         open        Serial-COM2
```

To close both `s1` and `s2`

```
fclose(newobjs)
```

More About

Tips

Refer to "Displaying Property Names and Property Values" for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

See Also

`clear` | `get`

Introduced before R2006a

instrfindall

Find visible and hidden serial port objects

Syntax

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

Description

`out = instrfindall` finds all serial port objects, regardless of the value of the object's `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'P1',V1,...)` restricts the search for objects with matching property name/value pairs to the serial port objects listed in `objs`.

Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

Examples

Suppose you create the following serial port objects on a Windows platform:

```
s1 = serial('COM1');
s2 = serial('COM2');
```

```
set(s2,'ObjectVisibility','off')
```

Because object `s2` has its `ObjectVisibility` set to `'off'`, it is not visible to commands like `instrfind`:

```
instrfind
```

```
   Serial Port Object : Serial-COM1
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall
```

```
 Instrument Object Array
 Index:   Type:          Status:     Name:
 1        serial         closed      Serial-COM1
 2        serial         closed      Serial-COM2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress};
vals = {2,0};
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

More About

Tips

`instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that returned by the `get` function. For example, if `get` returns the `Name` property value as `'MyObject'`, `instrfindall` will not find an object with a `Name` property value of `'myobject'`. However, this is not the case for properties that have a

finite set of string values. For example, `instrfindall` will find an object with a `Parity` property value of `'Even'` or `'even'`.

See Also

`get` | `instrfind` | `ObjectVisibility`

int2str

Convert integer to string

Syntax

```
str = int2str(N)
```

Description

`str = int2str(N)` converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.

Examples

`int2str(2+3)` is the string `'5'`.

One way to label a plot is

```
title(['case number ' int2str(n)])
```

For matrix or vector inputs, `int2str` returns a string matrix:

```
int2str(eye(3))

ans =

1  0  0
0  1  0
0  0  1
```

See Also

`fprintf` | `num2str` | `sprintf` | `cast`

Introduced before R2006a

int8

Convert to 8-bit signed integer

Syntax

```
intArray = int8(array)
```

Description

`intArray = int8(array)` converts the elements of an array into signed 8-bit (1-byte) integers of class `int8`.

Input Arguments

`array`

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int8`, the `int8` function has no effect.

Output Arguments

`intArray`

Array of class `int8`. Values range from -2^7 to $2^7 - 1$.

The `int8` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int8(2^7)   % 2^7 = 128
```

returns

```
ans =
    127
```

Examples

Convert a `double` array to `int8`:

```
mydata = int8(magic(10));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = int8(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int8');   % Preferred
```

See Also

`double` | `single` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `intmax` | `intmin`

Introduced before R2006a

int16

Convert to 16-bit signed integer

Syntax

```
intArray = int16(array)
```

Description

`intArray = int16(array)` converts the elements of an array into signed 16-bit (2-byte) integers of class `int16`.

Input Arguments

`array`

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int16`, the `int16` function has no effect.

Output Arguments

`intArray`

Array of class `int16`. Values range from -2^{15} to $2^{15} - 1$.

The `int16` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int16(2^15)   % 2^15 = 32768
```

returns

```
ans =
    32767
```

Examples

Convert a `double` array to `int16`:

```
mydata = int16(magic(100));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = int16(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int16');   % Preferred
```

See Also

`double` | `single` | `int8` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `intmax` | `intmin`

Introduced before R2006a

int32

Convert to 32-bit signed integer

Syntax

```
intArray = int32(array)
```

Description

`intArray = int32(array)` converts the elements of an array into signed 32-bit (4-byte) integers of class `int32`.

Input Arguments

`array`

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int32`, the `int32` function has no effect.

Output Arguments

`intArray`

Array of class `int32`. Values range from -2^{31} to $2^{31} - 1$.

The `int32` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int32(2^31)    % 2^31 = 2147483648
```

returns

```
ans =
    2147483647
```

Examples

Convert a `double` array to `int32`:

```
mydata = int32(magic(1000));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = int32(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int32');   % Preferred
```

See Also

`double` | `single` | `int8` | `int16` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `intmax` | `intmin`

Introduced before R2006a

int64

Convert to 64-bit signed integer

Syntax

```
intArray = int64(array)
```

Description

`intArray = int64(array)` converts the elements of an array into signed 64-bit (8-byte) integers of class `int64`.

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int64`, the `int64` function has no effect.

Output Arguments

intArray

Array of class `int64`. Values range from -2^{63} to $2^{63} - 1$.

The `int64` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int64(2^63)   % 2^63 = 9223372036854775808
```

returns

```
ans =
    9223372036854775807
```

Examples

Convert a literal value to `int64`:

```
x = int64(9007199254740993);
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = int64(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int64');   % Preferred
```

More About

Tips

Double-precision floating-point numbers have only 52 bits in the mantissa. Therefore, `double` values cannot represent all integers greater than 2^{53} correctly. Before performing arithmetic operations on values larger than 2^{53} in magnitude, convert the values to 64-bit integers. For example,

```
x = int64(2^53+1);     % Floating-point arithmetic, loses precision
```

is not as accurate as the 64-bit integer arithmetic operation:

```
x = int64(2^53) + 1;   % Preferred
```

See Also

`double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `uint64` | `intmax` | `intmin`

Introduced before R2006a

integral

Numerical integration

Syntax

```
q = integral(fun,xmin,xmax)
q = integral(fun,xmin,xmax,Name,Value)
```

Description

`q = integral(fun,xmin,xmax)` numerically integrates function `fun` from `xmin` to `xmax` using global adaptive quadrature and default error tolerances.

`q = integral(fun,xmin,xmax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments. For example, specify `'WayPoints'` followed by a vector of real or complex numbers to indicate specific points for the integrator to use.

Examples

Improper Integral

Create the function $f(x) = e^{-x^2}(\ln x)^2$.

```
fun = @(x) exp(-x.^2).*log(x).^2;
```

Evaluate the integral from `x=0` to `x=Inf`.

```
q = integral(fun,0,Inf)
```

```
q =

    1.9475
```

Parameterized Function

Create the function $f(x) = 1/(x^3 - 2x - c)$ with one parameter, c.

```
fun = @(x,c) 1./(x.^3-2*x-c);
```

Evaluate the integral from `x=0` to `x=2` at `c=5`.

```
q = integral(@(x)fun(x,5),0,2)
```

```
q =

   -0.4605
```

Singularity at Lower Limit

Create the function $f(x) = \ln(x)$.

```
fun = @(x)log(x);
```

Evaluate the integral from `x=0` to `x=1` with the default error tolerances.

```
format long
q1 = integral(fun,0,1)
```

```
q1 =

  -1.000000010959678
```

Evaluate the integral again, specifying 12 decimal places of accuracy.

```
q2 = integral(fun,0,1,'RelTol',0,'AbsTol',1e-12)
```

```
q2 =

  -1.000000000000010
```

Complex Contour Integration Using Waypoints

Create the function $f(z) = 1/(2z - 1)$.

```
fun = @(z) 1./(2*z-1);
```

Integrate in the complex plane over the triangular path from `0` to `1+1i` to `1-1i` to `0` by specifying waypoints.

```
q = integral(fun,0,0,'Waypoints',[1+1i,1-1i])
```

```
q =
```

```
        0 - 3.1416i
```

Vector-Valued Function

Create the vector-valued function $f(x) = [\sin x, \sin 2x, \sin 3x, \sin 4x, \sin 5x]$ and integrate from `x=0` to `x=1`. Specify `'ArrayValued',true` to evaluate the integral of an array-valued or vector-valued function.

```
fun = @(x)sin((1:5)*x);
q = integral(fun,0,1,'ArrayValued',true)
```

```
q =

    0.4597    0.7081    0.6633    0.4134    0.1433
```

Improper Integral of Oscillatory Function

Create the function $f(x) = x^5 e^{-x} \sin x$.

```
fun = @(x)x.^5.*exp(-x).*sin(x);
```

Evaluate the integral from `x=0` to `x=Inf` , adjusting the absolute and relative tolerances.

```
format long
q = integral(fun,0,Inf,'RelTol',1e-8,'AbsTol',1e-13)
```

```
q =

 -14.999999999998364
```

Input Arguments

`fun` — Integrand

function handle

Integrand, specified as a function handle, which defines the function to be integrated from `xmin` to `xmax`.

For scalar-valued problems, the function `y = fun(x)` must accept a vector argument, `x`, and return a vector result, `y`. This generally means that `fun` must use array operators

instead of matrix operators. For example, use `.*` (times) rather than `*` (mtimes). If you set the `'ArrayValued'` option to `true`, then `fun` must accept a scalar and return an array of fixed size.

`xmin` — Lower limit of *x*

real number | complex number

Lower limit of *x*, specified as a real (finite or infinite) scalar value or a complex (finite) scalar value. If either `xmin` or `xmax` are complex, then `integral` approximates the path integral from `xmin` to `xmax` over a straight line path.

Data Types: `double | single`
Complex Number Support: Yes

`xmax` — Upper limit of *x*

real number | complex number

Upper limit of *x*, specified as a real number (finite or infinite) or a complex number (finite). If either `xmin` or `xmax` are complex, `integral` approximates the path integral from `xmin` to `xmax` over a straight line path.

Data Types: `double | single`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

`'AbsTol'` — Absolute error tolerance

nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note: `AbsTol` and `RelTol` work together. `integral` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-3997 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types: `single` | `double`

`'RelTol'` — Relative error tolerance

nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note: `RelTol` and `AbsTol` work together. `integral` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the Tips section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types: `single` | `double`

`'ArrayValued'` — Array-valued function flag

`false` (default) | `true` | `0` | `1`

Array-valued function flag, specified as the comma-separated pair consisting of `'ArrayValued'` and either `false`, `true`, `0`, or `1`. Set this flag to `true` to indicate that `fun` is a function that accepts a scalar input and returns a vector, matrix, or N-D array output.

The default value of `'false'` indicates that `fun` is a function that accepts a vector input and returns a vector output.

Example: `'ArrayValued',true` indicates that the integrand is an array-valued function.

`'Waypoints'` — Integration waypoints

vector

Integration waypoints, specified as the comma-separated pair consisting of `'Waypoints'` and a vector of real or complex numbers. Use waypoints to indicate any points in the integration interval that you would like the integrator to use. You can use waypoints to integrate efficiently across discontinuities of the integrand. Specify the locations of the discontinuities in the vector you supply.

You can specify waypoints when you want to perform complex contour integration. If `xmin`, `xmax`, or any entry of the waypoints vector is complex, the integration is performed over a sequence of straight line paths in the complex plane.

Example: `'Waypoints',[1+1i,1-1i]` specifies two complex waypoints along the interval of integration.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Tips

- Do not use waypoints to specify singularities. Instead, split the interval and add the results of separate integrations with the singularities at the endpoints.
- The `integral` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.
- If you specify a complex value for `xmin`, `xmax`, or any waypoint, all of your limits and waypoints must be finite.
- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you might need to specify larger absolute and relative error tolerances.

- “Parameterizing Functions”

See Also

`function_handle` | `integral2` | `integral3` | `trapz`

Introduced in R2012a

integral2

Numerically evaluate double integral

Syntax

```
q = integral2(fun,xmin,xmax,ymin,ymax)
q = integral2(fun,xmin,xmax,ymin,ymax,Name,Value)
```

Description

`q = integral2(fun,xmin,xmax,ymin,ymax)` approximates the integral of the function `z = fun(x,y)` over the planar region `xmin` ≤ `x` ≤ `xmax` and `ymin(x)` ≤ `y` ≤ `ymax(x)`.

`q = integral2(fun,xmin,xmax,ymin,ymax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Examples

Integrate Triangular Region with Singularity at the Boundary

The function

$$f(x,y) = \frac{1}{(\sqrt{x+y})\ (1+x+y)}$$

is undefined when x and y are zero. `integral2` performs best when singularities are on the integration boundary.

Create the anonymous function.

```
fun = @(x,y) 1./( sqrt(x + y) .* (1 + x + y).^2 )
```

Integrate over the triangular region bounded by $0 \le x \le 1$ and $0 \le y \le 1 - x$.

```
ymax = @(x) 1 - x
q = integral2(fun,0,1,0,ymax)

q =

    0.2854
```

Evaluate Double Integral in Polar Coordinates

Define the function

$$f(\theta,r) = \frac{r}{\sqrt{r\cos\theta + r\sin\theta}\ (1 + r\cos\theta + r\sin\theta)^2}$$

```
fun = @(x,y) 1./( sqrt(x + y) .* (1 + x + y).^2 );
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
```

Define a function for the upper limit of r.

```
rmax = @(theta) 1./(sin(theta) + cos(theta));
```

Integrate over the region bounded by $0 \le \theta \le \pi/2$ and $0 \le r \le$ rmax.

```
q = integral2(polarfun,0,pi/2,0,rmax)

q =

    0.2854
```

Evaluate Double Integral of Parameterized Function with Specific Method and Error Tolerance

Create the anonymous parameterized function $f(x,y) = ax^2 + by^2$ with parameters a=3 and b=5.

```
a = 3; b = 5;
fun = @(x,y) a*x.^2 + b*y.^2;
```

Evaluate the integral over the region $0 \le x \le 5$ and $-5 \le y \le 0$. Specify the `'iterated'` method and approximately 10 significant digits of accuracy.

```
format long
q = integral2(fun,0,5,-5,0,'Method','iterated',...
```

```
'AbsTol',0,'RelTol',1e-10)

q =

     1.666666666666666e+03
```

Input Arguments

`fun` — Integrand
function handle

Integrand, specified as a function handle, defines the function to be integrated over the planar region `xmin` ≤ `x` ≤ `xmax` and `ymin`(`x`) ≤ `y` ≤ `ymax`(`x`). The function `fun` must accept two arrays of the same size and return an array of corresponding values. It must perform element-wise operations.

Data Types: `function_handle`

`xmin` — Lower limit of *x*
real number

Lower limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types: `double` | `single`

`xmax` — Upper limit of *x*
real number

Upper limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types: `double` | `single`

`ymin` — Lower limit of *y*
real number | function handle

Lower limit of *y*, specified as a real scalar value that is either finite or infinite. You can specify `ymin` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

`ymax` — Upper limit of *y*
real number | function handle

Upper limit of y, specified as a real scalar value that is either finite or infinite. You also can specify `ymax` to be a function handle (a function of x) when integrating over a nonrectangular region.

Data Types: `double | function_handle | single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

`'AbsTol'` — Absolute error tolerance

nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral2` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral2` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note: `AbsTol` and `RelTol` work together. `integral2` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-4004 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types: `double | single`

`'RelTol'` — Relative error tolerance

nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral2` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral

and Q is the (unknown) exact value. `integral2` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note: `RelTol` and `AbsTol` work together. `integral2` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-4004 section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types: `double` | `single`

`'Method'` — Integration method

`'auto'` (default) | `'tiled'` | `'iterated'`

Integration method, specified as the comma-separated pair consisting of `'Method'` and one of the methods described below.

Integration Method	Description
`'auto'`	For most cases, `integral2` uses the `'tiled'` method. It uses the `'iterated'` method when any of the integration limits are infinite. This is the default method.
`'tiled'`	`integral2` transforms the region of integration to a rectangular shape and subdivides it into smaller rectangular regions as needed. The integration limits must be finite.
`'iterated'`	`integral2` calls `integral` to perform an iterated integral. The outer integral is evaluated over `xmin` ≤ `x` ≤ `xmax`. The inner integral is evaluated over `ymin(x)` ≤ `y` ≤ `ymax(x)`. The integration limits can be infinite.

Example: `'Method','tiled'` specifies the tiled integration method.

Output Arguments

`q` — Computed integral

numeric value

Computed integral of `fun(x,y)` over the specified region, returned as a numeric value.

More About

Tips

- The `integral2` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.
- The `'iterated'` method can be more effective when your function has discontinuities within the integration region. However, the best performance and accuracy occurs when you split the integral at the points of discontinuity and sum the results of multiple integrations.
- When integrating over nonrectangular regions, the best performance and accuracy occurs when `ymin`, `ymax`, (or both) are function handles. Avoid setting integrand function values to zero to integrate over a nonrectangular region. If you must do this, specify `'iterated'` method.
- Use the `'iterated'` method when `ymin`, `ymax`, (or both) are unbounded functions.
- When paramaterizing anonymous functions, be aware that parameter values persist for the life of the function handle. For example, the function `fun = @(x,y) x + y + a` uses the value of `a` at the time `fun` was created. If you later decide to change the value of `a`, you must redefine the anonymous function with the new value.
- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you might need to specify larger absolute and relative error tolerances.

- “Parameterizing Functions”

See Also

`function_handle` | `integral` | `integral3` | `trapz`

integral3

Numerically evaluate triple integral

Syntax

```
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax)
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,Name,Value)
```

Description

`q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax)` approximates the integral of the function `z = fun(x,y,z)` over the region `xmin` $\le$ `x` $\le$ `xmax`, `ymin(x)` $\le$ `y` $\le$ `ymax(x)` and `zmin(x,y)` $\le$ `z` $\le$ `zmax(x,y)`.

`q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Examples

Triple Integral with Finite Limits

Define the anonymous function $f(x,y,z) = y \sin x + z \cos x$.

```
fun = @(x,y,z) y.*sin(x)+z.*cos(x)
```

Integrate over the region $0 \le x \le \pi$, $0 \le y \le 1$, and $-1 \le z \le 1$.

```
q = integral3(fun,0,pi,0,1,-1,1)
```

```
q =

    2.0000
```

Integral Over the Unit Sphere in Cartesian Coordinates

Define the anonymous function $f(x,y,z) = x \cos y + x^2 \cos z$.

```
fun = @(x,y,z) x.*cos(y) + x.^2.*cos(z)
```

Define the limits of integration.

```
xmin = -1;
xmax = 1;
ymin = @(x)-sqrt(1 - x.^2);
ymax = @(x) sqrt(1 - x.^2);
zmin = @(x,y)-sqrt(1 - x.^2 - y.^2);
zmax = @(x,y) sqrt(1 - x.^2 - y.^2);
```

Evaluate the definite integral with the `'tiled'` method.

```
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,'Method','tiled')
```

```
q =

    0.7796
```

Evaluate Improper Triple Integral of Parameterized Function

Define the anonymous parameterized function $f(x,y,z) = 10/(x^2 + y^2 + z^2 + a)$.

```
a = 2;
f = @(x,y,z) 10./(x.^2 + y.^2 + z.^2 + a);
```

Evaluate the triple integral over the region `-Inf` $\le x \le 0$, $-100 \le y \le 0$, and $-100 \le z \le 0$.

```
format long
q1 = integral3(f,-Inf,0,-100,0,-100,0)
```

```
q1 =

     2.734244598320928e+03
```

Evaluate the integral again and specify accuracy to approximately 9 significant digits.

```
q2 = integral3(f,-Inf,0,-100,0,-100,0,'AbsTol', 0,'RelTol',1e-9)
```

```
q2 =

     2.734244599944285e+03
```

Input Arguments

`fun` — Integrand

function handle

Integrand, specified as a function handle, defines the function to be integrated over the region `xmin` ≤ `x` ≤ `xmax`, `ymin`(`x`) ≤ `y` ≤ `ymax`(`x`), and `zmin`(`x`,`y`) ≤ `z` ≤ `zmax`(`x`,`y`). The function `fun` must accept three arrays of the same size and return an array of corresponding values. It must perform element-wise operations.

Data Types: `function_handle`

`xmin` — Lower limit of *x*

real number

Lower limit of *x*, specified as a real scalar value that is either finite or infinite.

Example:

Data Types: `double` | `single`

`xmax` — Upper limit of *x*

real number

Upper limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types: `double` | `single`

`ymin` — Lower limit of *y*

real number | function handle

Lower limit of *y*, specified as a real scalar value that is either finite or infinite. You also can specify `ymin` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Example:

Data Types: `double` | `function_handle` | `single`

`ymax` — Upper limit of *y*

real number | function handle

Upper limit of *y*, specified as a real scalar value that is either finite or infinite. You also can specify `ymax` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Example:

Data Types: `double` | `function_handle` | `single`

`zmin` — Lower limit of *z*

real number | function handle

Lower limit of *z*, specified as a real scalar value that is either finite or infinite. You also can specify `zmin` to be a function handle (a function of *x*,*y*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

`zmax` — Upper limit of *z*

real number | function handle

Upper limit of *z*, specified as a real scalar value that is either finite or infinite. You also can specify `zmax` to be a function handle (a function of *x*,*y*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

`'AbsTol'` — Absolute error tolerance

nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral3` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral3` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note: `AbsTol` and `RelTol` work together. `integral3` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-4010 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types: `double` | `single`

`'RelTol'` — Relative error tolerance

nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral3` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral3` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note: `RelTol` and `AbsTol` work together. `integral3` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-4010 section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types: `double` | `single`

`'Method'` — Integration method

`'auto'` (default) | `'tiled'` | `'iterated'`

Integration method, specified as the comma-separated pair consisting of `'Method'` and one of the methods described below.

Integration Method	Description
`'auto'`	For most cases, `integral3` uses the `'tiled'` method. It uses the `'iterated'` method when any of the integration limits are infinite. This is the default method.
`'tiled'`	`integral3` calls `integral` to integrate over `xmin` ≤ `x` ≤ `xmax`. It calls `integral2` with the `'tiled'` method to evaluate the double integral over `ymin(x)` ≤ `y` ≤ `ymax(x)` and `zmin(x,y)` ≤ `z` ≤ `zmax(x,y)`.
`'iterated'`	`integral3` calls `integral` to integrate over `xmin` ≤ `x` ≤ `xmax`. It calls `integral2` with the `'iterated'` method to evaluate the double integral over `ymin(x)` ≤ `y` ≤ `ymax(x)` and `zmin(x,y)` ≤ `z` ≤ `zmax(x,y)`. The integration limits can be infinite.

Example: `'Method','tiled'` specifies the tiled integration method.

Output Arguments

q — Computed integral

numeric value

Computed integral of `fun(x,y,z)` over the specified region, returned as a numeric value.

More About

Tips

- The `integral3` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.
- The `'iterated'` method can be more effective when your function has discontinuities within the integration region. However, the best performance and accuracy occurs when you split the integral at the points of discontinuity and sum the results of multiple integrations.
- When integrating over nonrectangular regions, the best performance and accuracy occurs when any or all of the limits: `ymin`, `ymax`, `zmin`, `zmax` are function handles. Avoid setting integrand function values to zero to integrate over a nonrectangular region. If you must do this, specify `'iterated'` method.
- Use the `'iterated'` method when any or all of the limits: `ymin(x)`, `ymax(x)`, `zmin(x,y)`, `zmax(x,y)` are unbounded functions.
- When paramaterizing anonymous functions, be aware that parameter values persist for the life of the function handle. For example, the function `fun = @(x,y,z) x + y + z + a` uses the value of `a` at the time `fun` was created. If you later decide to change the value of `a`, you must redefine the anonymous function with the new value.

- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you may need to specify larger absolute and relative error tolerances.

- "Parameterizing Functions"

See Also

`function_handle` | `integral` | `integral2` | `trapz`

interfaces

List custom interfaces exposed by COM server object

Syntax

```
customlist = interfaces(h)
```

Description

`customlist = interfaces(h)` returns cell array of strings `customlist` listing all custom interfaces implemented by the component in a specific COM server object. The server is designated by input argument `h`, the handle returned by the `actxcontrol` or `actxserver` function when creating that server.

The `interfaces` function only lists the custom interfaces exposed by the object; it does not return interfaces. Use the `invoke` function to return a handle to a specific custom interface.

COM functions are available on Microsoft Windows systems only.

More About

- "Custom Interfaces"

See Also

`actxcontrol` | `invoke` | `actxserver` | `get (COM)`

Introduced before R2006a

interp1

1-D data interpolation (table lookup)

Compatibility

If you pass nonuniformly spaced points and specify the `'v5cubic'` method, `interp1` issues a warning. In addition, the following syntaxes will be removed or changed in a future release:

- `interp1(...,'cubic')`
- `pp = interp1(...'pp')`
- `interp1(X,Y,Xq,[],...)`

For more information, and recommendations for updating your code, see "Functionality being removed or changed".

Syntax

```
vq = interp1(x,v,xq)
vq = interp1(x,v,xq,method)
vq = interp1(x,v,xq,method,extrapolation)

vq = interp1(v,xq)
vq = interp1(v,xq,method)
vq = interp1(v,xq,method,extrapolation)

pp = interp1(x,v,method,'pp')
```

Description

`vq = interp1(x,v,xq)` returns interpolated values of a 1-D function at specific query points using linear interpolation. Vector `x` contains the sample points, and `v` contains the corresponding values, $v(x)$. Vector `xq` contains the coordinates of the query points.

If you have multiple sets of data that are sampled at the same point coordinates, then you can pass `v` as an array. Each column of array `v` contains a different set of 1-D sample values.

`vq = interp1(x,v,xq,method)` specifies a string for choosing an alternative interpolation method: `'nearest'`, `'next'`, `'previous'`, `'linear'`,`'spline'`,`'pchip'`, or `'cubic'`. The default method is `'linear'`.

`vq = interp1(x,v,xq,method,extrapolation)` specifies a strategy for evaluating points that lie outside the domain of `x`. Set `extrapolation` to the string, `'extrap'`, when you want to use the method algorithm for extrapolation. Alternatively, you can specify a scalar value, in which case, `interp1` returns that value for all points outside the domain of `x`.

`vq = interp1(v,xq)` returns interpolated values and assumes a default set of sample point coordinates. The default points are the sequence of numbers from `1` to `n`, where `n` depends on the shape of `v`:

- When v is a vector, the default points are `1:length(v)`.
- When v is an array, the default points are `1:size(v,1)`.

Use this syntax when you are not concerned about the absolute distances between points.

`vq = interp1(v,xq,method)` specifies any of the alternative interpolation methods and uses the default sample points.

`vq = interp1(v,xq,method,extrapolation)` specifies an extrapolation strategy and uses the default sample points.

`pp = interp1(x,v,method,'pp')` returns the piece-wise polynomial form of $v(x)$ using the method algorithm.

Examples

Interpolation of Coarsely Sampled Sine Function

Define the sample points, `x`, and corresponding sample values, `v`.

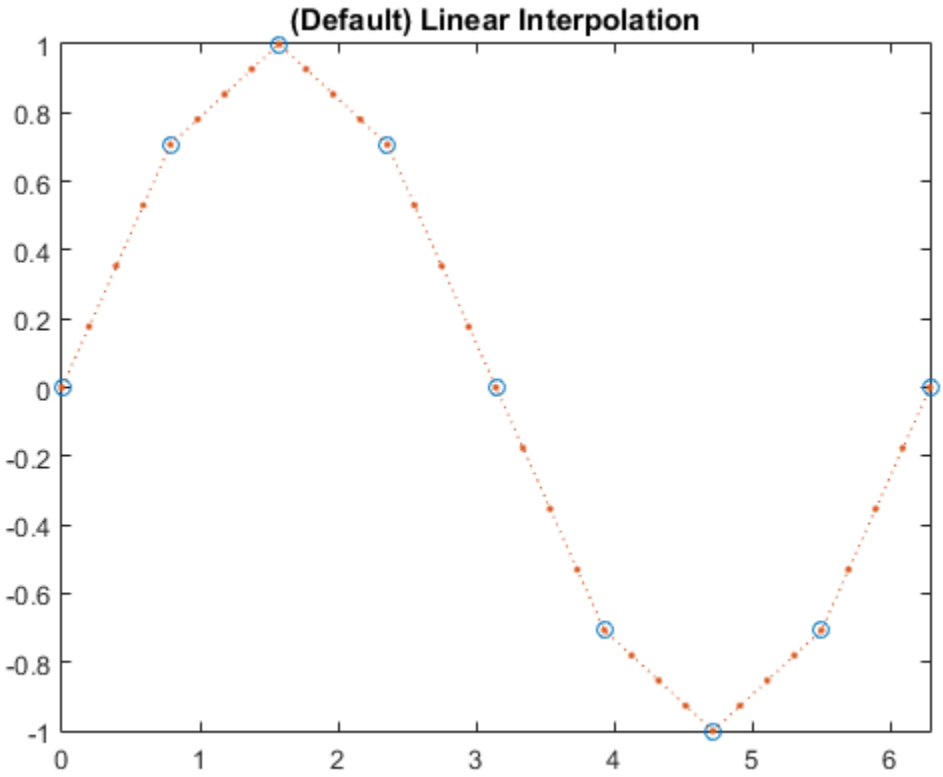
```
x = 0:pi/4:2*pi;
v = sin(x);
```

Define the query points to be a finer sampling over the range of `x`.

```
xq = 0:pi/16:2*pi;
```

Interpolate the function at the query points and plot the result.

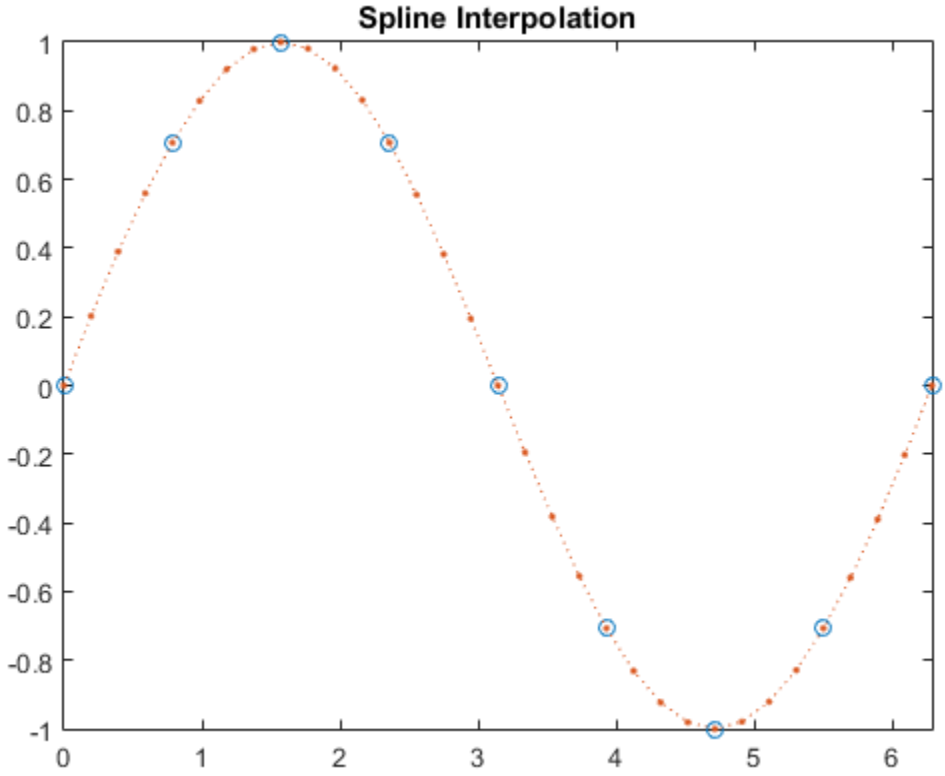
```
figure
vq1 = interp1(x,v,xq);
plot(x,v,'o',xq,vq1,':.');
xlim([0 2*pi]);
title('(Default) Linear Interpolation');
```



Now evaluate `v` at the same points using the `'spline'` method.

```
figure
```

```
vq2 = interp1(x,v,xq,'spline');
plot(x,v,'o',xq,vq2,':.');
xlim([0 2*pi]);
title('Spline Interpolation');
```



Interpolation Without Specifying Points

Define a set of function values.

```
v = [0  1.41  2  1.41  0  -1.41  -2  -1.41 0];
```

Define a set of query points that fall between the default points, `1:9`. In this case, the default points are `1:9` because `v` contains `9` values.

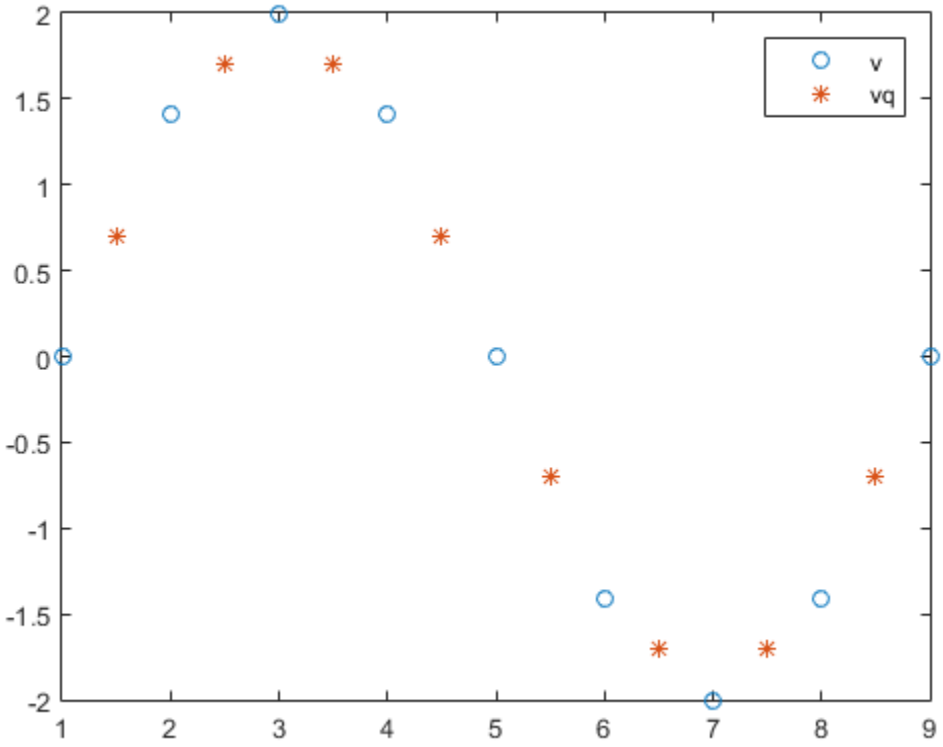
```
xq = 1.5:8.5;
```

Evaluate `v` at `xq`.

```
vq = interp1(v,xq);
```

Plot the result.

```
figure
plot((1:9),v,'o',xq,vq,'*');
legend('v','vq');
```



Interpolation of Complex Values

Define a set of sample points.

```
x = 1:10;
```

Define the values of the function, $v(x) = 5x + x^2 i$, at the sample points.

```
v = (5*x)+(x.^2*1i);
```

Define the query points to be a finer sampling over the range of `x`.

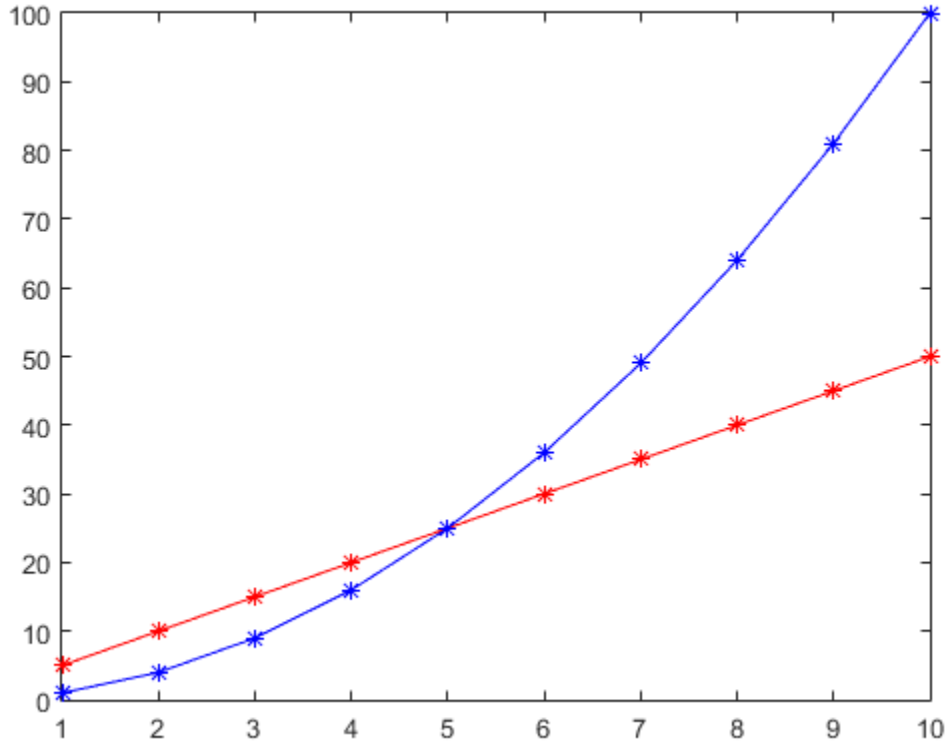
```
xq = 1:0.25:10;
```

Interpolate `v` at the query points.

```
vq = interp1(x,v,xq);
```

Plot the real part of the result in red and the imaginary part in blue.

```
figure
plot(x,real(v),'*r',xq,real(vq),'-r');
hold on
plot(x,imag(v),'*b',xq,imag(vq),'-b');
```



Extrapolation Using Two Different Methods

Define the sample points, `x`, and corresponding sample values, `v`.

```
x = [1 2 3 4 5];
v = [12 16 31 10 6];
```

Specify the query points, `xq`, that extend beyond the domain of `x`.

```
xq = [0 0.5 1.5 5.5 6];
```

Evaluate `v` at `xq` using the `'pchip'` method.

```
vq1 = interp1(x,v,xq,'pchip')
```

```
vq1 =

   19.3684   13.6316   13.2105    7.4800   12.5600
```

Next, evaluate `v` at `xq` using the `'linear'` method.

```
vq2 = interp1(x,v,xq,'linear')
```

```
vq2 =

   NaN   NaN    14   NaN   NaN
```

Now, use the `'linear'` method with the `'extrap'` option.

```
vq3 = interp1(x,v,xq,'linear','extrap')
```

```
vq3 =

     8    10    14     4     2
```

`'pchip'` extrapolates by default, but `'linear'` does not.

Designate Constant Value for All Queries Outside the Domain of x

Define the sample points, `x`, and corresponding sample values, `v`.

```
x = [-3 -2 -1 0 1 2 3];
v = 3*x.^2;
```

Specify the query points, `xq`, that extend beyond the domain of `x`.

```
xq = [-4 -2.5 -0.5 0.5 2.5 4];
```

Now evaluate `v` at `xq` using the `'pchip'` method and assign any values outside the domain of `x` to the value, `27`.

```
vq = interp1(x,v,xq,'pchip',27)
```

```
vq =

   27.0000   18.6563    0.9375    0.9375   18.6563   27.0000
```

Interpolate Multiple Sets of Data in One Pass

Define the sample points.

```
x = (-5:5)';
```

Sample three different parabolic functions at the points defined in `x`.

```
v1 = x.^2;
v2 = 2*x.^2 + 2;
v3 = 3*x.^2 + 4;
```

Create matrix `v`, whose columns are the vectors, `v1`, `v2`, and `v3`.

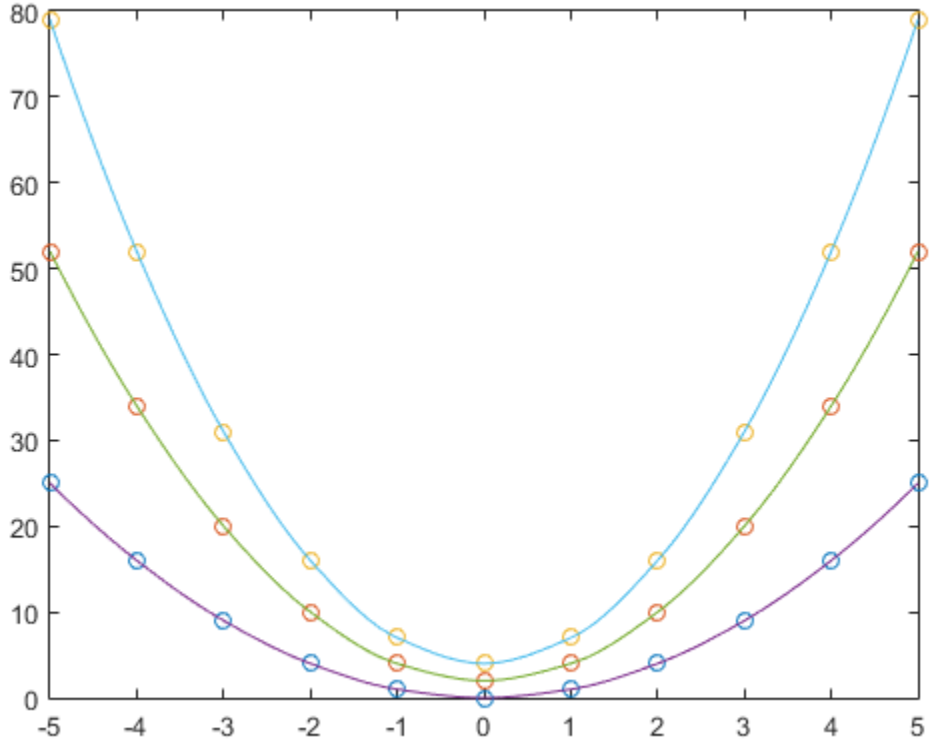
```
v = [v1 v2 v3];
```

Define a set of query points, `xq`, to be a finer sampling over the range of `x`.

```
xq = -5:0.1:5;
```

Evaluate all three functions at `xq` and plot the results.

```
vq = interp1(x,v,xq,'pchip');
figure
plot(x,v,'o',xq,vq);

h = gca;
h.XTick = -5:5;
```



The circles in the plot represent `v`, and the solid lines represent `vq`.

Input Arguments

x — Sample points

vector

Sample points, specified as a row or column vector of real numbers. The length of `x` must conform to one of the following requirements:

- If `v` is a vector, then `length(x)` must equal `length(v)`.
- If `v` is an array, then `length(x)` must equal `size(v,1)`.

Example: `[1 2 3 4 5 6 7 8 9 10]`

Example: `1:10`

Example: `[3 7 11 15 19 23 27 31]'`

Data Types: `single` | `double`

v — Sample values

vector | matrix | array

Sample values, specified as a vector, matrix, or array of real or complex numbers. If `v` is a matrix or an array, then each column contains a separate set of 1-D values.

Example: `rand(1,10)`

Example: `rand(10,1)`

Example: `rand(10,3)`

Data Types: `single` | `double`
Complex Number Support: Yes

xq — Query points

scalar | vector | matrix | array

Query points, specified as a scalar, vector, matrix, or array of real numbers.

Example: `5`

Example: `1:0.05:10`

Example: `(1:0.05:10)'`

Example: `[0 1 2 7.5 10]`

Data Types: `single` | `double`

method — Interpolation method

`'linear'` (default) | `'nearest'` | `'next'` | `'previous'` | `'spline'` | `'pchip'` | `'cubic'`

Interpolation method, specified as a string from the table below.

method string	Description	Continuity	Comments
`'linear'`	Linear interpolation. The interpolated value at a	C^0	• Requires at least 2 points.

method string	Description	Continuity	Comments
	query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.		• Requires more memory and computation time than nearest neighbor.
`'nearest'`	Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires at least 2 points. • Modest memory requirements • Fastest computation time
`'next'`	Next neighbor interpolation. The interpolated value at a query point is the value at the next sample grid point.	Discontinuous	• Requires at least 2 points. • Same memory requirements and computation time as `'nearest'`.
`'previous'`	Previous neighbor interpolation. The interpolated value at a query point is the value at the previous sample grid point.	Discontinuous	• Requires at least 2 points. • Same memory requirements and computation time as `'nearest'`.
`'pchip'`	Shape-preserving piecewise cubic interpolation. The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.	C^1	• Requires at least 4 points. • Requires more memory and computation time than linear.
`'cubic'`	Same as `'pchip'`.	C^1	This method currently returns the same result as `'pchip'`. In a future release, this method will perform cubic convolution.

method string	Description	Continuity	Comments
`'v5cubic'`	Cubic convolution used in MATLAB 5.	C^1	Points must be uniformly spaced. `'cubic'` will replace `'v5cubic'` in a future release.
`'spline'`	Spline interpolation using not-a-knot end conditions. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension.	C^2	• Requires at least 4 points. • Requires more memory and computation time than `'pchip'`.

`extrapolation` — Extrapolation strategy

`'extrap'` | scalar value

Extrapolation strategy, specified as the string, `'extrap'`, or a real scalar value.

- Specify `'extrap'` when you want `interp1` to evaluate points outside the domain using the same method it uses for interpolation.
- Specify a scalar value when you want `interp1` to return a specific constant value for points outside the domain.

The default behavior depends on the input arguments:

- If you specify the `'pchip'` or `'spline'` interpolation methods, then the default behavior is `'extrap'`.
- All other interpolation methods return `NaN` by default for query points outside the domain.

Example: `'extrap'`

Example: `5`

Data Types: `char` | `single` | `double`

Output Arguments

vq — Interpolated values

scalar | vector | matrix | array

Interpolated values, returned as a scalar, vector, matrix, or array. The size of `vq` depends on the shape of `v` and `xq`.

Shape of v	Shape of xq	Size of Vq	Example
Vector	Vector	`size(xq)`	If `size(v) = [1 100]` and `size(xq) = [1 500]`, then `size(vq) = [1 500]`.
Vector	Matrix or N-D Array	`size(xq)`	If `size(v) = [1 100]` and `size(xq) = [50 30]`, then `size(vq) = [50 30]`.
Matrix or N-D Array	Vector	`[length(xq) size(v,2),...,size(v,n`	If `size(v) = [100 3]` and `size(xq) = [1 500]`, then `size(vq) = [500 3]`.
Matrix or N-D Array	Matrix or N-D Array	`[size(xq,1),...,size(x` `size(v,2),...,size(v,m`	If `size(v) = [4 5 6]` and `size(xq) = [2 3 7]`, then `size(vq) = [2 3 7 5 6]`.

pp — Piecewise polynomial

structure

Piecewise polynomial, returned as a structure that you can pass to the `ppval` function for evaluation.

See Also

`griddedInterpolant` | `interp2` | `interp3` | `interpn`

Introduced before R2006a

interp1q

Quick 1-D linear interpolation

Note: `interp1q` is not recommended. Use `interp1` instead.

Syntax

```
yi = interp1q(x,Y,xi)
```

Description

`yi = interp1q(x,Y,xi)` returns the value of the 1-D function `Y` at the points of column vector `xi` using linear interpolation. The vector `x` specifies the coordinates of the underlying interval. The length of output `yi` is equal to the length of `xi`.

`interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking.

For `interp1q` to work properly,

- `x` must be a monotonically increasing column vector.
- `Y` must be a column vector or matrix with `length(x)` rows.
- `xi` must be a column vector

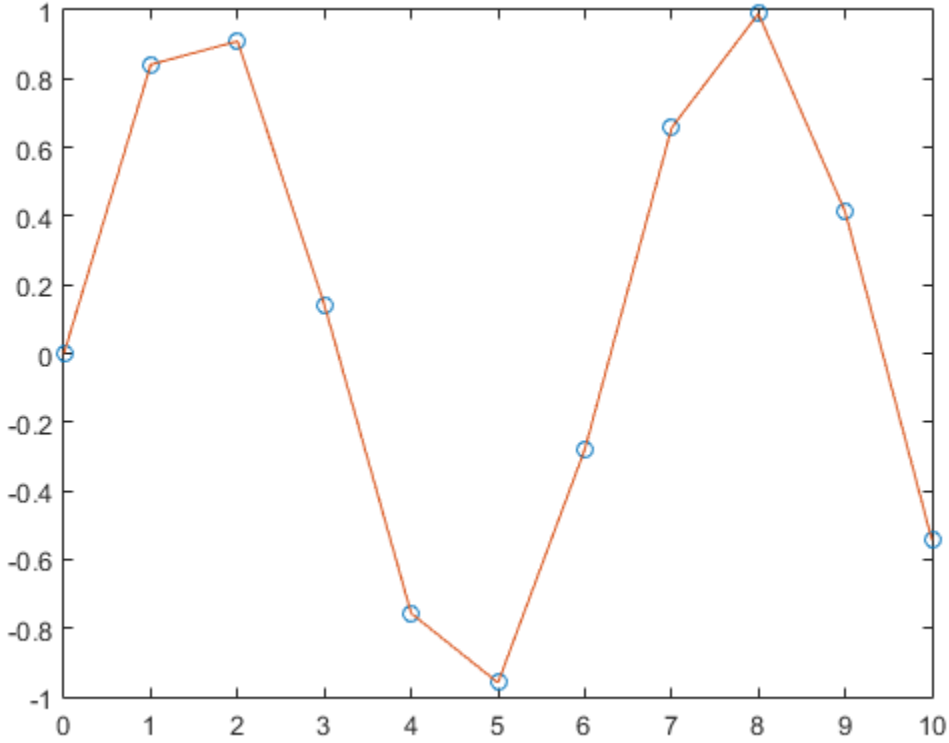
`interp1q` returns NaN for any values of `xi` that lie outside the coordinates in `x`. If `Y` is a matrix, then the interpolation is performed for each column of `Y`, in which case `yi` is `length(xi)-by-size(Y,2)`.

Examples

Linear Interpolation Using `interp1q`

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = (0:10)';
y = sin(x);
xi = (0:.25:10)';
yi = interp1q(x,y,xi);
plot(x,y,'o',xi,yi)
```



See Also

`interp1` | `interp2` | `interp3` | `interpn`

interp2

Interpolation for 2-D gridded data in meshgrid format

Compatibility

In a future release, `interp2` will not accept mixed combinations of row and column vectors for the sample and query grids. For more information, and recommendations for updating your code, see “Functionality being removed or changed”.

Syntax

```
Vq = interp2(X,Y,V,Xq,Yq)
Vq = interp2(V,Xq,Yq)
Vq = interp2(V)
Vq = interp2(V,k)

Vq = interp2( ___ ,method)
Vq = interp2( ___ ,method,extrapval)
```

Description

`Vq = interp2(X,Y,V,Xq,Yq)` returns interpolated values of a function of two variables at specific query points using linear interpolation. The results always pass through the original sampling of the function. `X` and `Y` contain the coordinates of the sample points. `V` contains the corresponding function values at each sample point. `Xq` and `Yq` contain the coordinates of the query points.

`Vq = interp2(V,Xq,Yq)` assumes a default grid of sample points. The default grid points cover the rectangular region, `X=1:n` and `Y=1:m`, where `[m,n] = size(V)`. Use this syntax to when you want to conserve memory and are not concerned about the absolute distances between points.

`Vq = interp2(V)` returns the interpolated values on a refined grid formed by dividing the interval between sample values once in each dimension.

`Vq = interp2(V,k)` returns the interpolated values on a refined grid formed by repeatedly dividing the intervals `k` times in each dimension.

`Vq = interp2(___,method)` specifies an optional, trailing input argument that you can pass with any of the previous syntaxes. The `method` argument can be any of the following strings that specify alternative interpolation methods: `'linear'`, `'nearest'`, `'cubic'`, or `'spline'`. The default method is `'linear'`.

`Vq = interp2(___,method,extrapval)` also specifies `extrapval`, a scalar value that is assigned to all queries that lie outside the domain of the sample points.

If you omit the `extrapval` argument for queries outside the domain of the sample points, then based on the `method` argument `interp2` returns one of the following:

- The extrapolated values for the `'spline'` method
- `NaN` values for interpolation methods other than `'spline'`

Examples

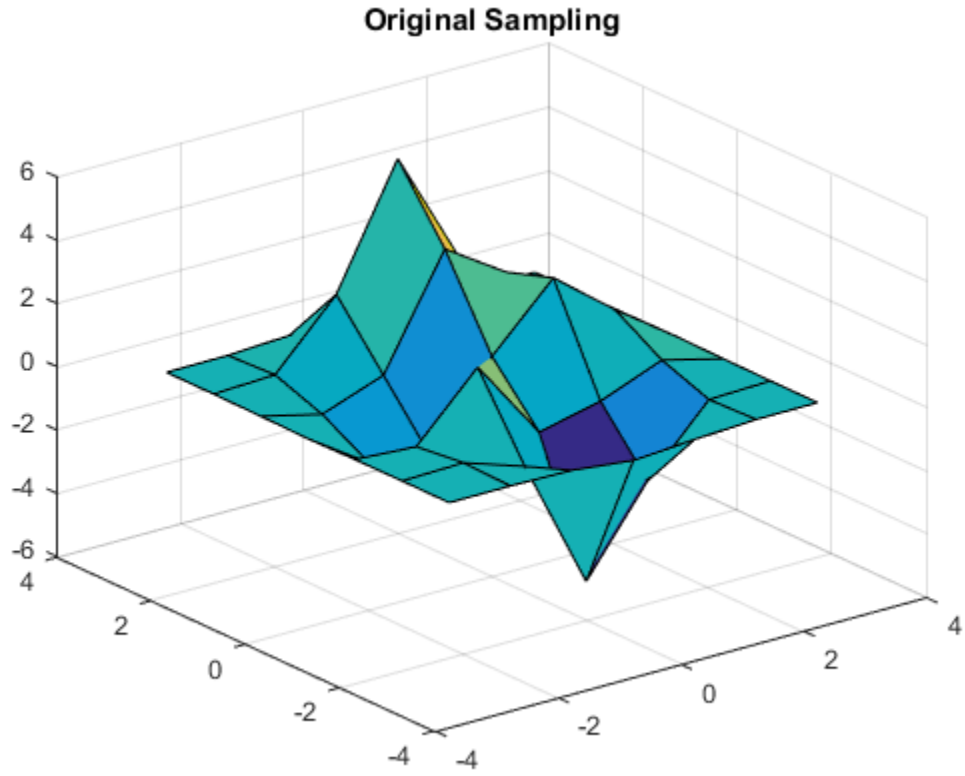
Interpolate Over a Grid Using Default Method

Coarsely sample the `peaks` function.

```
[X,Y] = meshgrid(-3:3);
V = peaks(X,Y);
```

Plot the coarse sampling.

```
figure
surf(X,Y,V)
title('Original Sampling');
```



Create the query grid with spacing of 0.25.

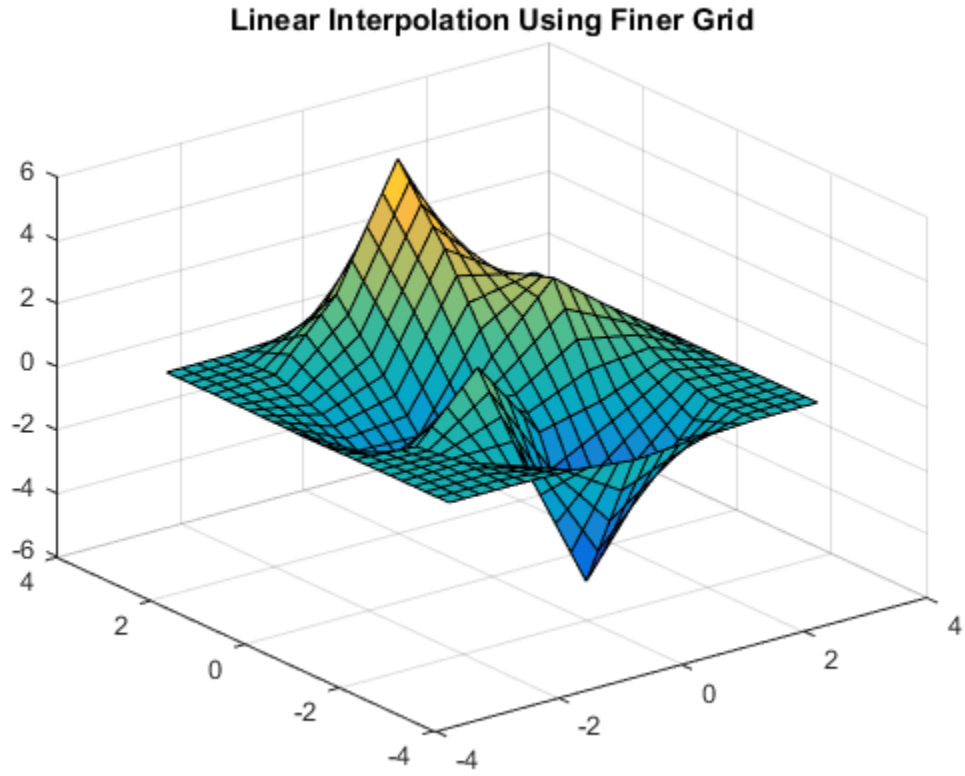
```
[Xq,Yq] = meshgrid(-3:0.25:3);
```

Interpolate at the query points.

```
Vq = interp2(X,Y,V,Xq,Yq);
```

Plot the result.

```
figure
surf(Xq,Yq,Vq);
title('Linear Interpolation Using Finer Grid');
```



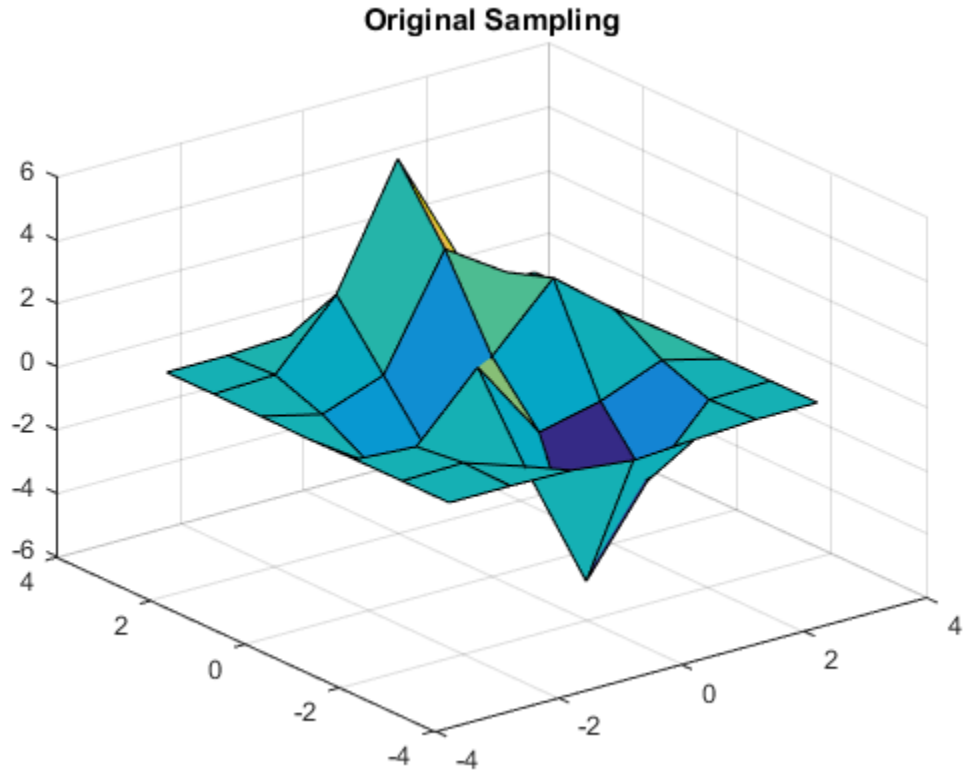
Interpolate Over a Grid Using Cubic Method

Coarsely sample the peaks function.

```
[X,Y] = meshgrid(-3:3);
V = peaks(7);
```

Plot the coarse sampling.

```
figure
surf(X,Y,V)
title('Original Sampling');
```



Create the query grid with spacing of 0.25.

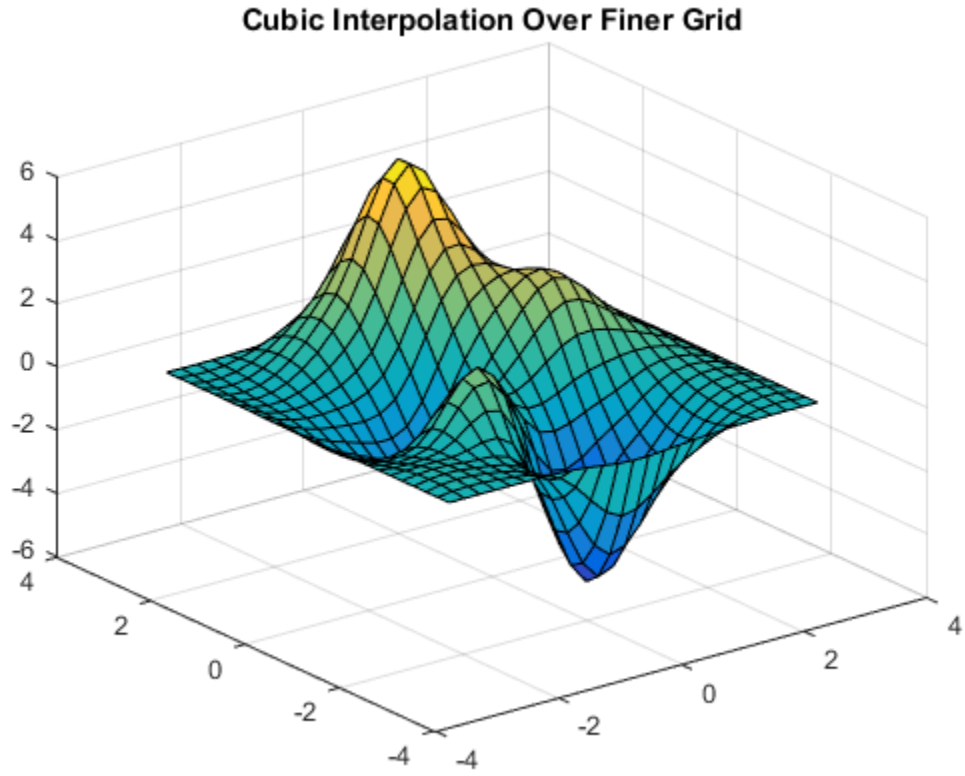
```
[Xq,Yq] = meshgrid(-3:0.25:3);
```

Interpolate at the query points, and specify cubic interpolation.

```
Vq = interp2(X,Y,V,Xq,Yq,'cubic');
```

Plot the result.

```
figure
surf(Xq,Yq,Vq);
title('Cubic Interpolation Over Finer Grid');
```



Refine Grayscale Image

Load some image data into the workspace.

```
load clown
```

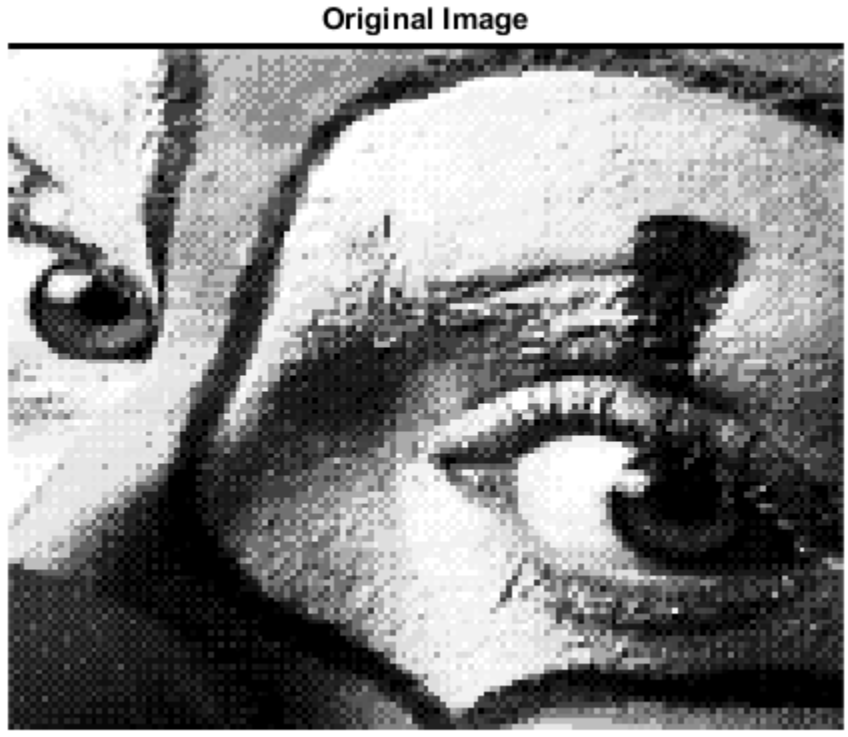
Isolate a small region of the image and cast it to single.

```
V = single(X(1:124,75:225));
```

Display the image.

```
figure
imagesc(V);
```

```
colormap gray
axis image
axis off
title('Original Image');
```



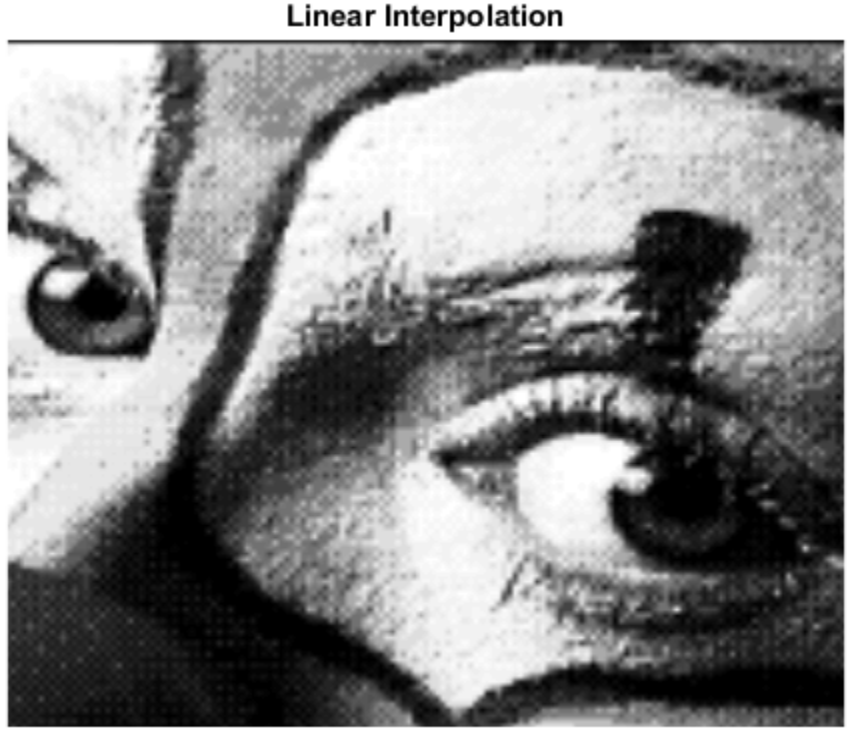
Insert interpolated values by repeatedly dividing the intervals between points of the refined grid five times in each dimension.

```
Vq = interp2(V,5);
```

Display the result.

```
figure
```

```
imagesc(Vq);
colormap gray
axis image
axis off
title('Linear Interpolation');
```



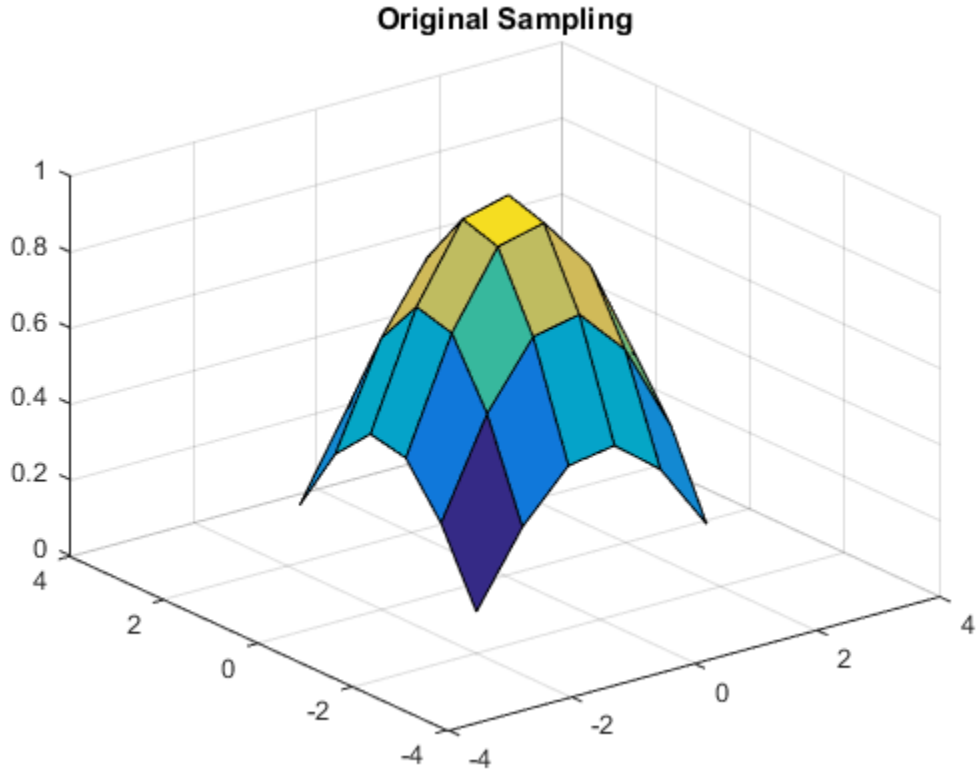
Evaluate Outside the Domain of X and Y

Coarsely sample a function over the range, `[-2, 2]` in both dimensions.

```
[X,Y] = meshgrid(-2:0.75:2);
R = sqrt(X.^2 + Y.^2)+ eps;
V = sin(R)./(R);
```

Plot the coarse sampling.

```
figure
surf(X,Y,V)
xlim([-4 4])
ylim([-4 4])
title('Original Sampling')
```



Create the query grid that extends beyond the domain of `X` and `Y`.

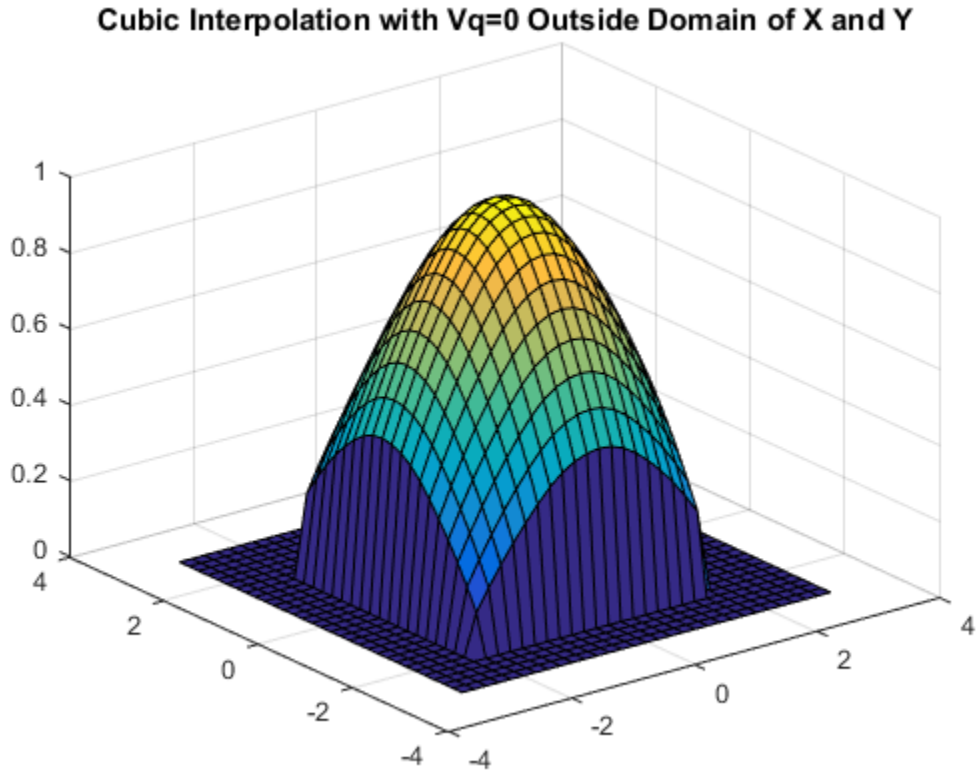
```
[Xq,Yq] = meshgrid(-3:0.2:3);
```

Perform cubic interpolation within the domain of `X` and `Y`, and assign all queries that fall outside to zero.

```
Vq = interp2(X,Y,V,Xq,Yq,'cubic',0);
```

Plot the result.

```
figure
surf(Xq,Yq,Vq)
title('Cubic Interpolation with Vq=0 Outside Domain of X and Y');
```



Input Arguments

X,Y — Sample grid points

matrices | vectors

Sample grid points, specified as real matrices or vectors.

- If `X` and `Y` are matrices, then they contain the coordinates of a full grid (in meshgrid format). Use the `meshgrid` function to create the `X` and `Y` matrices together. Both matrices must be the same size.
- If `X` and `Y` are vectors, then they are treated as a grid vectors. The values in both vectors must be strictly monotonic and increasing.

Example: `[X,Y] = meshgrid(1:30,-10:10)`

Data Types: `single` | `double`

`V` — Sample values

matrix

Sample values, specified as a real or complex matrix. The size requirements for `V` depend on the size of `X` and `Y`.

- If `X` and `Y` are matrices representing a full grid (in `meshgrid` format), then `V` must be the same size as `X` and `Y`.
- If `X` and `Y` are grid vectors, then `V` must be a matrix containing `length(Y)` rows and `length(X)` columns.

Example: `rand(10,10)`

Data Types: `single` | `double`
Complex Number Support: Yes

`Xq,Yq` — Query points

scalars | vectors | matrices | arrays

Query points, specified as a real scalars, vectors, matrices, or arrays.

- If `Xq` and `Yq` are scalars, then they are the coordinates of a single query point.
- If `Xq` and `Yq` are vectors of different orientations, then `Xq` and `Yq` are treated as grid vectors.
- If `Xq` and `Yq` are vectors of the same size and orientation, then `Xq` and `Yq` are treated as scattered points in 2-D space.
- If `Xq` and `Yq` are matrices, then they represent either a full grid of query points (in `meshgrid` format) or scattered points.

- If `Xq` and `Yq` are N-D arrays, then they represent scattered points in 2-D space.

Example: `[Xq,Yq] = meshgrid((1:0.1:10),(-5:0.1:0))`

Data Types: `single` | `double`

k — Refinement factor

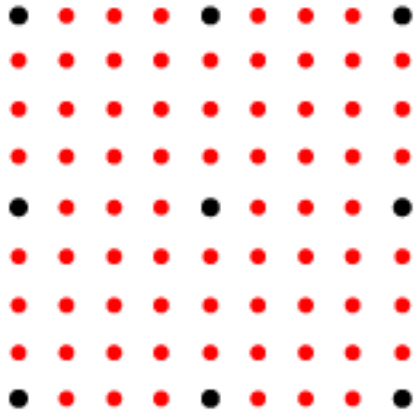
1 (default) | real, nonnegative, integer scalar

Refinement factor, specified as a real, nonnegative, integer scalar. This value specifies the number of times to repeatedly divide the intervals of the refined grid in each dimension. This results in `2^k-1` interpolated points between sample values.

If `k` is `0`, then `Vq` is the same as `V`.

`interp2(V,1)` is the same as `interp2(V)`.

The following illustration shows the placement of interpolated values (in red) among nine sample values (in black) for `k=2`.



Example: `interp2(V,2)`

Data Types: `single` | `double`

method — Interpolation method

`'linear'` (default) | `'nearest'` | `'cubic'` | `'spline'`

Interpolation method, specified as a string from this table.

Method	Description	Continuity	Comments
`'linear'`	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least two grid points in each dimension • Requires more memory than `'nearest'`
`'nearest'`	The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires two grid points in each dimension. • Fastest computation with modest memory requirements
`'cubic'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.	C^1	• Grid must have uniform spacing in each dimension, but the spacing does not have to be the same for all dimensions • Requires at least four points in each dimension • Requires more memory and computation time than `'linear'`
`'spline'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic spline using not-a-knot end conditions.	C^2	• Requires four points in each dimension • Requires more memory and computation time than `'cubic'`

`extrapval` — Function value outside domain of `X` and `Y`

scalar

Function value outside domain of `X` and `Y`, specified as a real or complex scalar. `interp2` returns this constant value for all points outside the domain of `X` and `Y`.

Example: `5`

Example: `5+1i`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`Vq` — Interpolated values

scalar | vector | matrix

Interpolated values, returned as a real or complex scalar, vector, or matrix. The size and shape of `Vq` depends on the syntax you use and, in some cases, the size and value of the input arguments.

Syntaxes	Special Conditions	Size of Vq	Example
`interp2(X,Y,V,Xq,Yq)` `interp2(V,Xq,Yq)` and variations of these syntaxes that include `method` or `extrapval`	`Xq`, `Yq` are scalars	Scalar	`size(Vq) = [1 1]` when you pass `Xq` and `Yq` as scalars.
Same as above	`Xq`, `Yq` are vectors of the same size and orientation	Vector of same size and orientation as `Xq` and `Yq`	If `size(Xq) = [100 1]` and `size(Yq) = [100 1]`, then `size(Vq) = [100 1]`.
Same as above	`Xq`, `Yq` are vectors of mixed orientation	Matrix in which the number of rows is `length(Yq)`, and the number of columns is `length(Xq)`	If `size(Xq) = [1 100]` and `size(Yq) = [50 1]`, then `size(Vq) = [50 100]`.
Same as above	`Xq`, `Yq` are matrices or arrays of the same size	Matrix or array of the same size as `Xq` and `Yq`	If `size(Xq) = [50 25]` and `size(Yq) = [50 25]`, then `size(Vq) = [50 25]`.

Syntaxes	Special Conditions	Size of Vq	Example
`interp2(V,k)` and variations of this syntax that include `method` or `extrapval`	None	Matrix in which the number of rows is: `2^k * (size(V,1)-1)+1`, and the number of columns is: `2^k * (size(V,2)-1)+1`	If `size(V) = [10 20]` and `k = 2`, then `size(Vq) = [37 77]`.

More About

Strictly Monotonic

A set of values that are always increasing or decreasing, without reversals. For example, the sequence, `a = [2 4 6 8]` is strictly monotonic and increasing. The sequence, `b = [2 4 4 6 8]` is not strictly monotonic because there is no change in value between `b(2)` and `b(3)`. The sequence, `c = [2 4 6 8 6]` contains a reversal between `c(4)` and `c(5)`, so it is not monotonic at all.

Full Grid (in meshgrid Format)

For `interp2`, the full grid is a pair of matrices whose elements represent a grid of points over a rectangular region. One matrix contains the x-coordinates, and the other matrix contains the y-coordinates. The values in the x-matrix are strictly monotonic and increasing along the rows. The values along its columns are constant. The values in the y-matrix are strictly monotonic and increasing along the columns. The values along its rows are constant. Use the `meshgrid` function to create a full grid that you can pass to `interp2`.

For example, the following code creates a full grid for the region, $-1 \le x \le 3$ and $1 \le y \le 4$:

```
[X,Y] = meshgrid(-1:3,(1:4))
```

```
X =

    -1     0     1     2     3
    -1     0     1     2     3
    -1     0     1     2     3
```

```
    -1     0     1     2     3

Y =

     1     1     1     1     1
     2     2     2     2     2
     3     3     3     3     3
     4     4     4     4     4
```

Grid Vectors

For `interp2`, grid vectors consist of a pair of mixed-orientation vectors that define the *x*- and *y*-coordinates in a grid. One vector is a row vector, and the other is a column vector.

For example, the following code creates the grid vectors that specify the region, $-1 \le x \le 3$ and $1 \le y \le 4$:

```
x = -1:3;
y = (1:4)';
```

Scattered Points

For `interp2`, scattered points consist of a pair of arrays that define a collection of points scattered in 2-D space. One array contains the *x*-coordinates, and the other contains the *y*-coordinates.

For example, the following code specifies the points, (2,7), (5,3), (4,1), and (10,9):

```
x = [2 5; 4 10];
y = [7 3; 1 9];
```

See Also

`griddata` | `griddedInterpolant` | `interp1` | `interp3` | `interpn` | `meshgrid` | `scatteredInterpolant`

Introduced before R2006a

interp3

Interpolation for 3-D gridded data in meshgrid format

Compatibility

In a future release, `interp3` will not accept mixed combinations of row and column vectors for the sample and query grids. For more information, and recommendations for updating your code, see “Functionality being removed or changed”.

Syntax

```
Vq = interp3(X,Y,Z,V,Xq,Yq,Zq)
Vq = interp3(V,Xq,Yq,Zq)
Vq = interp3(V)
Vq = interp3(V,k)

Vq = interp3(___,method)
Vq = interp3(___,method,extrapval)
```

Description

`Vq = interp3(X,Y,Z,V,Xq,Yq,Zq)` returns interpolated values of a function of three variables at specific query points using linear interpolation. The results always pass through the original sampling of the function. `X`, `Y`, and `Z` contain the coordinates of the sample points. `V` contains the corresponding function values at each sample point. `Xq`, `Yq`, and `Zq` contain the coordinates of the query points.

`Vq = interp3(V,Xq,Yq,Zq)` assumes a default grid of sample points. The default grid points cover the region, `X=1:n`, `Y=1:m`, `Z=1:p`, where `[m,n,p] = size(V)`. Use this syntax when you want to conserve memory and are not concerned about the absolute distances between points.

`Vq = interp3(V)` returns the interpolated values on a refined grid formed by dividing the interval between sample values once in each dimension.

`Vq = interp3(V,k)` returns the interpolated values on a refined grid formed by repeatedly dividing the intervals `k` times in each dimension.

`Vq = interp3(___,method)` specifies an optional, trailing input argument that you can pass with any of the previous syntaxes. The `method` argument can be any of the following strings that specify alternative interpolation methods: `'linear'`, `'nearest'`, `'cubic'`, or `'spline'`. The default method is `'linear'`.

`Vq = interp3(___,method,extrapval)` also specifies `extrapval`, a scalar value that is assigned to all queries that lie outside the domain of the sample points.

If you omit the `extrapval` argument for queries outside the domain of the sample points, then based on the `method` argument `interp3` returns one of the following:

- The extrapolated values for the `'spline'` method
- `NaN` values for interpolation methods other than `'spline'`

Examples

Interpolate Using Default Method

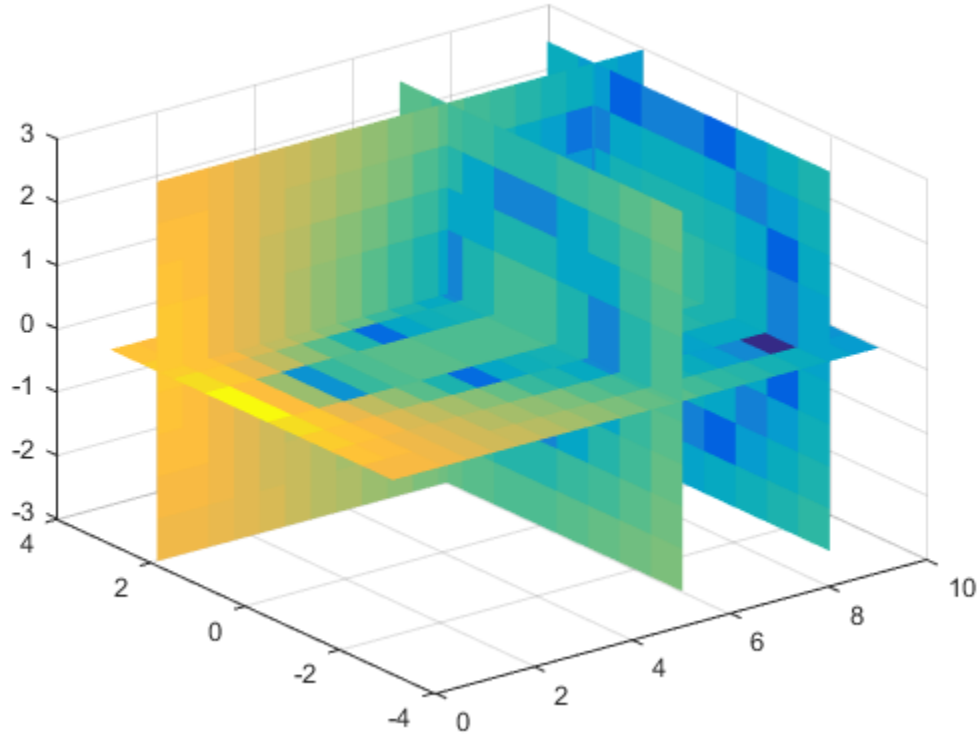
Load the points and values of the flow function, sampled at 10 points in each dimension.

```
[X,Y,Z,V] = flow(10);
```

The `flow` function returns the grid in the arrays, `X`, `Y`, `Z`. The grid covers the region, $0.1 \le X \le 10$, $-3 \le Y \le 3$, $-3 \le Z \le 3$, and the spacing is $\Delta X = 0.5$, $\Delta Y = 0.7$, and $\Delta Z = 0.7$.

Now, plot slices through the volume of the sample at: `X=6`, `X=9`, `Y=2`, and `Z=0`.

```
figure
slice(X,Y,Z,V,[6 9],2,0);
shading flat
```

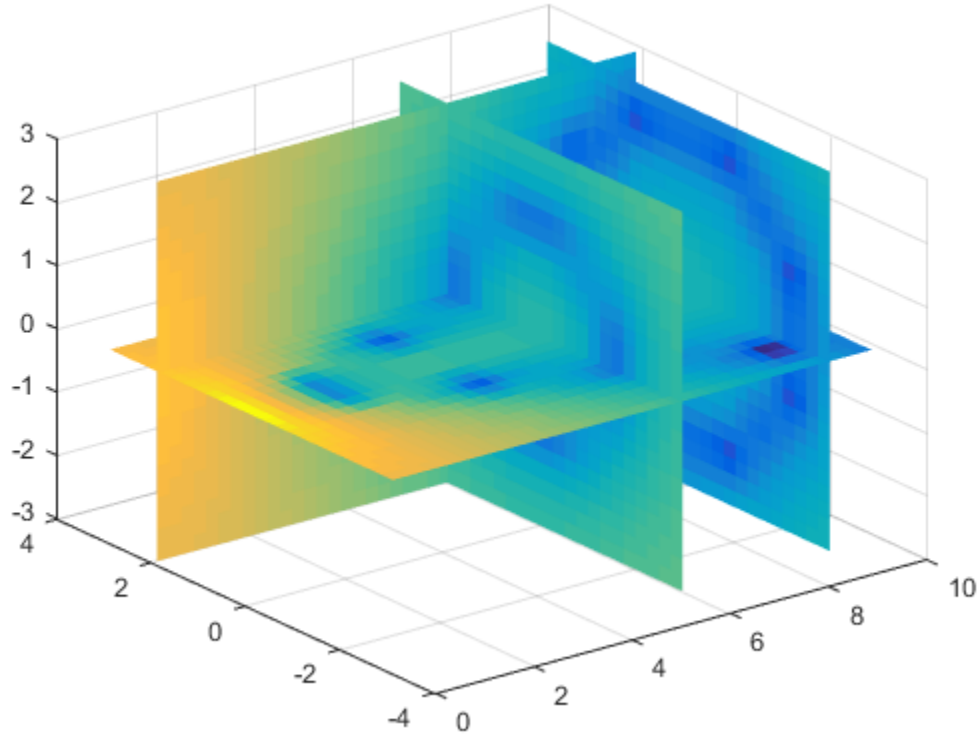


Create a query grid with spacing of 0.25.

```
[Xq,Yq,Zq] = meshgrid(.1:.25:10,-3:.25:3,-3:.25:3);
```

Interpolate at the points in the query grid and plot the results using the same slice planes.

```
Vq = interp3(X,Y,Z,V,Xq,Yq,Zq);
figure
slice(Xq,Yq,Zq,Vq,[6 9],2,0);
shading flat
```



Interpolate Using Cubic Method

Load the points and values of the flow function, sampled at 10 points in each dimension.

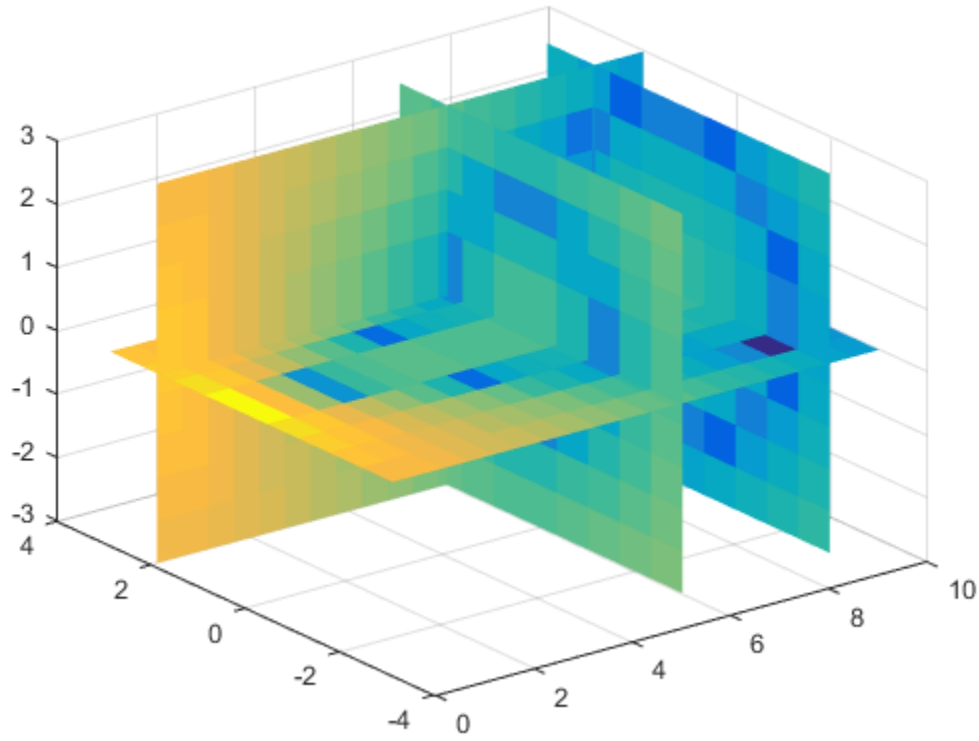
```
[X,Y,Z,V] = flow(10);
```

The `flow` function returns the grid in the arrays, `X`, `Y`, `Z`. The grid covers the region, $0.1 \le X \le 10$, $-3 \le Y \le 3$, $-3 \le Z \le 3$, and the spacing is $\Delta X = 0.5$, $\Delta Y = 0.7$, and $\Delta Z = 0.7$.

Plot slices through the volume of the sample at: `X=6`, `X=9`, `Y=2`, and `Z =0`.

```
figure
```

```
slice(X,Y,Z,V,[6 9],2,0);
shading flat
```



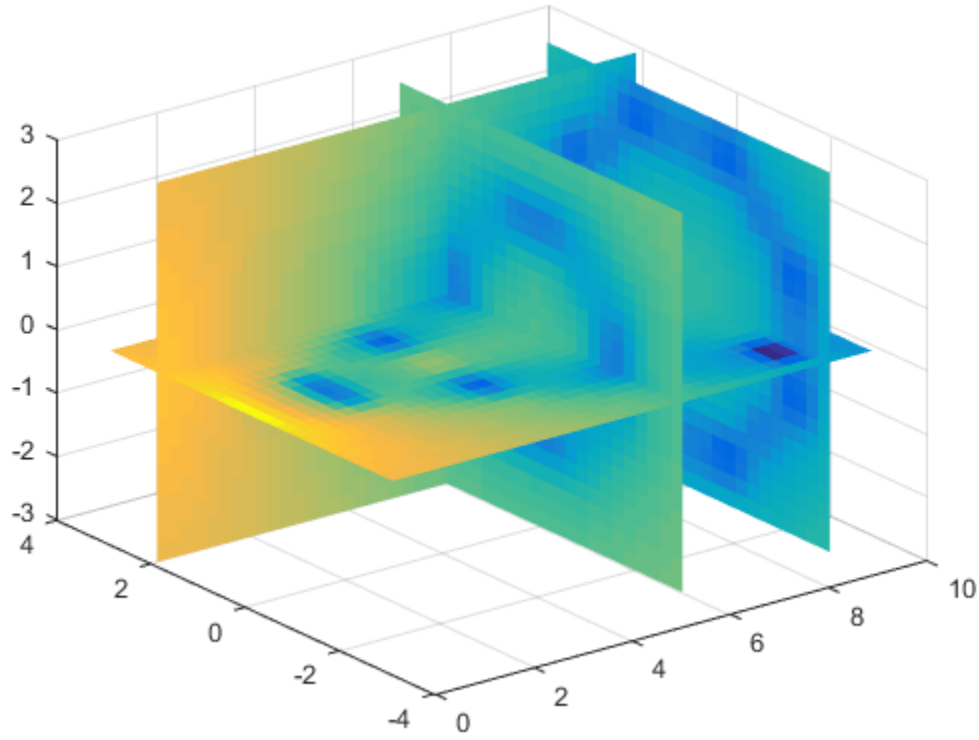
Create a query grid with spacing of 0.25.

```
[Xq,Yq,Zq] = meshgrid(.1:.25:10,-3:.25:3,-3:.25:3);
```

Interpolate at the points in the query grid using the `'cubic'` interpolation method. Then plot the results.

```
Vq = interp3(X,Y,Z,V,Xq,Yq,Zq,'cubic');
figure
slice(Xq,Yq,Zq,Vq,[6 9],2,0);
```

```
shading flat
```



Evaluate Outside the Domain of X, Y, and Z

Create the grid vectors, `x`, `y`, and `z`. These vectors define the points associated with values in `V`.

```
x = 1:100;
y = (1:50)';
z = 1:30;
```

Define the sample values to be a 50-by-100-by-30 random number array, `V`. Use the `gallery` function to create the array.

```
V = gallery('uniformdata',50,100,30,0);
```

Evaluate V at three points outside the domain of x, y, and z. Specify `extrapval = -1`.

```
xq = [0 0 0];
yq = [0 0 51];
zq = [0 101 102];
vq = interp3(x,y,z,V,xq,yq,zq,'linear',-1)
```

```
vq =

    -1    -1    -1
```

All three points evaluate to `-1` because they are outside the domain of x, y, and z.

Input Arguments

X,Y,Z — Sample grid points

arrays | vectors

Sample grid points, specified as real arrays or vectors.

- If X, Y, and Z are arrays, then they contain the coordinates of a full grid (in meshgrid format). Use the `meshgrid` function to create the X, Y, and Z arrays together. These arrays must be the same size.
- If X, Y, and Z are vectors, then they are treated as a grid vectors. The values in these vectors must be strictly monotonic and increasing.

Example: `[X,Y,Z] = meshgrid(1:30,-10:10,1:5)`

Data Types: `single | double`

V — Sample values

array

Sample values, specified as a real or complex array. The size requirements for V depend on the size of X, Y, and Z.

- If X, Y, and Z are arrays representing a full grid (in `meshgrid` format), then the size of V matches the size of X, Y, or Z .

- If `X`, `Y`, and `Z` are grid vectors, then `size(V) = [length(Y) length(X) length(Z)]`.

Example: `rand(10,10,10)`

Data Types: `single` | `double`
Complex Number Support: Yes

`Xq,Yq,Zq` — Query points

scalars | vectors | arrays

Query points, specified as a real scalars, vectors, or arrays.

- If `Xq`, `Yq`, and `Zq` are scalars, then they are the coordinates of a single query point in R^3.
- If `Xq`, `Yq`, and `Zq` are vectors of different orientations, then `Xq`, `Yq`, and `Zq` are treated as grid vectors in R^3.
- If `Xq`, `Yq`, and `Zq` are vectors of the same size and orientation, then `Xq`, `Yq`, and `Zq` are treated as scattered points in R^3.
- If `Xq`, `Yq`, and `Zq` are arrays of the same size, then they represent either a full grid of query points (in `meshgrid` format) or scattered points in R^3.

Example: `[Xq,Yq,Zq] = meshgrid((1:0.1:10),(-5:0.1:0),3:5)`

Data Types: `single` | `double`

`k` — Refinement factor

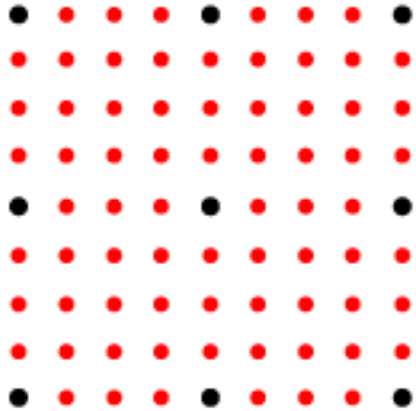
1 (default) | real, nonnegative, integer scalar

Refinement factor, specified as a real, nonnegative, integer scalar. This value specifies the number of times to repeatedly divide the intervals of the refined grid in each dimension. This results in `2^k-1` interpolated points between sample values.

If `k` is `0`, then `Vq` is the same as `V`.

`interp3(V,1)` is the same as `interp3(V)`.

The following illustration depicts `k=2` in one plane of R^3. There are 72 interpolated values in red and 9 sample values in black.



Example: `interp3(V,2)`

Data Types: `single` | `double`

`method` — Interpolation method

`'linear'` (default) | `'nearest'` | `'cubic'` | `'spline'`

Interpolation method, specified as a string from this table.

Method	Description	Continuity	Comments
`'linear'`	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least two grid points in each dimension • Requires more memory than `'nearest'`
`'nearest'`	The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires two grid points in each dimension • Fastest computation with modest memory requirements
`'cubic'`	The interpolated value at a query point is based on a cubic	C^1	• Grid must have uniform spacing in each dimension,

Method	Description	Continuity	Comments
	interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.		but the spacing does not have to be the same for all dimensions • Requires at least four points in each dimension • Requires more memory and computation time than `'linear'`
`'spline'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic spline using not-a-knot end conditions.	C^2	• Requires four points in each dimension • Requires more memory and computation time than `'cubic'`

`extrapval` — Function value outside domain of `X`, `Y`, and `Z`
scalar

Function value outside domain of `X`, `Y`, and `Z`, specified as a real or complex scalar. `interp3` returns this constant value for all points outside the domain of `X`, `Y`, and `Z`.

Example: `5`

Example: `5+1i`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`Vq` — Interpolated values
scalar | vector | array

Interpolated values, returned as a real or complex scalar, vector, or array. The size and shape of `Vq` depends on the syntax you use and, in some cases, the size and value of the input arguments.

Syntaxes	Special Conditions	Size of Vq	Example

`interp3(X,Y,Z,V,Xq,Yq)`
`interp3(V,Xq,Yq,Zq)`
and variations of these syntaxes that include `method` or `extrapval`
	`Xq`, `Yq`, and `Zq` are scalars.	Scalar	`size(Vq) = [1 1]` when you pass `Xq`, `Yq`, and `Zq` as scalars.

Same as above	`Xq`, `Yq`, and `Zq` are vectors of the same size and orientation.	Vector of same size and orientation as `Xq`, `Yq`, and `Zq`	If `size(Xq) = [100 1]`, and `size(Yq) = [100 1]`, and `size(Zq) = [100 1]`, then `size(Vq) = [100 1]`.
Same as above	`Xq`, `Yq`, and `Zq` are vectors of mixed orientation.	`size(Vq) = [length(Y) length(X) length(Z)]`	If `size(Xq) = [1 100]`, and `size(Yq) = [50 1]`, and `size(Zq) = [1 5]`, then `size(Vq) = [50 100 5]`.
Same as above	`Xq`, `Yq`, and `Zq` are arrays of the same size.	Array of the same size as `Xq`, `Yq`, and `Zq`	If `size(Xq) = [50 25]`, and `size(Yq) = [50 25]`, and `size(Zq) = [50 25]`, then `size(Vq) = [50 25]`.
`interp3(V,k)` and variations of this syntax that include `method` or `extrapval`	None	Array in which the length of the `i`th dimension is `2^k * (size(V,i)-1)+1`	If `size(V) = [10 12 5]`, and `k = 3`, then `size(Vq) = [73 89 33]`.

More About

Strictly Monotonic

A set of values that are always increasing or decreasing, without reversals. For example, the sequence, `a = [2 4 6 8]` is strictly monotonic and increasing. The sequence, `b`

= [2 4 4 6 8] is not strictly monotonic because there is no change in value between b(2) and b(3). The sequence, c = [2 4 6 8 6] contains a reversal between c(4) and c(5), so it is not monotonic at all.

Full Grid (in meshgrid Format)

For interp3, a full grid consists of three arrays whose elements represent a grid of points that define a region in R^3. The first array contains the x-coordinates, the second array contains the y-coordinates, and the third array contains the z-coordinates. The values in each array vary along a single dimension and are constant along the other dimensions.

The values in the x-array are strictly monotonic, increasing, and vary along the second dimension. The values in the y-array are strictly monotonic, increasing, and vary along the first dimension. The values in the z-array are strictly monotonic, increasing, and vary along the third dimension. Use the meshgrid function to create a full grid that you can pass to interp3.

Grid Vectors

For interp3, grid vectors consist of three vectors of mixed-orientation that define the points on a grid in R^3.

For example, the following code creates the grid vectors for the region, $1 \le x \le 3$, $4 \le y \le 5$, and $6 \le z \le 8$:

```
x = 1:3;
y = (4:5)';
z = 6:8;
```

Scattered Points

For interp3, scattered points consist of three arrays or vectors, Xq, Yq, and Zq, that define a collection of points scattered in R^3. The ith array contains the coordinates in the ith dimension.

For example, the following code specifies the points, (1, 19, 10), (6, 40, 1), (15, 33, 22), and (0, 61, 13).

```
Xq = [1 6; 15 0];
Yq = [19 40; 33 61];
Zq = [10 1; 22 13];
```

See Also

`interp1` | `interp2` | `interpn` | `meshgrid`

Introduced before R2006a

interpft

1-D interpolation using FFT method

Syntax

```
y = interpft(x,n)
y = interpft(x,n,dim)
```

Description

`y = interpft(x,n)` returns the vector `y` that contains the value of the periodic function `x` resampled to `n` equally spaced points.

If `length(x) = m`, and `x` has sample interval `dx`, then the new sample interval for `y` is `dy = dx*m/n`. Note that `n` cannot be smaller than `m`.

If `X` is a matrix, `interpft` operates on the columns of `X`, returning a matrix `Y` with the same number of columns as `X`, but with `n` rows.

`y = interpft(x,n,dim)` operates along the specified dimension.

Examples

Interpolate a triangle-like signal using an interpolation factor of 5. First, set up signal to be interpolated:

```
y = [0 .5 1 1.5 2 1.5 1 .5 0 -.5 -1 -1.5 -2 -1.5 -1 -.5 0];
N = length(y);
```

Perform the interpolation:

```
L = 5;
M = N*L;
x = 0:L:L*N-1;
xi = 0:M-1;
yi = interpft(y,M);
```

```
plot(x,y,'o',xi,yi,'*')
legend('Original data','Interpolated data')
```

More About

Algorithms

The `interpft` command uses the FFT method. The original vector `x` is transformed to the Fourier domain using `fft` and then transformed back with more points.

See Also

`interp1`

Introduced before R2006a

interpn

Interpolation for 1-D, 2-D, 3-D, and N-D gridded data in ndgrid format

Compatibility

In a future release, `interpn` will not accept mixed combinations of row and column vectors for the sample and query grids. For more information, and recommendations for updating your code, see "Functionality being removed or changed".

Syntax

```
Vq = interpn(X1,X2,...,Xn,V,Xq1,Xq2,...,Xqn)
Vq = interpn(V,Xq1,Xq2,...,Xqn)
Vq = interpn(V)
Vq = interpn(V,k)

Vq = interpn( ___ ,method)
Vq = interpn( ___ ,method,extrapval)
```

Description

`Vq = interpn(X1,X2,...,Xn,V,Xq1,Xq2,...,Xqn)` returns interpolated values of a function of n variables at specific query points using linear interpolation. The results always pass through the original sampling of the function. `X1,X2,...,Xn` contain the coordinates of the sample points. `V` contains the corresponding function values at each sample point. `Xq1,Xq2,...,Xqn` contain the coordinates of the query points.

`Vq = interpn(V,Xq1,Xq2,...,Xqn)` assumes a default grid of sample points. The default grid consists of the points, 1,2,3,...n_i in each dimension. The value of n_i is the length of the ith dimension in `V`. Use this syntax to when you want to conserve memory and are not concerned about the absolute distances between points.

`Vq = interpn(V)` returns the interpolated values on a refined grid formed by dividing the interval between sample values once in each dimension.

`Vq = interpn(V,k)` returns the interpolated values on a refined grid formed by repeatedly dividing the intervals `k` times in each dimension.

`Vq = interpn(___,method)` specifies an optional, trailing input argument that you can pass with any of the previous syntaxes. The `method` argument can be any of the following strings that specify alternative interpolation methods: `'linear'`, `'nearest'`, `'pchip'`,`'cubic'`, or `'spline'`. The default method is `'linear'`.

`Vq = interpn(___,method,extrapval)` also specifies `extrapval`, a scalar value that is assigned to all queries that lie outside the domain of the sample points.

If you omit the `extrapval` argument for queries outside the domain of the sample points, then based on the `method` argument `interpn` returns one of the following:

- The extrapolated values for the `'spline'` method
- `NaN` values for interpolation methods other than `'spline'`

Examples

1-D Interpolation

Define the sample points and values.

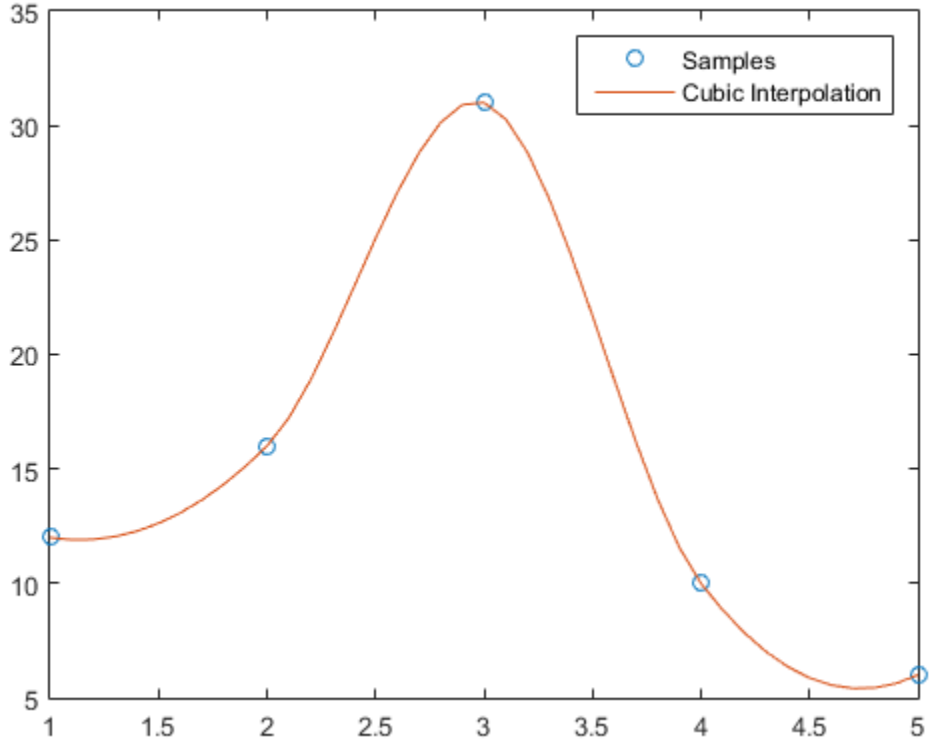
```
x = [1 2 3 4 5];
v = [12 16 31 10 6];
```

Define the query points, `xq`, and interpolate.

```
xq = (1:0.1:5);
vq = interpn(x,v,xq,'cubic');
```

Plot the result.

```
figure
plot(x,v,'o',xq,vq,'-');
legend('Samples','Cubic Interpolation');
```



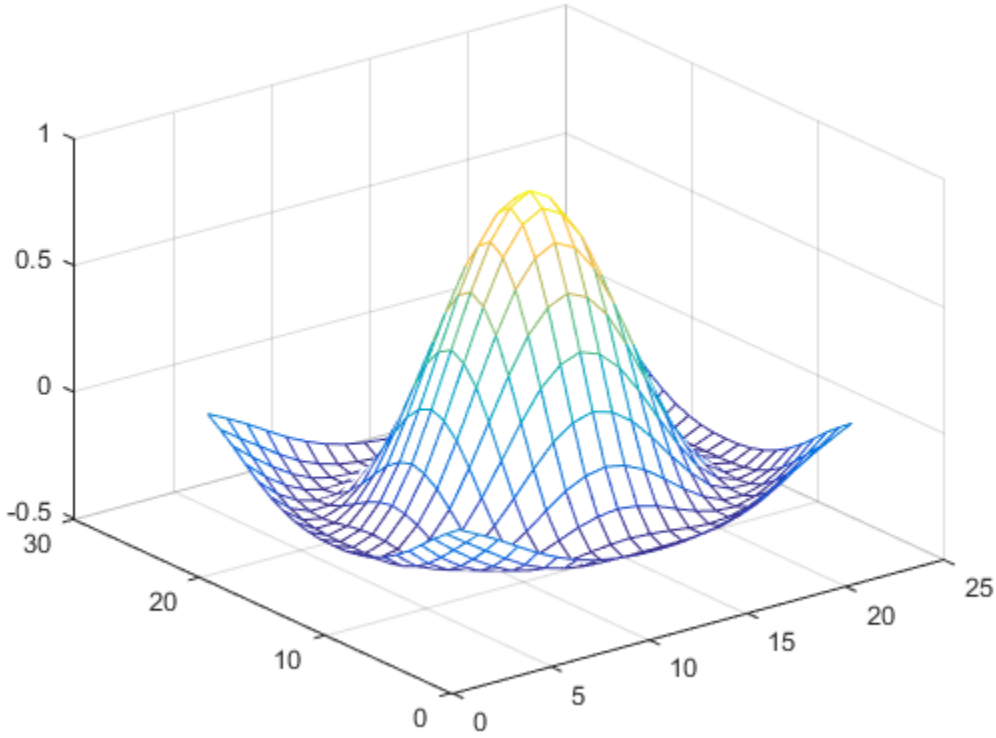
2-D Interpolation

Create a set of grid points and corresponding sample values.

```
[X1,X2] = ndgrid((-5:1:5));
R = sqrt(X1.^2 + X2.^2)+ eps;
V = sin(R)./(R);
```

Interpolate over a finer grid using `ntimes=1`.

```
Vq = interpn(V,'cubic');
mesh(Vq);
```



Evaluate Outside Domain of 3-D Function

Create the grid vectors, `x1`, `x2`, and `x3`. These vectors define the points associated with the values in `V`.

```
x1 = 1:100;
x2 = (1:50)';
x3 = 1:30;
```

Define the sample values to be a 100-by-50-by-30 random number array, `V`. Use the `gallery` function to create the array.

```
V = gallery('uniformdata',100,50,30,0);
```

Evaluate V at three points outside the domain of x1, x2, and x3. Specify extrapval = -1.

```
xq1 = [0 0 0];
xq2 = [0 0 51];
xq3 = [0 101 102];
vq = interpn(x1,x2,x3,V,xq1,xq2,xq3,'linear',-1)
```

```
vq =

    -1    -1    -1
```

All three points evaluate to -1 because they are outside the domain of x1, x2, and x3.

4-D Interpolation

Define an anonymous function that represents $f = te^{-x^2-y^2-z^2}$.

```
f = @(x,y,z,t) t.*exp(-x.^2 - y.^2 - z.^2);
```

Create a grid of points in R^4. Then, pass the points through the function to create the sample values, V.

```
[x,y,z,t] = ndgrid(-1:0.2:1,-1:0.2:1,-1:0.2:1,0:2:10);
V = f(x,y,z,t);
```

Now, create the query grid.

```
[xq,yq,zq,tq] = ...
ndgrid(-1:0.05:1,-1:0.08:1,-1:0.05:1,0:0.5:10);
```

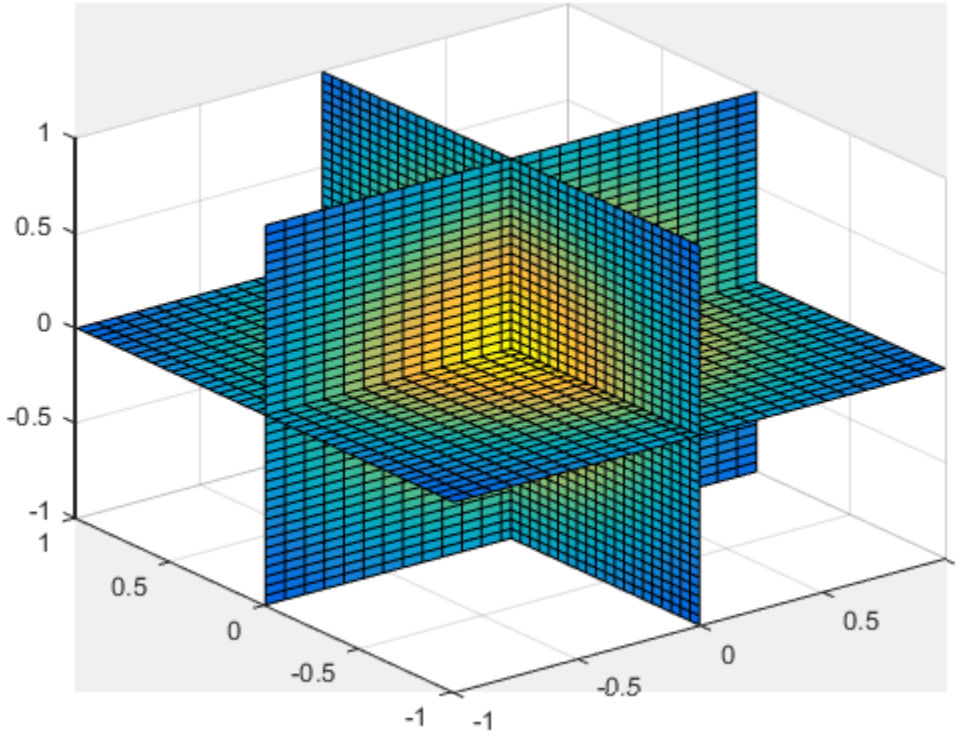
Interpolate V at the query points.

```
Vq = interpn(x,y,z,t,V,xq,yq,zq,tq);
```

Create a movie to show the results.

```
figure('renderer','zbuffer');
nframes = size(tq, 4);
for j = 1:nframes
   slice(yq(:,:,:,j),xq(:,:,:,j),zq(:,:,:,j),...
```

```
        Vq(:,:,:,j),0,0,0);
   caxis([0 10]);
   M(j) = getframe;
end
movie(M);
```



Input Arguments

X1,X2,...,Xn — Sample grid points

arrays | vectors

Sample grid points, specified as real arrays or vectors.

- If `X1,X2,...,Xn` are arrays, then they contain the coordinates of a full grid (in ndgrid format). Use the `ndgrid` function to create the `X1,X2,...,Xn` arrays together. These arrays must be the same size.
- If `X1,X2,...,Xn` are vectors, then they are treated as grid vectors. The values in these vectors must be strictly monotonic and increasing.

Example: `[X1,X2,X3] = ndgrid(1:30,-10:10,1:5)`

Data Types: `single | double`

`V` — Sample values

array

Sample values, specified as a real or complex array. The size requirements for `V` depend on the size of `X1,X2,...,Xn`.

- If `X1,X2,...,Xn` are arrays representing a full grid (in `ndgrid` format), then the size of `V` matches the size of any array, `X1,X2,...,Xn`.
- If `X1,X2,...,Xn` are grid vectors, then `V` is an array whose `i`th dimension is the same length as grid vector `Xi`, where `i= 1,2,...n`.

Example: `rand(10,5,3,2)`

Data Types: `single | double`
Complex Number Support: Yes

`Xq1,Xq2,...,Xqn` — Query points

scalars | vectors | arrays

Query points, specified as a real scalars, vectors, or arrays.

- If `Xq1,Xq2,...,Xqn` are scalars, then they are the coordinates of a single query point in R^n.
- If `Xq1,Xq2,...,Xqn` are vectors of different orientations, then `Xq1,Xq2,...,Xqn` are treated as grid vectors in R^n.
- If `Xq1,Xq2,...,Xqn` are vectors of the same size and orientation, then `Xq1,Xq2,...,Xqn` are treated as scattered points in R^n.
- If `Xq1,Xq2,...,Xqn` are arrays of the same size, then they represent either a full grid of query points (in `ndgrid` format) or scattered points in R^n.

Example: `[X1,X2,X3,X4] = ndgrid(1:10,1:5,7:9,10:11)`

Data Types: `single` | `double`

`k` — Refinement factor

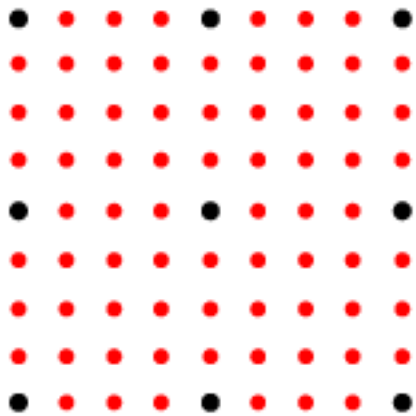
1 (default) | real, nonnegative, integer scalar

Refinement factor, specified as a real, nonnegative, integer scalar. This value specifies the number of times to repeatedly divide the intervals of the refined grid in each dimension. This results in `2^k-1` interpolated points between sample values.

If `k` is `0`, then `Vq` is the same as `V`.

`interpn(V,1)` is the same as `interpn(V)`.

The following illustration depicts `k=2` in R^2. There are 72 interpolated values in red and 9 sample values in black.



Example: `interpn(V,2)`

Data Types: `single` | `double`

`method` — Interpolation method

`'linear'` (default) | `'nearest'` | `'pchip'` | `'cubic'` | `'spline'`

Interpolation method, specified as a string from this table.

Method	Description	Continuity	Comments
`'linear'`	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least two grid points in each dimension • Requires more memory than `'nearest'`
`'nearest'`	The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires two grid points in each dimension. • Fastest computation with modest memory requirements
`'pchip'`	Shape-preserving piecewise cubic interpolation (for 1-D only). The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.	C^1	• Requires at least four points • Requires more memory and computation time than `'linear'`
`'cubic'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.	C^1	• Grid must have uniform spacing in each dimension, but the spacing does not have to be the same for all dimensions • Requires at least four points in each dimension • Requires more memory and computation time than `'linear'`
`'spline'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic	C^2	• Requires four points in each dimension • Requires more memory and computation time than `'cubic'`

Method	Description	Continuity	Comments
	spline using not-a-knot end conditions.		

`extrapval` — Function value outside domain of `X1,X2,...,Xn`

scalar

Function value outside domain of `X1,X2,...,Xn`, specified as a real or complex scalar. `interpn` returns this constant value for all points outside the domain of `X1,X2,...,Xn`.

Example: `5`

Example: `5+1i`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`Vq` — Interpolated values

scalar | vector | array

Interpolated values, returned as a real or complex scalar, vector, or array. The size and shape of `Vq` depends on the syntax you use and, in some cases, the size and value of the input arguments.

Syntaxes	Special Conditions	Size of Vq	Example
`interpn(X1,...,Xn,V,X` `interpn(V,Xq1,...,Xqn` and variations of these syntaxes that include `method` or `extrapval`	`Xq1,...,Xqn` are scalars	Scalar	`size(Vq) = [1 1]` when you pass `Xq1,...,Xqn` as scalars.
Same as above	`Xq1,...,Xqn` are vectors of the same size and orientation	Vector of same size and orientation as `Xq1,...,Xqn`	In 3-D, if `size(Xq1) = [100 1]`, and `size(Xq2) = [100 1]`, and `size(Xq3) = [100 1]`,

Syntaxes	Special Conditions	Size of Vq	Example
			then `size(Vq) = [100 1]`.
Same as above	`Xq1,...,Xqn` are vectors of mixed orientation	`size(Vq) = [length(Xq1),...,leng`	In 3-D, if `size(Xq1) = [1 100]`, and `size(Xq2) = [50 1]`, and `size(Xq3) = [1 5]`, then `size(Vq) = [100 50 5]`.
Same as above	`Xq1,...,Xqn` are arrays of the same size	Array of the same size as `Xq1,...,Xqn`	In 3-D, if `size(Xq1) = [50 25]`, and `size(Xq2) = [50 25]`, and `size(Xq3) = [50 25]`, then `size(Vq) = [50 25]`.
`interpn(V,k)` and variations of this syntax that include `method` or `extrapval`	None	Array in which the length of the `i`th dimension is `2^k * (size(V,i)-1)+1`,	In 3-D, if `size(V) = [10 12 5]`, and `k = 3`, then `size(Vq) = [73 89 33]`.

More About

Strictly Monotonic

A set of values that are always increasing or decreasing, without reversals. For example, the sequence, `a = [2 4 6 8]` is strictly monotonic and increasing. The sequence, `b = [2 4 4 6 8]` is not strictly monotonic because there is no change in value between `b(2)` and `b(3)`. The sequence, `c = [2 4 6 8 6]` contains a reversal between `c(4)` and `c(5)`, so it is not monotonic at all.

Full Grid (in ndgrid Format)

For `interpn`, the full grid consists of n arrays, `X1,X2,...,Xn`, whose elements represent a grid of points in R^n. The ith array, X_i, contains strictly monotonic, increasing values that vary most rapidly along the ith dimension.

Use the `ndgrid` function to create a full grid that you can pass to `interpn`. For example, the following code creates a full grid in R^2 for the region, $1 \le X1 \le 3$, $1 \le X2 \le 4$.

```
[X1,X2] = ndgrid(-1:3,(1:4))
```

```
X1 =

    -1    -1    -1    -1
     0     0     0     0
     1     1     1     1
     2     2     2     2
     3     3     3     3


X2 =

     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
```

Grid Vectors

For `interpn`, grid vectors consist of n vectors of mixed-orientation that define the points of a grid in R^n.

For example, the following code creates the grid vectors in R^3 for the region, $1 \le x1 \le 3$, $4 \le x2 \le 5$, and $6 \le x3 \le 8$:

```
x1 = 1:3;
x2 = (4:5)';
x3 = 6:8;
```

Scattered Points

For `interpn`, scattered points consist of n arrays or vectors, `Xq1,Xq2,...,Xqn`, that define a collection of points scattered in R^n. The `i`th array, `Xi`, contains the coordinates in the `i`th dimension.

For example, the following code specifies the points, (1, 19, 10), (6, 40, 1), (15, 33, 22), and (0, 61, 13) in R^3.

```
Xq1 = [1 6; 15 0];
Xq2 = [19 40; 33 61];
Xq3 = [10 1; 22 13];
```

See Also

`interp1` | `interp2` | `interp3` | `ndgrid`

Introduced before R2006a

interpstreamspeed

Interpolate stream-line vertices from flow speed

Syntax

```
interpstreamspeed(X,Y,Z,U,V,W,vertices)
interpstreamspeed(U,V,W,vertices)
interpstreamspeed(X,Y,Z,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(X,Y,U,V,vertices)
interpstreamspeed(U,V,vertices)
interpstreamspeed(X,Y,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(...,sf)
vertsout = interpstreamspeed(...)
```

Description

`interpstreamspeed(X,Y,Z,U,V,W,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`interpstreamspeed(U,V,W,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(U)`.

`interpstreamspeed(X,Y,Z,speed,vertices)` uses the 3-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p]=size(speed)`.

`interpstreamspeed(X,Y,U,V,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`interpstreamspeed(U,V,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M N]=size(U)`.

`interpstreamspeed(X,Y,speed,vertices)` uses the 2-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M,N]= size(speed)`.

`interpstreamspeed(...,sf)` uses `sf` to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if `sf` is 3, then `interpstreamspeed` creates only one-third of the vertices.

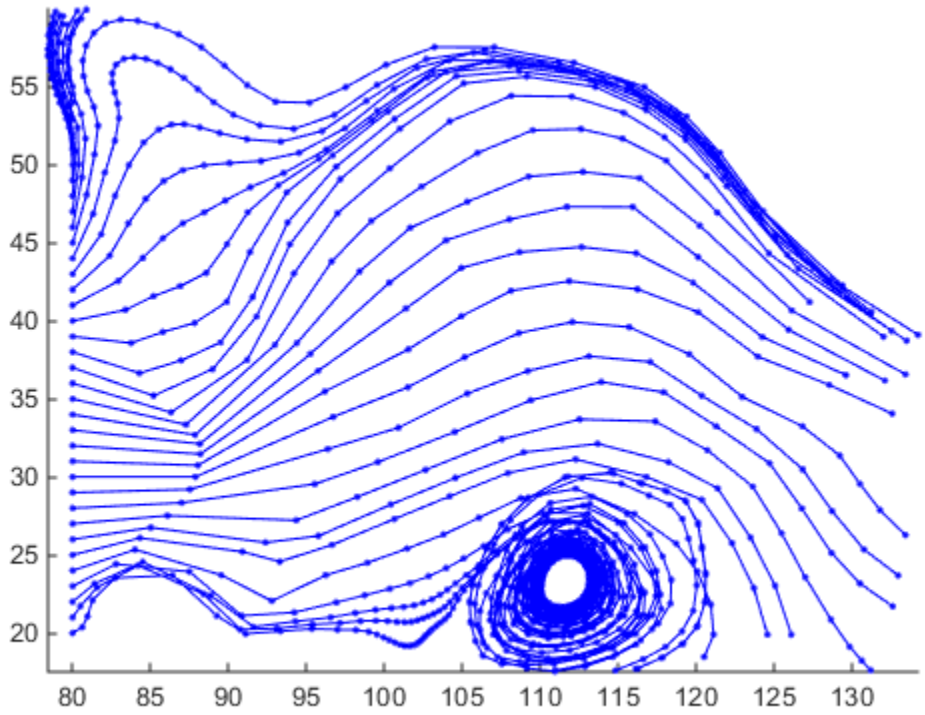
`vertsout = interpstreamspeed(...)` returns a cell array of vertex arrays.

Examples

This example draws streamlines using the vertices returned by `interpstreamspeed`. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Streamlines having widely spaced vertices indicate faster flow; those with closely spaced vertices indicate slower flow.

```
load wind
```

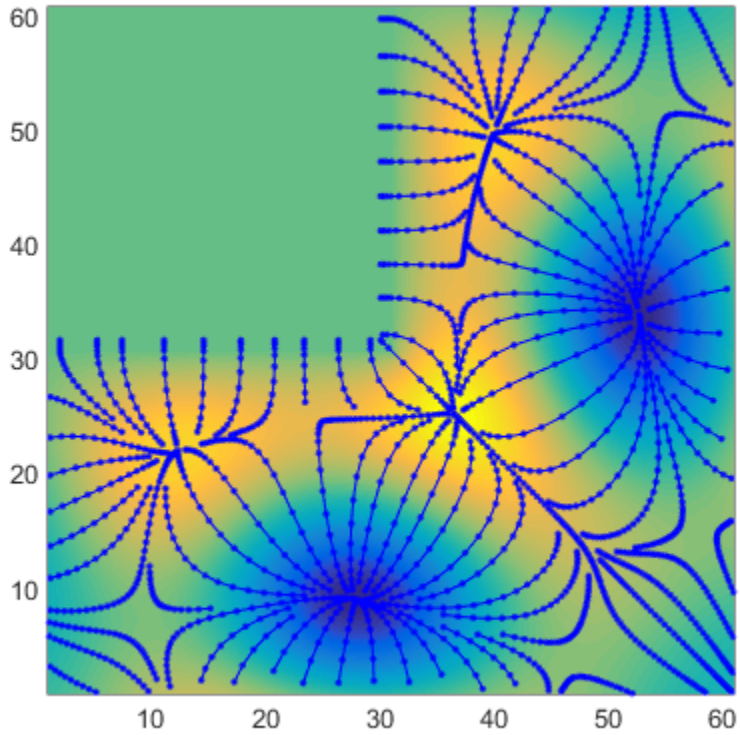
```
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,0.2);
sl = streamline(iverts);
set(sl,'Marker','.')
axis tight; view(2); daspect([1 1 1])
```



This example plots streamlines whose vertex spacing indicates the value of the gradient along the streamline.

```
figure
z = membrane(6,30);
[u v] = gradient(z);
pcolor(z)
```

```
hold on
[verts averts] = streamslice(u,v);
iverts = interpstreamspeed(u,v,verts,15);
sl = streamline(iverts);
set(sl,'Marker','.')
shading interp
axis tight
view(2)
daspect([1,1,1])
hold off
```



See Also

stream2 | stream3 | streamline | streamslice | streamparticles

Introduced before R2006a

intersect

Set intersection of two arrays

Syntax

```
C = intersect(A,B)
C = intersect(A,B,'rows')
[C,ia,ib] = intersect(A,B)
[C,ia,ib] = intersect(A,B,'rows')

[C,ia,ib] = intersect( ___ ,setOrder)

[C,ia,ib] = intersect(A,B,'legacy')
[C,ia,ib] = intersect(A,B,'rows','legacy')
```

Description

`C = intersect(A,B)` returns the data common to both `A` and `B` with no repetitions.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `intersect` returns the values common to both `A` and `B`. The values of `C` are in sorted order.
- If `A` and `B` are tables, then `intersect` returns the set of rows common to both tables. The rows of table `C` are in sorted order.

`C = intersect(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the rows common to both `A` and `B`. The rows of `C` are in sorted order.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia,ib] = intersect(A,B)` also returns index vectors `ia` and `ib`.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `C = A(ia)` and `C = B(ib)`.

- If `A` and `B` are tables, then `C = A(ia,:)` and `C = B(ib,:)`.

`[C,ia,ib] = intersect(A,B,'rows')` also returns index vectors `ia` and `ib`, such that `C = A(ia,:)` and `C = B(ib,:)`.

`[C,ia,ib] = intersect( ___ ,setOrder)` returns `C` in a specific order using any of the input arguments in the previous syntaxes. `setOrder='sorted'` returns the values (or rows) of `C` in sorted order. `setOrder='stable'` returns the values (or rows) of `C` in the same order as `A`, and then `B`. If no value is specified, the default is `'sorted'`.

`[C,ia,ib] = intersect(A,B,'legacy')` and `[C,ia,ib] = intersect( A,B,'rows','legacy')` preserve the behavior of the `intersect` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays, tables, datetime arrays, or duration arrays.

Examples

Intersection of Two Vectors

Define two vectors with values in common.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
```

Find the values common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C =

     4     7
```

Intersection of Two Tables

Define two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
A =
```

```
    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    2       B       true
    3       C       false
    4       D       true
    5       E       false


B =

    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    3       C       false
    5       E       false
    7       G       false
    9       I       false
```

Find the rows common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C =

    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    3       C       false
    5       E       false
```

Intersection of Two Vectors and Their Indices

Define two vectors with values in common.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
```

Find the values common to both `A` and `B`, as well as the index vectors `ia` and `ib`, such that `C = A(ia)` and `C = B(ib)`.

```
[C,ia,ib] = intersect(A,B)
```

```
C =
```

```
     4     7


ia =

     5
     1


ib =

     3
     1
```

Intersection of Two Tables and Their Indices

Define a table, A, of gender, age, and height for five people.

```
A = table(['M';'M';'F';'M';'F'],[27;52;31;46;35],[74;68;64;61;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty' 'Bob' 'Judy'})
```

```
A =

             Gender    Age    Height
             ------    ---    ------
    Ted      M         27     74
    Fred     M         52     68
    Betty    F         31     64
    Bob      M         46     61
    Judy     F         35     64
```

Define a table, B, with rows in common with A.

```
B = table(['F';'M';'F';'F'],[31;47;35;23],[64;68;62;58],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Meg' 'Joe' 'Beth' 'Amy'})
```

```
B =

             Gender    Age    Height
             ------    ---    ------
    Meg      F         31     64
    Joe      M         47     68
    Beth     F         35     62
```

```
    Amy      F          23      58
```

Find the rows common to both `A` and `B`, as well as the index vectors ia and ib, such that `C = A(ia,:)` and `C = B(ib,:)`.

```
[C,ia,ib] = intersect(A,B)
```

```
C =

              Gender    Age    Height
              ------    ---    ------
    Betty     F         31     64


ia =

     3


ib =

     1
```

Two rows that have the same values, but different names, are considered equal. Therefore, we discover that Betty, `A(3,:)`, and Meg, `B(1,:)` have the same gender, age, and height.

Intersection of Rows in Two Matrices

Define two matrices with rows in common.

```
A = [2 2 2; 0 0 1; 1 2 3; 1 1 1];
B = [1 2 3; 2 2 2; 2 2 0];
```

Find the rows common to both `A` and `B` as well as the index vectors `ia` and `ib`, such that `C = A(ia,:)` and `C = B(ib,:)`.

```
[C,ia,ib] = intersect(A,B,'rows')
```

```
C =

     1     2     3
     2     2     2
```

```
ia =

     3
     1


ib =

     1
     2
```

A and B do not need to have the same number of rows, but they must have the same number of columns.

Intersection with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in C.

Specify `'stable'` if you want the values in C to have the same order as in A.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
[C,ia,ib] = intersect(A,B,'stable')
```

```
C =

     7     4


ia =

     1
     5


ib =

     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C,ia,ib] = intersect(A,B,'sorted')
```

```
C =
```

```
     4     7


ia =

     5
     1


ib =

     3
     1
```

Intersection of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN NaN];
```

Find the values common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C =

     5
```

`intersect` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the strings common to both `A` and `B`.

```
[C,ia,ib] = intersect(A,B)
```

```
C =
```

```
    'cat'    'horse'


ia =

     2
     4


ib =

     2
     4
```

`intersect` treats trailing white space in cell arrays of strings as distinct characters.

Intersection of Arrays of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A =

A
B
C


ans =

char
```

Create a 2-by-3 matrix containing elements of numeric type `double`.

```
B = [65 66 67;68 69 70], class(B)
```

```
B =

    65    66    67
    68    69    70
```

```
ans =

double
```

Find the values common to both A and B.

```
[C,ia,ib] = intersect(A,B)
```

```
C =

A
B
C


ia =

     1
     2
     3


ib =

     1
     3
     5
```

`intersect` interprets B as a character array and returns a character array, C.

```
class(C)
```

```
ans =

char
```

Intersection of Char and Cell Array of Strings

Create a character array containing animal names that have three letters.

```
A = ['dog';'cat';'fox';'pig'];
class(A)
```

```
ans =
```

```
char
```

Create a cell array of strings containing animal names of varying lengths.

```
B = {'cat','dog','fish','horse'};
class(B)
```

```
ans =

cell
```

Find the strings common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C =

    'cat'
    'dog'
```

The result, `C`, is a cell array of strings.

```
class(C)
```

```
ans =

cell
```

Preserve Legacy Behavior of `intersect`

Use the `'legacy'` flag to preserve the behavior of `intersect` from R2012b and prior releases in your code.

Find the intersection of `A` and `B` with the current behavior.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
[C1,ia1,ib1] = intersect(A,B)
```

```
C1 =

     4     7


ia1 =
```

```
     5
     1


ib1 =

     3
     1
```

Find the unique elements of A and preserve the legacy behavior.

```
[C2,ia2,ib2] = intersect(A,B,'legacy')
```

```
C2 =

     4     7


ia2 =

     5     4


ib2 =

     4     1
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of strings | tables

Input arrays, specified as numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of strings, or tables.

A and B must be of the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.

- Categorical arrays can combine with cell arrays of strings or single strings.
- Datetime arrays can combine with cell arrays of date strings or single date strings.

If A and B are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither A nor B are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of C are the sorted union of the categories from A and B.

If you specify the `'rows'` option, A and B must have the same number of columns.

If A and B are tables, they must have the same variable names. Conversely, the row names do not matter. Two rows that have the same values, but different names, are considered equal.

If A and B are datetime arrays, they must be consistent with each other in whether they specify a time zone.

Furthermore, A and B can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class.

`setOrder` — Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in C.

Order Flag	Meaning
`'sorted'`	The values (or rows) in C return in sorted order. For example: `C = intersect([7 0 5],[7 1 5],'sorted')` returns `C = [5 7]`.
`'stable'`	The values (or rows) in C return in the same order as they appear in A and B. For example: `C = intersect([7 0 5],[7 1 5],'stable')` returns `C = [7 5]`.

Output Arguments

C — Data common to A and B

vector | matrix | table

Data common to A and B, returned as a vector, matrix, or table. If the inputs A and B are tables, the order of the variables in the resulting table, C, is the same as the order of the variables in A.

The following describes the shape of C when the inputs are vectors or matrices and when the 'legacy' flag is not specified:

- If the 'rows' flag is not specified, then C is a column vector unless both A and B are row vectors.
- If the 'rows' flag is not specified and both A and B are row vectors, then C is a row vector.
- If the 'rows' flag is specified, then C is a matrix containing the rows in common from A and B.
- If A and B have no values (or rows) in common, then C is an empty matrix.

The class of the inputs A and B determines the class of C:

- If the class of A and B are the same, then C is the same class.
- If you combine a char or nondouble numeric class with double, then C is the same class as the nondouble input.
- If you combine a logical class with double, then C is double.
- If you combine a cell array of strings with char, then C is a cell array of strings.
- If you combine a categorical array with a cell array of strings or single string, then C is a categorical array.
- If you combine a datetime array with a cell array of date strings or single date string, then C is a datetime array.

ia — Index to A

column vector

Index to A, returned as a column vector when the 'legacy' flag is not specified. ia identifies the values (or rows) in A that are common to B. If there is a repeated value (or row) in A, then ia contains the index to the first occurrence of the value (or row).

`ib` — Index to `B`

column vector

Index to `B`, returned as a column vector when the `'legacy'` flag is not specified. `ib` identifies the values (or rows) in `B` that are common to `A`. If there is a repeated value (or row) in `B`, then `ib` contains the index to the first occurrence of the value (or row).

More About

Tips

- To find the intersection with respect to a subset of variables from a table, you can use column subscripting. For example, you can use `intersect(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.
- "Combine Categorical Arrays"

See Also

`ismember` | `issorted` | `setdiff` | `setxor` | `sort` | `union` | `unique`

Introduced before R2006a

intmax

Largest value of specified integer type

Syntax

```
v = intmax
v = intmax('classname')
```

Description

`v = intmax` is the largest positive value that can be represented in the MATLAB software with a 32-bit integer. Any value larger than the value returned by `intmax` saturates to the `intmax` value when cast to a 32-bit integer.

`v = intmax('classname')` is the largest positive value in the integer class `classname`. Valid values for the string `classname` are

`'int8'`	`'int16'`	`'int32'`	`'int64'`
`'uint8'`	`'uint16'`	`'uint32'`	`'uint64'`

`intmax('int32')` is the same as `intmax` with no arguments.

Examples

Find the maximum value for a 64-bit signed integer:

```
v = intmax('int64')
v =
  9223372036854775807
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmax`:

```
isequal(x, intmax)
ans =
     1
```

See Also

`intmin` | `realmax` | `int8` | `uint8` | `isa` | `class` | `realmin`

Introduced before R2006a

intmin

Smallest value of specified integer type

Syntax

```
v = intmin
v = intmin('classname')
```

Description

`v = intmin` is the smallest value that can be represented in the MATLAB software with a 32-bit integer. Any value smaller than the value returned by `intmin` saturates to the `intmin` value when cast to a 32-bit integer.

`v = intmin('classname')` is the smallest positive value in the integer class `classname`. Valid values for the string `classname` are

`'int8'`	`'int16'`	`'int32'`	`'int64'`
`'uint8'`	`'uint16'`	`'uint32'`	`'uint64'`

`intmin('int32')` is the same as `intmin` with no arguments.

Examples

Find the minimum value for a 64-bit signed integer:

```
v = intmin('int64')
v =

 -9223372036854775808
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
```

```
 -2147483648
```

Compare the result with the default value returned by `intmin`:

```
isequal(x, intmin)
ans =

     1
```

See Also

`intmax` | `realmin` | `int8` | `uint8` | `isa` | `class` | `realmax`

Introduced before R2006a

inv

Matrix inverse

Syntax

```
Y = inv(X)
```

Description

`Y = inv(X)` returns the inverse of the square matrix `X`. A warning message is printed if `X` is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations $Ax = b$. One way to solve this is with `x = inv(A)*b`. A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator `x = A\b`. This produces the solution using Gaussian elimination, without forming the inverse. See `mldivide` (\) for further information.

Note: MATLAB computes `X^(-1)` and `inv(X)` in the same manner, and both are subject to the same limitations.

Examples

Here is an example demonstrating the difference between solving a linear system by inverting the matrix with `inv(A)*b` and solving it directly with `A\b`. A random matrix `A` of order 500 is constructed so that its condition number, `cond(A)`, is `1.e10`, and its norm, `norm(A)`, is `1`. The exact solution `x` is a random vector of length 500 and the right-hand side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

On a 300 MHz, laptop computer the statements

```
n = 500;
Q = orth(randn(n,n));
d = logspace(0,-10,n);
```

```
A = Q*diag(d)*Q';
x = randn(n,1);
b = A*x;
tic, y = inv(A)*b; toc
err = norm(y-x)
res = norm(A*y-b)
```

produce

```
elapsed_time =
    1.4320
err =
    7.3260e-006
res =
    4.7511e-007
```

while the statements

```
tic, z = A\b, toc
err = norm(z-x)
res = norm(A*z-b)
```

produce

```
elapsed_time =
    0.6410
err =
    7.1209e-006
res =
    4.4509e-015
```

It takes almost two and one half times as long to compute the solution with `y = inv(A)*b` as with `z = A\b`. Both produce computed solutions with about the same error, `1.e-6`, reflecting the condition number of the matrix. But the size of the residuals, obtained by plugging the computed solution back into the original equations, differs by several orders of magnitude. The direct solution produces residuals on the order of the machine accuracy, even though the system is badly conditioned.

The behavior of this example is typical. Using `A\b` instead of `inv(A)*b` is two to three times as fast and produces residuals on the order of machine accuracy, relative to the magnitude of the data.

See Also

`det` | `lu` | `mldivide` | `rref`

Introduced before R2006a

invhilb

Inverse of Hilbert matrix

Syntax

```
H = invhilb(n)
```

Description

`H = invhilb(n)` generates the exact inverse of the exact Hilbert matrix for `n` less than about 15. For larger `n`, `invhilb(n)` generates an approximation to the inverse Hilbert matrix.

Limitations

The exact inverse of the exact Hilbert matrix is a matrix whose elements are large integers. These integers may be represented as floating-point numbers without roundoff error as long as the order of the matrix, `n`, is less than 15.

Comparing `invhilb(n)` with `inv(hilb(n))` involves the effects of two or three sets of roundoff errors:

- The errors caused by representing `hilb(n)`
- The errors in the matrix inversion process
- The errors, if any, in representing `invhilb(n)`

It turns out that the first of these, which involves representing fractions like 1/3 and 1/5 in floating-point, is the most significant.

Examples

`invhilb(4)` is

```
     16    -120     240    -140
```

```
  -120     1200    -2700     1680
   240    -2700     6480    -4200
  -140     1680    -4200     2800
```

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

See Also

`hilb`

Introduced before R2006a

invoke

Invoke method on COM object or interface, or display methods

Syntax

```
S = invoke(h)
S = invoke(h,'methodname')
S = invoke(h,'methodname',arg1,arg2, ...)
S = invoke(h,'custominterfacename')
```

Description

`S = invoke(h)` returns structure array, `S`, containing a list of all methods supported by the object or interface, `h`, along with the prototypes for these methods. If `S` is empty, either there are no properties or methods in the object, or MATLAB cannot read the object's type library. Refer to the COM vendor documentation.

`S = invoke(h,'methodname')` invokes the method specified in the string `methodname`, and returns an output value, if any, in `S`. The data type of the return value depends on the invoked method, which is determined by the control or server.

`S = invoke(h,'methodname',arg1,arg2, ...)` invokes the method specified in the string `methodname` with input arguments `arg1`, `arg2`, etc.

`S = invoke(h,'custominterfacename')` returns an `Interface` object `S`, which is a handle to a custom interface implemented by the COM component. The `h` argument is a handle to the COM object. The `custominterfacename` argument is a string returned by the `interfaces` function.

If the method returns a COM interface, then `invoke` returns a new MATLAB COM object that represents the interface returned. For a description of how MATLAB converts COM types, see "Handling COM Data in MATLAB Software".

COM functions are available on Microsoft Windows systems only.

Examples

Invoke the `Redraw` method in the `mwsamp` control.

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1',[0 0 200 200],f);
h.Radius = 100;
invoke(h,'Redraw');
```

Alternatively, call the method directly.

```
Redraw(h);
```

Display all `mwsamp` methods.

```
invoke(h)
```

```
ans =
   AboutBox = void AboutBox(handle)
   Beep = void Beep(handle)
   FireClickEvent = void FireClickEvent(handle)
            .
            .
```

Getting a Custom Interface Example

Once you have created a COM server, you can query the server component to see if any custom interfaces are implemented. Use the `interfaces` function to return a list of all available custom interfaces:

```
h = actxserver('mytestenv.calculator');
customlist = interfaces(h)
```

```
customlist =
   ICalc1
   ICalc2
   ICalc3
```

To get a handle to the custom interface you want, use the `invoke` function.

```
c1 = invoke(h,'ICalc1')
```

```
c1 =
   Interface.Calc_1.0_Type_Library.ICalc_Interface
```

filUse this handle with COM client functions to access the properties and methods of the object through the selected custom interface.

More About

- “Handling COM Data in MATLAB Software”
- “Custom Interfaces”

See Also

`methods` | `ismethod` | `interfaces`

Introduced before R2006a

ipermute

Inverse permute dimensions of N-D array

Syntax

```
A = ipermute(B,order)
```

Description

`A = ipermute(B,order)` is the inverse of `permute`. `ipermute` rearranges the dimensions of `B` so that `permute(A,order)` will produce `B`. `B` has the same values as `A` but the order of the subscripts needed to access any particular element are rearranged as specified by `order`. All the elements of `order` must be unique.

Examples

Consider the 2-by-2-by-3 array `a`:

```
a = cat(3,eye(2),2*eye(2),3*eye(2))

a(:,:,1) =              a(:,:,2) =
     1     0                   2     0
     0     1                   0     2

a(:,:,3) =
     3     0
     0     3
```

Permuting and inverse permuting `a` in the same fashion restores the array to its original form:

```
B = permute(a,[3 2 1]);
C = ipermute(B,[3 2 1]);
isequal(a,C)
ans=

   1
```

More About

Tips

`permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

See Also

`permute`

Introduced before R2006a

is*

Detect state

Description

These functions detect the state of MATLAB entities:

`isa`	Detect object of given MATLAB class or Java class
`isappdata`	Determine if object has specific application-defined data
`isbetween`	Array elements occurring within date and time interval
`iscalendarduration`	Determine if input is duration array
`iscategorical`	Determine whether input is categorical array
`iscategory`	Test for categorical array categories
`iscell`	Determine if input is cell array
`iscellstr`	Determine if input is cell array of strings
`ischar`	Determine if input is character array
`iscolumn`	Determine whether input is column vector
`iscom`	Determine if input is Component Object Model (COM) object
`isdatetime`	Determine if input is datetime array
`isdir`	Determine if input is folder
`isdst`	Datetime values occurring during daylight saving time
`isduration`	Determine if input is duration array
`isempty`	Determine if input is empty array
`isequal`	Determine if arrays are numerically equal
`isequaln`	Determine if arrays are numerically equal, treating `NaN`s as equal
`isevent`	Determine if input is Component Object Model (COM) object event
`isfield`	Determine if input is MATLAB structure array field
`isfinite`	Detect finite elements of array

isfloat	Determine if input is floating-point array
ishandle	Detect valid graphics object handles
ishold	Determine if graphics hold state is on
isinf	Detect infinite elements of array
isinteger	Determine if input is integer array
isinterface	Determine if input is Component Object Model (COM) interface
isjava	Determine if input is Java object
iskeyword	Determine if input is MATLAB keyword
isletter	Detect elements that are alphabetic letters
islogical	Determine if input is logical array
ismac	Determine if running MATLAB for Macintosh OS X platform
ismatrix	Determine whether input is matrix
ismember	Detect members of specific set
ismethod	Determine if input is object method
ismissing	Find table elements with missing values
isnan	Detect elements of array that are not a number (NaN)
isnumeric	Determine if input is numeric array
isobject	Determine if input is MATLAB object
isordinal	Determine whether input is ordinal categorical array
ispc	Determine if running MATLAB for PC (Windows) platform
isprime	Detect prime elements of array
isprop	Determine if input is object property
isprotected	Determine whether categories of categorical array are protected
isreal	Determine if all array elements are real numbers
isrow	Determine whether input is row vector
isscalar	Determine if input is scalar
issorted	Determine if set elements are in sorted order

`isspace`	Detect space characters in array
`issparse`	Determine if input is sparse array
`isstrprop`	Determine if string is of specified category
`isstruct`	Determine if input is MATLAB structure array
`isstudent`	Determine if Student Version of MATLAB
`istable`	Determine whether input is table
`isundefined`	Find undefined elements in categorical array
`isunix`	Determine if running MATLAB for UNIX[a] platform.
`isvarname`	Determine if input is valid variable name
`isvector`	Determine if input is vector
`isweekend`	Datetime values occurring during weekend

a. UNIX is a registered trademark of The Open Group in the United States and other countries.

See Also

`isa` | `exist`

isa

Determine if input is object of specified class

Syntax

```
tf = isa(obj,ClassName)
tf = isa(obj,classCategory)
```

Description

`tf = isa(obj,ClassName)` returns `true` if `obj` is an instance of the class specified by *`ClassName`*, and `false` otherwise. `isa` also returns true if `obj` is an instance of a class that is derived from *`ClassName`*.

`obj` can be any MATLAB variable.

`ClassName` can be any of the following:

- Name of any MATLAB class or fundamental type
- Name of a Java, or .NET class

MATLAB Fundamental Types

`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer

`'logical'`	Logical `true` or `false`
`'char'`	Character or string
`'struct'`	Structure array
`'cell'`	Cell array
`'function_handle'`	function handle

`tf = isa(obj,classCategory)` returns `true` if `obj` is an instance of any of the classes in the specified *`classCategory`*, and `false` otherwise. `isa` also returns true if `obj` is an instance of a class that is derived from any of the classes in *`classCategory`*.

`classCategory` can be `'numeric'`, `'float'`, or `'integer'`, representing a category of numeric types:

Categories of Numeric Types

`'numeric'`	Integer or floating-point array (`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`)
`'float'`	Single- or double-precision floating-point array (`double`, `single`)
`'integer'`	Signed or unsigned integer array (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`)

To test for a sparse array, use `issparse`. To test for a complex array, use `~isreal`.

Examples

These examples show the values returned by `isa` when passed different types:

Determine if the value returned by the `pi` function is of class `double`:

```
isa(pi,'double')
```

```
ans =

     1
```

More generally, determine if the value returned by the `pi` function is a numeric value:

```
isa(pi,'numeric')
```

```
ans =

     1
```

`isa` also returns `true` for the `float` category because the class `double` is a floating-point type. However, `pi` does not return an integer type:

```
isa(pi,'integer')

ans =

     0
```

Determine if the 2-by-3 array returned by `true` is of type logical:

```
isa(true(2,3),'logical')

ans =

     1
```

Identify an instance of the MATLAB `containers.Map` class:

```
colorcodes = containers.Map({'Color','RGB'},...
    {'Yellow',uint8([255,255,0])});
isa(colorcodes,'containers.Map')

ans =

     1
```

The map key, `RGB`, references a `uint8` array:

```
isa(colorcodes('RGB'),'integer')

ans =

     1
```

Specifying a particular integer class provides more specific testing:

```
if strcmp(colorcodes('Color'),'Yellow') && ...
        isa(colorcodes('RGB'),'uint8')
    disp '''The Color is Yellow and the RGB numbers are uint8 values'''
```

```
end
'The Color is Yellow and the RGB numbers are uint8 values'
```

See Also

`class` | `is*` | `isnumeric` | `isfloat` | `isinteger` | `exist`

Introduced before R2006a

isappdata

True if application-defined data exists

Syntax

```
tf = isappdata(h,name)
```

Description

`tf = isappdata(h,name)` returns `logical(1)` if application-defined data exists and these conditions are met:

- The application data has the specified `name` value.
- The application data is associated with the UI component, `h`.

Otherwise, `isappdata` returns `logical(0)`

See Also

`getappdata` | `rmappdata` | `setappdata`

Introduced before R2006a

isbanded

Determine if matrix is within specific bandwidth

Syntax

```
tf = isbanded(A,lower,upper)
```

Description

`tf = isbanded(A,lower,upper)` returns logical `1` (`true`) if matrix A is within the specified lower bandwidth, `lower`, and upper bandwidth, `upper`; otherwise, it returns logical `0` (`false`).

Examples

Test Square Matrix

Create a 5-by-5 square matrix.

```
A = [2 3 0 0 0 ; 1 -2 -3 0 0; 0 -1 2 3 0 ; 0 0 1 -2 -3; 0 0 0 -1 2]
```

```
A =

     2     3     0     0     0
     1    -2    -3     0     0
     0    -1     2     3     0
     0     0     1    -2    -3
     0     0     0    -1     2
```

The result is a square matrix with nonzero diagonals above and below the main diagonal.

Specify both bandwidths, `lower` and `upper`, as `1` to test if `A` is tridiagonal.

```
isbanded(A,1,1)
```

```
ans =

     1
```

The result is logical `1` (`true`).

Test if `A` has nonzero elements below the main diagonal by specifying `lower` as `0`.

```
isbanded(A,0,1)
```

```
ans =

     0
```

The result is logical `0` (`false`) because `A` has nonzero elements below the main diagonal.

Test Nonsquare Matrix

Create a 3-by-5 matrix.

```
A = [1 0 0 0 0; 2 1 0 0 0; 3 2 1 0 0]
```

```
A =

     1     0     0     0     0
     2     1     0     0     0
     3     2     1     0     0
```

The result is a rectangular matrix.

Test if `A` has nonzero elements above the main diagonal.

```
isbanded(A,2,0)
```

```
ans =

     1
```

The result is logical `1` (`true`) because the elements above the main diagonal are all zero.

Test Sparse Block Matrix

Create a 100-by-100 sparse block matrix.

```
B = kron(speye(25),ones(4));
```

Test if `B` has a lower and upper bandwidth of `1`.

```
isbanded(B,1,1)
```

```
ans =

     0
```

The result is logical `0` (`false`) because the nonzero blocks centered on the main diagonal are larger than 2-by-2.

Test if `B` has a lower and upper bandwidth of `3`.

```
isbanded(B,3,3)
```

```
ans =

     1
```

The result is logical `1` (`true`). The matrix, `B`, has an upper and lower bandwidth of `3` since the nonzero diagonal blocks are 4-by-4.

Input Arguments

`A` — Input array

numeric array

Input array, specified as a numeric array. `isbanded` returns logical `0` (`false`) if `A` has more than two dimensions.

Data Types: `single` | `double`
Complex Number Support: Yes

`lower` — Lower bandwidth

nonnegative integer scalar

Lower bandwidth, specified as a nonnegative integer scalar. The lower bandwidth is the number of nonzero diagonals below the main diagonal. `isbanded` returns logical `0` (`false`) if there are nonzero elements below the boundary diagonal, `diag(A,-lower)`.

`upper` — Upper bandwidth

nonnegative integer scalar

Upper bandwidth, specified as a nonnegative integer scalar. The upper bandwidth is the number of nonzero diagonals above the main diagonal. `isbanded` returns logical `0` (`false`) if there are nonzero elements above the boundary diagonal, `diag(A,upper)`.

More About

Tips

- Use the `bandwidth` function to find the upper and lower bandwidths of a given matrix.
- Use `isbanded` to test for several different matrix structures by specifying appropriate upper and lower bandwidths. The table below lists some common tests.

Lower Bandwidth	Upper Bandwidth	Function Call	Tests for
`0`	`0`	`isbanded(A,0,0)`	Diagonal matrix
`1`	`1`	`isbanded(A,1,1)`	Tridiagonal matrix
`0`	`size(A,2)`	`isbanded(A,0,size(A,`	Upper triangular matrix
`size(A,1)`	`0`	`isbanded(A,size(A,1)`	Lower triangular matrix
`1`	`size(A,2)`	`isbanded(A,1,size(A,`	Upper Hessenberg matrix
`size(A,1)`	`1`	`isbanded(A,size(A,1)`	Lower Hessenberg matrix

See Also

`bandwidth` | `diag` | `isdiag` | `istril` | `istriu`

Introduced in R2014a

isbetween

Determine elements within date and time interval

Syntax

```
tf = isbetween(t,tlower,tupper)
```

Description

`tf = isbetween(t,tlower,tupper)` returns an array the same size as `t` containing logical `1` (true) where the corresponding element of `t` is a datetime that lies within the closed interval specified by the corresponding elements of `tlower` and `tupper`. The output `tf` indicates which elements of `t` satisfy:

```
tlower <= t <= tupper
```

Examples

Determine if Dates Occur Within Interval

Define a lower bound and an upper bound for dates.

```
tlower = datetime(2014,05,16)
```

```
tlower =

   16-May-2014
```

```
tupper = '23-May-2014'
```

```
tupper =

23-May-2014
```

`tlower` and `tupper` can be `datetime` arrays or strings. Here, `tlower` is a `datetime` array and `tupper` is a single string.

Create an array of datetime values and determine if each datetime lies within the interval bounded by `tlower` and `tupper`.

```
t = tlower + caldays(2:2:10)
```

```
t =

   18-May-2014   20-May-2014   22-May-2014   24-May-2014   26-May-2014
```

```
tf = isbetween(t,tlower,tupper)
```

```
tf =

     1     1     1     0     0
```

Input Arguments

`t` — Input date and time

`datetime` array | cell array of datetime strings | datetime string

Input date and time, specified as a `datetime` array, a cell array of datetime strings, or a single datetime string.

`tlower` — Lower bound of date and time interval

`datetime` array | cell array of datetime strings | datetime string

Lower bound of date and time interval, specified as a `datetime` array, a cell array of datetime strings, or a single datetime string.

`tupper` — Upper bound of date and time interval

`datetime` array | cell array of date strings | datetime string

Upper bound of date and time interval, specified as a `datetime` array, a cell array of datetime strings, or a single datetime string.

See Also

`ge` | `gt` | `ismember` | `le` | `lt`

Introduced in R2014b

iscalendarduration

Determine if input is calendar duration array

Syntax

```
tf = iscalendarduration(t)
```

Description

`tf = iscalendarduration(t)` returns logical `1` (`true`) if `t` is a `calendarDuration` array. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Array Contains Calendar Duration Values

Determine if the output of an arithmetic calculation is a `calendarDuration` array.

Add two `calendarDuration` arrays.

```
d1 = calyears(1:4);
d2 = caldays(1:4);
d = d1 + d2
```

```
d =

   1y 1d   2y 2d   3y 3d   4y 4d
```

Determine if the output is a `calendarDuration` array.

```
tf = iscalendarduration(d)
```

```
tf =
```

```
1
```

Input Arguments

`t` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `t` can be any data type.

See Also

`calendarDuration` | `isdatetime` | `isduration`

Introduced in R2014b

iscategorical

Determine whether input is categorical array

Syntax

```
tf = iscategorical(A)
```

Description

`tf = iscategorical(A)` returns logical `1` (`true`) if `A` is a categorical array. Otherwise, `iscategorical` returns logical `0` (`false`).

Examples

Determine Whether Workspace Variable Is Categorical Array

Create a workspace variable, `A`.

```
A = categorical({'red' 'green' 'violet'; 'orange' 'red' 'yellow'})
```

```
A =

     red          green      violet
     orange       red        yellow
```

Verify that the workspace variable, `A`, is a categorical array.

```
tf = iscategorical(A)
```

```
tf =

     1
```

A is a 2-by-3 categorical array.

Input Arguments

A — Input variable

workspace variable

Input variable, specified as a workspace variable. A can be any data type.

See Also

`categorical` | `iscell` | `islogical` | `isnumeric` | `isobject` | `isstruct` | `istable`

iscategory

Test for categorical array categories

Syntax

```
tf = iscategory(A,catnames)
```

Description

`tf = iscategory(A,catnames)` returns an array containing logical `1` (`true`) where the data in catnames is a category of A. Otherwise, `iscategory` returns logical `0` (`false`).

`tf` is the same size as `catnames`.

Examples

Test for Categories

Create an ordinal categorical array, `A`.

```
A = categorical({'shirt' 'pants'; 'pants' 'hat'; 'shirt' 'pants'})
```

```
A =

     shirt      pants
     pants      hat
     shirt      pants
```

`A` is a 3-by-2 categorical array.

Test if the articles of clothing, `shirt`, `pants`, `socks`, and `shoes`, are categories of `A`.

```
catnames = {'shirt' 'pants' 'socks' 'shoes'};
tf = iscategory(A,catnames)
```

```
tf =
```

```
     1     1     0     0
```

`shirt` and `pants` are categories of `A`, but `socks`, and `shoes` are not.

`iscategory` does not tell us anything about the category, `hat`, which we did not include in `catnames`.

Test for Category with No Corresponding Data

Create a categorical array, `A`.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'},...
    {'boat' 'car' 'plane' 'train'})
```

```
A =

     plane      car      train      car      plane
```

`A` is a 1-by-5 categorical array.

Test to see if `boat` is a category in `A`.

```
tf = iscategory(A,'boat')
```

```
tf =

     1
```

`iscategory` returns true, even though `A` does not contain any values from the category `boat`.

Input Arguments

`A` — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`catnames` — Category names

string | cell array of strings | categorical array

Category names, specified as a string, cell array of strings, or categorical array.

See Also

addcats | categorical | categories | ismember | mergecats | removecats | renamecats | reordercats | setcats | unique

iscell

Determine whether input is cell array

Syntax

```
tf = iscell(A)
```

Description

`tf = iscell(A)` returns logical `1` (`true`) if `A` is a cell array and logical `0` (`false`) otherwise.

Examples

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
A{2,1} = 3+7i;
A{2,2} = -pi:pi/10:pi;

iscell(A)

ans =

     1
```

See Also

`cell` | `istable` | `iscellstr` | `isstruct` | `isnumeric` | `islogical` | `isobject` | `isa` | `is*`

Introduced before R2006a

iscellstr

Determine whether input is cell array of strings

Syntax

```
tf = iscellstr(A)
```

Description

`tf = iscellstr(A)` returns logical `1` (`true`) if `A` is a cell array of strings (or an empty cell array), and logical `0` (`false`) otherwise. A cell array of strings is a cell array where every element is a character array.

Examples

```
A{1,1} = 'Thomas Lee';
A{1,2} = 'Marketing';
A{2,1} = 'Allison Jones';
A{2,2} = 'Development';

iscellstr(A)

ans =

     1
```

See Also

`cellstr` | `istable` | `iscategorical` | `iscell` | `isstrprop` | `strings` | `char` | `isstruct` | `isa` | `is*`

Introduced before R2006a

ischar

Determine whether item is character array

Syntax

```
tf = ischar(A)
```

Description

`tf = ischar(A)` returns logical `1` (`true`) if `A` is a character array and logical `0` (`false`) otherwise.

Examples

Given the following cell array,

```
C{1,1} = magic(3);           % double array
C{1,2} = 'John Doe';         % char array
C{1,3} = 2 + 4i              % complex double

C =

    [3x3 double]    'John Doe'    [2.0000+ 4.0000i]
```

`ischar` shows that only `C{1,2}` is a character array.

```
for k = 1:3
x(k) = ischar(C{1,k});
end

x

x =

     0     1     0
```

See Also

`char` | `strings` | `isletter` | `isspace` | `isstrprop` | `iscellstr` | `isnumeric` | `isa` | `is*`

Introduced before R2006a

iscolumn

Determine whether input is column vector

Syntax

```
iscolumn(V)
```

Description

`iscolumn(V)` returns logical 1 (`true`) if `size`(V) returns `[n 1]` with a nonnegative integer value `n`, and logical 0 (`false`) otherwise.

Examples

Determine if a vector is a column. This example is a row so `iscolumn` returns 0:

```
V = rand(1,5);
iscolumn(V)
ans =
     0
```

Transpose the vector to make it a column. `iscolumn` returns 1:

```
V1 = V';
iscolumn(V1)
ans =
     1
```

See Also

`ismatrix` | `isrow` | `isscalar` | `isvector`

iscom

Determine whether input is COM or ActiveX object

Syntax

```
tf = iscom(h)
```

Description

`tf = iscom(h)` returns logical `1` (`true`) if handle `h` is a COM or Microsoft ActiveX object. Otherwise, returns logical `0` (`false`).

COM functions are available on Microsoft Windows systems only.

Examples

Test an instance of a Microsoft Excel application.

```
h = actxserver('Excel.Application');
iscom(h)
```

MATLAB displays `true`, indicating object `h` is a COM object.

Test an Excel interface:

```
h = actxserver('Excel.Application');
% Create a workbooks object
w = get(h,workbooks');
iscom(w)
```

MATLAB displays `false`, indicating object `w` is not a COM object.

More About

- "MATLAB COM Integration"

Introduced before R2006a

isdatetime

Determine if input is datetime array

Syntax

```
tf = isdatetime(t)
```

Description

`tf = isdatetime(t)` returns logical `1` (`true`) if `t` is a datetime array. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Array Contains Datetime Values

Define an array.

```
A = [datetime('now');datetime('tomorrow');'January 15, 2015']
```

```
A =

   23-Feb-2015 09:54:40
   24-Feb-2015 00:00:00
   15-Jan-2015 00:00:00
```

Determine if the array is a `datetime` array.

```
tf = isdatetime(A)
```

```
tf =

     1
```

Input Arguments

`t` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `t` can be any data type.

See Also

`datetime` | `iscalendarduration` | `isduration`

Introduced in R2014b

isdiag

Determine if matrix is diagonal

Syntax

```
tf = isdiag(A)
```

Description

`tf = isdiag(A)` returns logical `1` (`true`) if `A` is a diagonal matrix; otherwise, it returns logical `0` (`false`).

Examples

Test Diagonal Matrix

Create a 4-by-4 identity matrix.

```
I = eye(4)
```

```
I =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Test to see if the matrix is diagonal.

```
isdiag(I)
```

```
ans =

     1
```

The result is logical 1 (`true`) because all of the nonzero elements in `I` are on the main diagonal.

Test Banded Matrix

Create a matrix with nonzero elements on the main and first diagonals.

```
A = 3*eye(4) + diag([2 2 2],1)
```

```
A =

     3     2     0     0
     0     3     2     0
     0     0     3     2
     0     0     0     3
```

Test to see if the matrix is diagonal.

```
isdiag(A)
```

```
ans =

     0
```

The matrix is not diagonal since there are nonzero elements above the main diagonal.

Create a new matrix, `B`, from the main diagonal elements of `A`.

```
B = diag(diag(A));
```

Test to see if `B` is a diagonal matrix.

```
isdiag(B)
```

```
ans =

     1
```

The result is logical 1 (`true`) because there are no nonzero elements above or below the main diagonal of `B`.

Input Arguments

`A` — Input array

numeric array

Input array, specified as a numeric array. `isdiag` returns logical `0` (`false`) if `A` has more than two dimensions.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Diagonal Matrix

A matrix is diagonal if all elements above and below the main diagonal are zero. Any number of the elements on the main diagonal can also be zero.

For example, the 4-by-4 identity matrix,

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is a diagonal matrix. Diagonal matrices are typically, but not always, square.

Tips

- Use the `diag` function to produce diagonal matrices for which `isdiag` returns logical `1` (`true`).
- The functions `isdiag`, `istriu`, and `istril` are special cases of the function `isbanded`, which can perform all of the same tests with suitably defined upper and lower bandwidths. For example, `isdiag(A) == isbanded(A,0,0)`.

See Also

`diag` | `istril` | `istriu` | `tril` | `triu`

Introduced in R2014a

isdir

Determine whether input is folder

Syntax

```
tf = isdir('A')
```

Description

`tf = isdir('A')` returns logical `1` (`true`) if `A` is a folder. Otherwise, it returns logical `0` (`false`).

Examples

Run:

```
tf=isdir('myfiles/results')
```

MATLAB returns

```
tf =
 1
```

indicating that `myfiles/results` is a folder.

See Also

`dir` | `is*`

Introduced before R2006a

isdst

Determine daylight saving time elements

Syntax

```
tf = isdst(t)
```

Description

`tf = isdst(t)` returns an array the same size as `t` containing logical 1 (`true`) where the corresponding element of `t` is a datetime that occurs during Daylight Saving Time, and logical 0 (`false`) otherwise. `isdst` returns `false` for all elements when the `TimeZone` property of `t` is empty (`''`).

Examples

Determine If Datetime Occurs During Daylight Saving Time

```
t = datetime(2014,3,7:11,'TimeZone','America/New_York')
```

```
t =

   07-Mar-2014   08-Mar-2014   09-Mar-2014   10-Mar-2014   11-Mar-2014
```

```
tf = isdst(t)
```

```
tf =

     0     0     0     1     1
```

March 10 and March 11, 2014 in the `America/New_York` time zone occur during daylight saving time.

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

datetime Properties | `isweekend` | `tzoffset`

Introduced in R2014b

isduration

Determine if input is duration array

Syntax

```
tf = isduration(t)
```

Description

`tf = isduration(t)` returns logical `1` (`true`) if `t` is a `duration` array. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Array Contains Duration Values

Determine if the output of an arithmetic calculation is a `duration` array.

Subtract a `datetime` array from another.

```
t1 = datetime(2014,03,16:17);
t2 = datetime(2014,03,20);
dt = t2 - t1
```

```
dt =

   96:00:00   72:00:00
```

Determine if the output is a `duration` array.

```
tf = isduration(dt)
```

```
tf =
```

1

Input Arguments

`t` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `t` can be any data type.

See Also

`duration` | `iscalendarduration` | `isdatetime`

Introduced in R2014b

isEdge

Class: TriRep

(Will be removed) Test if vertices are joined by edge

Note: `isEdge(TriRep)` will be removed in a future release. Use `isConnected(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
TF = isEdge(TR, V1, V2)
TF = isEdge(TR, EDGE)
```

Description

`TF = isEdge(TR, V1, V2)` returns an array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation. `V1`, `V2` are column vectors representing the indices of the vertices in the mesh, that is, indices into the vertex coordinate arrays.

`TF = isEdge(TR, EDGE)` specifies the edge start and end indices in matrix format.

Input Arguments

`TR`	Triangulation representation.
`V1, V2`	Column vectors of mesh vertices.
`EDGE`	Matrix of size `n`-by-2 where `n` is the number of query edges.

Output Arguments

`TF`	Array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to query the presence of an edge between pairs of points.

```
load trimesh2d
trep = TriRep(tri, x,y);
```

Test if vertices 3 and 117 are connected by an edge

```
isEdge(trep, 3, 117)
```

Test if vertices 3 and 164 are connected by an edge

```
isEdge(trep, 3, 164)
```

Example 2

Direct query of a 3-D Delaunay triangulation created using `DelaunayTri`.

```
X = rand(10,3)
dt = DelaunayTri(X)
```

Test if vertices 2 and 7 are connected by an edge

```
isEdge(dt, 2, 7);
```

See Also

`triangulation` | `delaunayTriangulation`

isempty

Determine whether array is empty

Syntax

```
TF = isempty(A)
```

Description

`TF = isempty(A)` returns logical 1 (`true`) if `A` is an empty array and logical 0 (`false`) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

Examples

```
B = rand(2,2,2);
B(:,:,:) = [];

isempty(B)

ans = 1
```

See Also

`is*`

Introduced before R2006a

isempty (tscollection)

Determine whether `tscollection` object is empty

Syntax

```
isempty(tsc)
```

Description

`isempty(tsc)` returns a logical value for `tscollection` object `tsc`, as follows:

- `1` — When `tsc` contains neither `timeseries` members nor a time vector
- `0` — When `tsc` contains either `timeseries` members or a time vector

See Also

`timeseries` | `length (tscollection)` | `size (tscollection)` | `tscollection`

Introduced before R2006a

isequal

Determine array equality

Syntax

```
tf = isequal(A,B)
tf = isequal(A1,A2,...,An)
```

Description

`tf = isequal(A,B)` returns logical `1` (`true`) if `A` and `B` are the same size and their contents are of equal value; otherwise, it returns logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays. `isequal` ignores the data type of the values in determining whether they are equal. For cell arrays, categorical arrays, tables, or structures, the function returns logical `1` (`true`) only when all elements and properties are equal. Undefined categorical elements, `NaN` (Not a Number), or `NaT` (Not a Time) values are considered to be unequal to other elements, as well as themselves.

`tf = isequal(A1,A2,...,An)` returns logical `1` (`true`) if all the inputs are numerically equal.

Examples

Compare Two Numeric Matrices

Create two numeric matrices and compare them for equality.

```
A = zeros(3,3)+1e-20;
B = zeros(3,3);
tf = isequal(A,B)
```

```
tf =

     0
```

The function returns logical `0` (`false`) because the matrices differ by a very small amount and are not *exactly* equal.

Compare Two Structures

Create two structures and specify the fields in a different order.

```
A = struct('field1',0.005,'field2',2500);
B = struct('field2',2500,'field1',0.005);
```

Compare the structures for equality.

```
tf = isequaln(A,B)
```

```
tf =

     1
```

Even though the ordering of the fields in each structure is different, `isequal` treats them as the same because the values are equal.

Comparing Numeric Values with Special Nonnumeric Values

Compare the logical value `true` to the double integer `1`.

```
isequal(true,1)
```

```
ans =

     1
```

Notice that `isequal` does not consider data type when it tests for equality.

Similarly, compare `'A'` to the ASCII-equivalent integer, `65`.

```
isequal('A',65)
```

```
ans =

     1
```

The result is logical `1` (`true`) since `double('A')` equals `65`.

Compare Vectors Containing NaN Values

Create three vectors containing NaN values.

```
A1 = [1 NaN NaN];
A2 = [1 NaN NaN];
A3 = [1 NaN NaN];
```

Compare the vectors for equality.

```
tf = isequal(A1,A2,A3)
```

```
tf =

     0
```

The result is logical `0` (`false`) because `isequal` does not treat NaN values as equal to each other.

Compare Two Datetime Values

Determine if midnight on January 13, 2013 in Anchorage, Alaska is equal to 11 AM on the same date in Cairo.

```
t1 = datetime(2013,1,13,0,0,0,'TimeZone','America/Anchorage');
t2 = datetime(2013,1,13,11,0,0,'TimeZone','Africa/Cairo');
tf = isequal(t1,t2)
```

```
tf =

     1
```

Add 8 months to the date, and compare the datetime values for equality.

```
t1 = datetime(2013,9,13,0,0,0,'TimeZone','America/Anchorage');
t2 = datetime(2013,9,13,11,0,0,'TimeZone','Africa/Cairo');
tf = isequal(t1,t2)
```

```
tf =

     0
```

The datetime values are no longer equal since Cairo does not observe daylight saving time.

Input Arguments

`A,B` — Inputs to be compared

numeric arrays | categorical arrays | cell arrays | tables | structures | ...

Inputs to be compared, specified as numeric arrays, categorical arrays, cell arrays, tables, or structures. Also, you can specify the inputs as logical arrays, character arrays, datetime arrays, duration arrays, calendarDuration arrays, or objects. The numeric types of `A` and `B` do not have to match.

You can compare a categorical array to a cell array of strings of the same size, or a single categorical element to a single string.

You can compare a datetime array to a cell array of date strings or a single date string.

If inputs `A` and `B` are *both*

- Structures -- Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays -- Must have the same sets of categories, including their order.
- Categorical arrays that are not ordinal -- Can have different sets of categories, and `isequal` compares the category names of each pair of elements.
- Datetime arrays -- `isequal` ignores time zone and display format when it compares points in time.
- Objects of different classes -- `isequal` returns logical `0` (`false`). This applies even when the objects have the same properties and their values match.

`A1,A2,...,An` — Series of inputs to be compared

numeric arrays | categorical arrays | cell arrays | tables | structures | ...

Series of inputs to be compared, specified as numeric arrays, categorical arrays, cell arrays, tables, or structures. Also, you can specify the inputs as logical arrays, character arrays, datetime arrays, duration arrays, calendarDuration arrays, or objects. The numeric types of the inputs do not have to match.

You can compare categorical arrays to cell arrays of strings of the same size, or single categorical elements to single strings.

You can compare a datetime array to a cell array of date strings or a single date string.

If the inputs are *all*

- Structures -- Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays -- Must have the same sets of categories, including their order.
- Categorical arrays that are not ordinal -- Can have different sets of categories, and `isequal` compares the category names of each pair of elements.
- Datetime arrays -- `isequal` ignores time zone and display format when it compares points in time.
- Objects of different classes -- `isequal` returns logical `0` (`false`). This applies even when the objects have the same properties and their values match.

More About

Tips

- When comparing handle objects, use `==` to test whether objects have the same handle. Use `isequal` to determine if objects with different handles have equal property values.
- Use `isequaln` if you want to test for equality and treat `NaN` or `NaT` values as equal.

See Also

`eq` | `is*` | `isa*` | `isequaln` | `relational operators` | `strcmp`

Introduced before R2006a

isenum

Determine if variable is enumeration

Syntax

```
tf = isenum(e)
```

Description

`tf = isenum(e)` returns logical 1 (`true`) if `e` is an enumeration. Otherwise, it returns logical 0 (`false`). Empty enumeration objects return `true`.

If `e` is a heterogeneous array, `isenum` always returns `false`.

Examples

Test for Enumeration

Determine if a variable is an enumeration.

The `PPM` class defines enumerations for three levels:

```
classdef PPM < int32
   enumeration
      High (1000)
      Medium (100)
      Low (10)
   end
end
```

Create a variable representing a level. Use `isenum` to determine if the variable an enumeration:

```
currentLevel = PPM.High;
isenum(currentLevel)
```

```
ans =
```

1

Input Arguments

e — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as any MATLAB variable.

More About

- "Working with Enumerations"

See Also

`islogical` | `isnumeric` | `isobject` | `isstruct`

Introduced in R2015a

isequaln

Determine array equality, treating NaN values as equal

Syntax

```
tf = isequaln(A,B)
tf = isequaln(A1,A2,...,An)
```

Description

`tf = isequaln(A,B)` returns logical `1` (`true`) if `A` and `B` are the same size and their contents are of equal value; otherwise, it returns logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays. `isequaln` ignores the data type of the values in determining whether they are equal. For cell arrays, categorical arrays, tables, or structures, the function returns logical `1` (`true`) only when all elements and properties are equal. Undefined categorical elements, `NaN` (Not a Number), or `NaT` (Not a Time) values are considered to be equal to other such values.

`tf = isequaln(A1,A2,...,An)` returns logical `1` (`true`) if all the inputs are numerically equal.

Examples

Compare Two Numeric Matrices

Create two numeric matrices and compare them for equality.

```
A = zeros(3,3)+1e-20;
B = zeros(3,3);
tf = isequaln(A,B)
```

```
tf =

     0
```

The function returns logical `0` (`false`) because the matrices differ by a very small amount and are not *exactly* equal.

Compare Two Structures

Create two structures and specify the fields in a different order.

```
A = struct('field1',0.005,'field2',2500);
B = struct('field2',2500,'field1',0.005);
```

Compare the structures for equality.

```
tf = isequaln(A,B)
```

```
tf =

     1
```

Even though the ordering of the fields in each structure is different, `isequaln` treats them as the same because the values are equal.

Comparing Numeric Values with Special Nonnumeric Values

Compare the logical value `true` to the double integer `1`.

```
isequaln(true,1)
```

```
ans =

     1
```

Notice that `isequaln` does not consider data type when it tests for equality.

Similarly, compare `'A'` to the ASCII-equivalent integer, `65`.

```
isequaln('A',65)
```

```
ans =

     1
```

The result is logical `1` (`true`) since `double('A')` equals `65`.

Compare Vectors Containing NaN Values

Create three vectors containing NaN values.

```
A1 = [1 NaN NaN];
A2 = [1 NaN NaN];
A3 = [1 NaN NaN];
```

Compare the vectors for equality.

```
tf = isequaln(A1,A2,A3)
```

```
tf =

     1
```

The result is logical `1` (`true`) because `isequaln` treats the NaN values as equal to each other.

Input Arguments

`A,B` — Inputs to be compared

numeric arrays | categorical arrays | cell arrays | tables | structures | ...

Inputs to be compared, specified as numeric arrays, categorical arrays, cell arrays, tables, or structures. Also, you can specify the inputs as logical arrays, character arrays, datetime arrays, duration arrays, calendarDuration arrays, or objects. The numeric types of `A` and `B` do not have to match.

You can compare a categorical array to a cell array of strings of the same size, or a single categorical element to a single string.

You can compare a datetime array to a cell array of date strings or a single date string.

If inputs `A` and `B` are *both*

- Structures -- Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays -- Must have the same sets of categories, including their order.
- Categorical arrays that are not ordinal -- Can have different sets of categories, and `isequaln` compares the category names of each pair of elements.
- Datetime arrays -- `isequal` ignores time zone and display format when it compares points in time.
- Objects of different classes -- `isequaln` returns logical `0` (`false`). This applies even when the objects have the same properties and their values match.

`A1,A2,...,An` — Series of inputs to be compared

numeric arrays | categorical arrays | cell arrays | tables | structures | ...

Series of inputs to be compared, specified as numeric arrays, categorical arrays, cell arrays, tables, or structures. Also, you can specify the inputs as logical arrays, character arrays, datetime arrays, duration arrays, calendarDuration arrays, or objects. The numeric types of the inputs do not have to match.

You can compare categorical arrays to cell arrays of strings of the same size, or single categorical elements to single strings.

You can compare a datetime array to a cell array of date strings or a single date string.

If the inputs are *all*

- Structures -- Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays -- Must have the same sets of categories, including their order.
- Categorical arrays that are not ordinal -- Can have different sets of categories, and `isequaln` compares the category names of each pair of elements.
- Datetime arrays -- `isequaln` ignores time zone and display format when it compares points in time.
- Objects of different classes -- `isequaln` returns logical `0` (`false`). This applies even when the objects have the same properties and their values match.

More About

Tips

- When comparing handle objects, use == to test whether objects have the same handle. Use `isequaln` to treat NaN values as equal and determine if objects with different handles have equal property values.
- Use `isequal` if you want to test for equality and treat `NaN` or `NaT` values as unequal.

See Also

`eq` | `is*` | `isa*` | `isequal` | `relational operators` | `strcmp`

isequalwithequalnans

Test arrays for equality, treating NaNs as equal

Note: `isequalwithequalnans` is not recommended. Use `isequaln` instead.

Syntax

```
tf = isequalwithequalnans(A, B, ...)
```

Description

`tf = isequalwithequalnans(A, B, ...)` returns logical `1` (`true`) if the input arrays are the same type and size and hold the same contents, and logical `0` (`false`) otherwise. `NaN` (Not a Number) values are considered to be equal to each other. Numeric data types and structure field order do not have to match.

Examples

Arrays containing `NaN`s are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider `NaN`s to be equal, while `isequalwithequalnans` does.

```
A = [32 8 -29 NaN 0 5.7];
B = A;
isequal(A, B)
ans =
     0

isequalwithequalnans(A, B)
ans =
     1
```

The position of `NaN` elements in the array does matter. If they are not in the same position in the arrays being compared, then `isequalwithequalnans` returns zero.

```
A = [2 4 6 NaN 8];   B = [2 4 NaN 6 8];
isequalwithequalnans(A, B)
ans =
     0
```

More About

Tips

`isequalwithequalnans` is the same as `isequal`, except `isequalwithequalnans` considers `NaN` (Not a Number) values to be equal, and `isequal` does not.

`isequalwithequalnans` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequalwithequalnans` returns logical `1`.

See Also

`isequal` | `strcmp` | `isa` | `is*` | `relational operators` | `isequaln`

Introduced before R2006a

isevent

Determine whether input is COM object event

Syntax

```
tf = isevent(h,'eventname')
```

Description

`tf = isevent(h,'eventname')` returns logical `1` (`true`) if `eventname` is an event recognized by COM object `h`. Otherwise, returns logical `0` (`false`). The `eventname` argument is not case-sensitive.

COM functions are available on Microsoft Windows systems only.

Examples

This example tests events in a MATLAB sample control object.

1. Create an instance of the `mwsamp` control and test `DblClick`.

   ```
   f = figure('position',[100 200 200 200]);
   h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f);
   isevent(h,'DblClick')
   ```

   MATLAB displays `true`, indicating `DblClick` is an event.

2. Try the same test on`Redraw`.

   ```
   isevent(h,'Redraw')
   ```

   MATLAB displays `false`, indicating `Redraw` is not an event; it is a method.

Test events in a Microsoft Excel workbook object.

1. Create a `Workbook` object `wb`

   ```
   myApp = actxserver('Excel.Application');
   ```

```
wbs = myApp.Workbooks;
wb = Add(wbs);
```

2 Test `Activate`.

```
isevent(wb,'Activate')
```

MATLAB displays `true`, indicating `Activate` is an event.

3 Test `Save`

```
isevent(wb,'Save')
```

MATLAB displays `false`, indicating `Save` is not an event; it is a method.

More About

- "Exploring Events"
- "Functions for Working with Events"

See Also

`events (COM)` | `eventlisteners` | `registerevent`

Introduced before R2006a

isfield

Determine whether input is structure array field

Syntax

```
tf = isfield(S, 'fieldname')
tf = isfield(S, C)
```

Description

`tf = isfield(S, 'fieldname')` examines structure `S` to see if it includes the field specified by the quoted string `'fieldname'`. Output `tf` is set to logical `1` (`true`) if `S` contains the field, or logical `0` (`false`) if not. If `S` is not a structure array, `isfield` returns `false`.

`tf = isfield(S, C)` examines structure `S` for multiple fieldnames as specified in cell array of strings `C`, and returns an array of logical values to indicate which of these fields are part of the structure. Elements of output array `tf` are set to a logical 1 (`true`) if the corresponding element of `C` holds a fieldname that belongs to structure `S`. Otherwise, logical 0 (`false`) is returned in that element. In other words, if structure `S` contains the field specified in `C{m,n}`, `isfield` returns a logical 1 (`true`) in `tf(m,n)`.

Note `isfield` returns `false` if the `field` or `fieldnames` input is empty.

Examples

Example 1 — Single Fieldname Syntax

Given the following MATLAB structure,

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

`isfield` identifies `billing` as a field of that structure.

```
isfield(patient,'billing')
ans =
     1
```

Example 2 — Multiple Fieldname Syntax

Check structure `S` for any of four possible fieldnames. Only the first is found, so the first element of the return value is set to `true`:

```
S = struct('one', 1, 'two', 2);

fields = isfield(S, {'two', 'pi', 'One', 3.14})
fields =
     1     0     0     0
```

More About

- dynamic field names

See Also

`fieldnames` | `setfield` | `getfield` | `orderfields` | `rmfield` | `struct` | `isstruct` | `iscell` | `isa` | `is*`

Introduced before R2006a

isfinite

Array elements that are finite

Syntax

```
TF = isfinite(A)
```

Description

`TF = isfinite(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of the array `A` are finite and logical 0 (`false`) where they are infinite or `NaN`. For a complex number `z`, `isfinite(z)` returns `1` if both the real and imaginary parts of `z` are finite, and `0` if either the real or the imaginary part is infinite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2];
isfinite(1./a)
ans =
     1     1     0     1     1
isfinite(0./a)
ans =
     1     1     0     1     1
```

See Also

`isinf` | `isnan` | `is*`

Introduced before R2006a

isfloat

Determine if input is floating-point array

Syntax

```
tf = isfloat(A)
```

Description

`tf = isfloat(A)` returns `true` if `A` is a floating-point array and `false` otherwise. The floating-point types are `single` and `double`, and subclasses of `single` and `double`.

Examples

These examples show the values `isfloat` returns when passed specific types:

```
% pi returns a floating-point value
isfloat(pi)
ans =
     1
% Complex numbers are floating-point values
isfloat(3+7i)
ans =
     1
% Single-precision numbers are floating-point values
isfloat(realmax('single'))
ans =
     1
% isfloat returns a logical value
isfloat(isfloat(pi))
ans =
     0
```

See Also

`isa` | `isinteger` | `double` | `single` | `isnumeric`

Introduced before R2006a

isglobal

(To be removed) Determine if input is global variable

Note `isglobal` has been removed. Avoid conditions in your code that check variable scope. If you need to convert your code quickly in the short term, replace `isglobal` with `~isempty(whos('global','variable'))`. However, you should refactor your code to avoid conditional checks on variable scope.

Syntax

```
tf = isglobal(A)
```

Description

`tf = isglobal(A)` returns logical `1` (`true`) if `A` has been declared to be a global variable in the context from which `isglobal` is called, and logical `0` (`false`) otherwise.

More About

Tips

`isglobal` is most commonly used in conjunction with conditional global declaration. An alternate approach is to use a pair of variables, one local and one declared global.

Instead of using

```
if condition
   global x
end

x = some_value

if isglobal(x)
```

```
   do_something
end
```

You can use

```
global gx
if condition
   gx = some_value
else
   x = some_value
end

if condition
   do_something
end
```

If no other workaround is possible, you can replace the command

```
isglobal(variable)
```

with

```
~isempty(whos('global','variable'))
```

See Also

`global` | `isvarname` | `isa` | `is*`

Introduced before R2006a

isgraphics

True for valid graphics object handles

Syntax

```
tf = isgraphics(H)
tf = isgraphics(H,type)
```

Description

`tf = isgraphics(H)` returns `true` for elements of `H` that are valid graphics object and `false` where elements are not valid graphics objects.

`tf = isgraphics(H,type)` returns `true` for elements of `H` that are valid graphics objects of the type specified by the `type` argument. An object type is the string contained in the object's `Type` property.

Examples

Test for Valid Handles

Create a plot and return the handle array. Test array for valid handles

```
H = plot(rand(5));
isgraphics(H)
```

```
ans =

     1
     1
     1
     1
     1
```

Test for Handle Types

Create a plot and return the handle array. Concatenate with other graphics objects and test for handles that are of type line.

```
H = plot(rand(5));
a = [H;gca;gcf];
isgraphics(a,'line')
```

```
ans =

     1
     1
     1
     1
     1
     0
     0
```

Input Arguments

`H` — Input variable or expression

graphics object array

Input variable or expression that evaluates to graphics object handles.

`type` — The object type

char

The object type, specified as a quoted string. An object's type is contained in its `Type` property.

Output Arguments

`tf` — true or false

logical array

Logical array of `1`s and `0`s indicating the elements of the input array that are valid object handles or valid handles of a specific type if you specify a type argument.

More About

- “Graphics Object Handles”

See Also

Functions

`isa` | `ishghandle`

ishandle

Test for valid graphics or Java object handle

Syntax

```
ishandle(H)
```

Description

`ishandle(H)` returns an array whose elements are `1` where the elements of `H` are graphics or Java object handles, and `0` where they are not.

Note: Use the most specific function for your application instead of `ishandle`, as described in the following sections.

MATLAB Handle Objects

Use the `isa` function to determine the class of MATLAB objects.

Use the `handle` class `isvalid` method to determine the validity of handle objects. See "Testing Handle Validity" for information on testing for MATLAB handle objects.

Graphics Object Handles

Use `isgraphics` for graphics objects.

Java Object Handles

Use `isjava` for Java objects.

See Also

`isa` | `isjava` | `isgraphics`

Introduced before R2006a

ishermitian

Determine if matrix is Hermitian or skew-Hermitian

Syntax

```
tf = ishermitian(A)
tf = ishermitian(A,skewOption)
```

Description

`tf = ishermitian(A)` returns logical `1` (`true`) if square matrix `A` is Hermitian; otherwise, it returns logical `0` (`false`).

`tf = ishermitian(A,skewOption)` specifies the type of the test. Specify `skewOption` as `'skew'` to determine if `A` is skew-Hermitian.

Examples

Test if Symmetric Matrix Is Hermitian

Create a 3-by-3 matrix.

```
A = [1 0 1i; 0 1 0; 1i 0 1]
```

```
A =

   1.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 1.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 1.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i
```

The matrix is symmetric with respect to its real-valued diagonal.

Test whether the matrix is Hermitian.

```
tf = ishermitian(A)
```

```
tf =

     0
```

The result is logical `0` (`false`) because `A` is not Hermitian. In this case, `A` is equal to its transpose, `A.'`, but not its complex conjugate transpose, `A'`.

Change the element in `A(3,1)` to be `-1i`.

```
A(3,1) = -1i;
```

Determine if the modified matrix is Hermitian.

```
tf = ishermitian(A)
```

```
tf =

     1
```

The matrix, `A`, is now Hermitian because it is equal to its complex conjugate transpose, `A'`.

Test if Matrix Is Skew-Hermitian

Create a 3-by-3 matrix.

```
A = [-1i -1 1-i;1 -1i -1;-1-i 1 -1i]
```

```
A =

   0.0000 - 1.0000i  -1.0000 + 0.0000i   1.0000 - 1.0000i
   1.0000 + 0.0000i   0.0000 - 1.0000i  -1.0000 + 0.0000i
  -1.0000 - 1.0000i   1.0000 + 0.0000i   0.0000 - 1.0000i
```

The matrix has pure imaginary numbers on the main diagonal.

Specify `skewOption` as `'skew'` to determine whether the matrix is skew-Hermitian.

```
tf = ishermitian(A,'skew')
```

```
tf =

     1
```

The matrix, `A`, is skew-Hermitian since it is equal to the negation of its complex conjugate transpose, `-A'`.

Input Arguments

`A` — Input matrix

numeric matrix

Input matrix, specified as a numeric matrix. If `A` is not square, then `ishermitian` returns logical `0` (`false`).

Data Types: `single` | `double`
Complex Number Support: Yes

`skewOption` — Test type

`'nonskew'` (default) | `'skew'`

Test type, specified as `'nonskew'` (default) or `'skew'`. Specify `'skew'` to test whether `A` is skew-Hermitian. Specifying `ishermitian(A,'nonskew')` is equivalent to `ishermitian(A)`.

Data Types: `char`

More About

Hermitian Matrix

- A square matrix, `A`, is Hermitian if it is equal to its complex conjugate transpose, `A = A'`.

  In terms of the matrix elements, this means that

  $$a_{i,j} = \overline{a}_{j,i} .$$

- The entries on the diagonal of a Hermitian matrix are always real. Since real matrices are unaffected by complex conjugation, a real matrix that is symmetric is also Hermitian. For example, the matrix

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

is both symmetric and Hermitian.

- The eigenvalues of a Hermitian matrix are real.

Skew-Hermitian Matrix

- A square matrix, `A`, is skew-Hermitian if it is equal to the negation of its complex conjugate transpose, `A = -A'`.

  In terms of the matrix elements, this means that

  $a_{i,j} = -\overline{a}_{j,i}$.

- The entries on the diagonal of a skew-Hermitian matrix are always pure imaginary or zero. Since real matrices are unaffected by complex conjugation, a real matrix that is skew-symmetric is also skew-Hermitian. For example, the matrix

  $$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

  is both skew-Hermitian and skew-symmetric.
- The eigenvalues of a skew-Hermitian matrix are purely imaginary or zero.

See Also

`ctranspose` | `eig` | `isreal` | `issymmetric` | `transpose`

Introduced in R2014a

ishghandle

True for graphics object handles

Syntax

```
ishghandle(h)
```

Description

`ishghandle(h)` returns an array that contains 1's where the elements of `h` are handles to existing graphic objects and 0's where they are not. Differs from `ishandle` in that Simulink objects handles return false.

Examples

Create a plot and find the valid handles:

```
x = [1:10];
y = [1:10];
p = plot(x,y);
ishghandle(p)
```

```
ans =

     1
```

More About

- “Graphics Object Handles”

See Also

`isa` | `isgraphics` | `gca`

ishold

Current hold state

Syntax

```
tf = ishold
```

Description

`tf = ishold` returns `1` if `hold` is `on`, and `0` if it is `off`. When `hold` is `on`, the current plot and most axis properties are held so that subsequent graphing commands add to the existing graph.

A state of `hold on` implies that both figure and axes `NextPlot` properties are set to `add`.

More About

- "Control Graph Display"

See Also

`hold` | `newplot`

Introduced before R2006a

isinf

Array elements that are infinite

Syntax

```
TF = isinf(A)
```

Description

`TF = isinf(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `+Inf` or `-Inf` and logical 0 (`false`) where they are not. For a complex number `z`, `isinf(z)` returns `1` if either the real or imaginary part of `z` is infinite, and `0` if both the real and imaginary parts are finite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2];

isinf(1./a)

ans =
     0     0     1     0     0

isinf(0./a)

ans =
     0     0     0     0     0
```

See Also

`isfinite` | `isnan` | `is*`

Introduced before R2006a

isinteger

Determine if input is integer array

Syntax

```
tf = isinteger(A)
```

Description

`tf = isinteger(A)` returns `true` if the array `A` is an integer type and `false` otherwise.

An integer array is any of the following integer types and any subclasses of those types:

MATLAB Integer Types

`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array

Examples

These examples show the values `isinteger` returns when passed specific types:

```
% uint8 is one of the integer types
isinteger(uint8(1:255))
ans =
     1
```

```
% pi returns a double-precision value
isinteger(pi)
ans =
     0
% Constants are double-precision by default
isinteger(3)
ans =
     0
% isinteger returns a logical value
isinteger(isinteger(uint8(3)))
ans =
     0
```

See Also

isa | isnumeric | isfloat

Introduced before R2006a

isinterface

Determine whether input is COM interface

Syntax

```
tf = isinterface(h)
```

Description

`tf = isinterface(h)` returns logical `1` (`true`) if handle `h` is a COM interface. Otherwise, returns logical `0` (`false`).

COM functions are available on Microsoft Windows systems only.

Examples

Test an instance of a Microsoft Excel application.

```
h = actxserver('Excel.Application');
isinterface(h)
```

MATLAB displays `false`, indicating object `h` is not an interface.

Test a `workbooks` object

```
w = get(h,'workbooks');
isinterface(w)
```

MATLAB displays `true`, indicating object `w` is an interface.

More About

- “Exploring Interfaces”

See Also

`iscom` | `interfaces`

isjava

Determine if input is Java object

Syntax

```
tf = isjava(A)
```

Description

`tf = isjava(A)` returns logical `1` (`true`) if object A is a Java object. Otherwise, it returns logical `0` (`false`).

Examples

Test if java.util.Date Is Java Object

Create an instance of the Java Date class.

```
myDate = java.util.Date;
isjava(myDate)
```

```
ans =

     1
```

`myDate` is a Java object.

However, `myDate` is not a MATLAB object.

```
isobject(myDate)
```

```
ans =

     0
```

See Also

`is*` | `isa` | `isobject` | `javaArray` | `javaMethod` | `javaObject`

Introduced before R2006a

isKey

Class: containers.Map
Package: containers

Determine if `containers.Map` object contains key

Syntax

```
tf = isKey(mapObj,keySet)
```

Description

`tf = isKey(mapObj,keySet)` looks for the specified keys in `mapObj`, and returns logical `true` (`1`) for the keys that it finds, and logical `false` (`0`) for those it does not. `keySet` is a scalar key or a cell array of keys.

Input Arguments

`mapObj`

Object of class `containers.Map`.

`keySet`

Scalar value, string, or cell array that specifies keys to find in `mapObj`.

Output Arguments

`tf`

Array of logical values. If `keySet` is a scalar or a string, `tf` is a scalar. Otherwise, `tf` has the same size and dimensions as `keySet`.

Examples

Find Keys in a Map

Construct a map that contains rainfall data for several months:

```
months =   {'Jan', 'Feb', 'Mar', 'Apr'};
rainfall = [327.2, 368.2, 197.6, 178.4];
mapObj = containers.Map(months,rainfall);
```

Determine if keys `Apr`, `May`, and `Jun` are in the map:

```
keySet = {'Apr','May','Jun'};
tf = isKey(mapObj,keySet)
```

This code returns 1-by-3 vector `tf`:

```
tf =
     1     0     0
```

Find a Single Key

Determine if `mapObj` from the previous example contains key `Feb`:

```
keySet = 'Feb';
tf = isKey(mapObj,keySet)
```

This code returns scalar `tf`:

```
tf =
     1
```

See Also

`keys` | `values` | `containers.Map` | `remove`

iskeyword

Determine whether input is MATLAB keyword

Syntax

```
tf = iskeyword('str')
iskeyword str
iskeyword
```

Description

`tf = iskeyword('str')` returns logical `1` (`true`) if the string `str` is a keyword in the MATLAB language and logical `0` (`false`) otherwise. MATLAB keywords cannot be used as variable names.

`iskeyword str` uses the MATLAB command format.

`iskeyword` returns a list of all MATLAB keywords.

Examples

To test if the word `while` is a MATLAB keyword,

```
iskeyword while
ans =
    1
```

To obtain a list of all MATLAB keywords,

```
iskeyword
    'break'
    'case'
    'catch'
    'classdef'
    'continue'
    'else'
    'elseif'
```

```
'end'
'for'
'function'
'global'
'if'
'otherwise'
'parfor'
'persistent'
'return'
'spmd'
'switch'
'try'
'while'
```

See Also

`isvarname` | `matlab.lang.makeValidName` | `matlab.lang.makeUniqueStrings` | `is*`

Introduced before R2006a

isletter

Array elements that are alphabetic letters

Syntax

```
tf = isletter('str')
```

Description

`tf = isletter('str')` returns an array the same size as `str` containing logical `1` (`true`) where the elements of `str` are letters of the alphabet and logical `0` (`false`) where they are not.

Examples

Find the letters in character array `s`.

```
s = 'A1,B2,C3';

isletter(s)
ans =
     1     0     0     1     0     0     1     0
```

See Also

`ischar` | `isspace` | `isstrprop` | `iscellstr` | `isnumeric` | `char` | `strings` | `isa` | `is*`

Introduced before R2006a

islogical

Determine if input is logical array

Syntax

```
tf = islogical(A)
```

Description

`tf = islogical(A)` returns `true` if `A` is a logical array and `false` otherwise. `islogical` also returns `true` if `A` is an instance of a class that is derived from the `logical` class.

Examples

These examples show the values `islogical` returns when passed specific types:

```
% Relational operators return logical values
islogical(5<7)
ans =
     1

% true and false return logical values
islogical(true) & islogical(false)
ans =
     1

% Constants are double-precision by default
islogical(1)
ans =
     0

% logical creates logical values
islogical(logical(1))
ans =
     1
```

More About

- "Determine if Arrays Are Logical"

See Also

`is*` | `isa` | `logical`

Introduced before R2006a

ismac

Determine if version is for Mac OS X platform

Syntax

```
tf = ismac
```

Description

`tf = ismac` returns logical `1` (`true`) if the version of MATLAB software is for the Apple Mac OS X platform. Otherwise, it returns logical `0` (`false`).

More About

Tips

- The `isunix` function also determines if version is for Mac OS X platforms.

See Also

`computer` | `is*` | `ispc` | `isstudent` | `isunix`

ismatrix

Determine whether input is matrix

Syntax

```
ismatrix(V)
```

Description

`ismatrix(V)` returns logical 1 (`true`) if `size`(V) returns `[m n]` with nonnegative integer values `m` and `n`, and logical 0 (`false`) otherwise.

Examples

Create three vectors:

```
V1 = rand(5,1);
V2 = rand(5,1);
V3 = rand(5,1);
```

Concatenate the vectors and check that the result is a matrix. `ismatrix` returns 1:

```
M = cat(2,V1,V2,V3);
ismatrix(M)
ans =
     1
```

See Also

`iscolumn` | `isrow` | `isscalar` | `isvector`

ismember

Array elements that are members of set array

Syntax

```
Lia = ismember(A,B)
Lia = ismember(A,B,'rows')
[Lia,Locb] = ismember(A,B)
[Lia,Locb] = ismember(A,B,'rows')

[Lia,Locb] = ismember( ___ ,'legacy')
```

Description

`Lia = ismember(A,B)` returns an array containing `1` (`true`) where the data in `A` is found in `B`. Elsewhere, it returns `0` (`false`).

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `ismember` returns a logical value for each element of `A`. The output, `Lia`, is an array of the same size as `A`.
- If `A` and `B` are tables, then `ismember` returns a logical value for each row of `A`. The output, `Lia`, is a column vector with the same number of rows as `A`.

`Lia = ismember(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns a column vector containing `1` (`true`) where the rows of `A` are also rows of `B`. Elsewhere, it returns `0` (`false`).

`A` and `B` must have the same number of columns when you use the `'rows'` option. Furthermore, the `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[Lia,Locb] = ismember(A,B)` also returns an array, `Locb`.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `Locb` contains the lowest index in `B` for each value in `A` that is a member of `B`. The output array, `Locb`, contains `0` wherever `A` is not a member of `B`.

- If `A` and `B` are tables, then `Locb` contains the lowest index in `B` for each row in `A` that is also a row in `B`. The output vector, `Locb`, contains `0` whenever `A` is not a row of `B`.

`[Lia,Locb] = ismember(A,B,'rows')` also returns a column vector, `Locb`, containing the lowest index in `B` for each row in `A` that is also a row in `B`. The output vector, `Locb`, contains `0` wherever `A` is not a row of `B`.

`[Lia,Locb] = ismember( ___ ,'legacy')` preserves the behavior of the `ismember` function from R2012b and prior releases using any of the input arguments in previous syntaxes.

The `'legacy'` option does not support categorical arrays, tables, datetime arrays, or duration arrays.

Examples

Values That Are Members of Set

Create two vectors with values in common.

```
A = [5 3 4 2];
B = [2 4 4 4 6 8];
```

Determine which elements of `A` are also in `B`.

```
Lia = ismember(A,B)
```

```
Lia =

     0     0     1     1
```

`A(3)` and `A(4)` are found in `B`.

Table Rows Found in Another Table

Create two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
A =

    Var1    Var2    Var3
    ____    ____    _____

    1       A       false
    2       B       true
    3       C       false
    4       D       true
    5       E       false


B =

    Var1    Var2    Var3
    ____    ____    _____

    1       A       false
    3       C       false
    5       E       false
    7       G       false
    9       I       false
```

Determine which rows of `A` are also in `B`.

```
Lia = ismember(A,B)
```

```
Lia =

     1
     0
     1
     0
     1
```

`A(1,:)`, `A(3,:)`, and `A(5,:)` are found in `B`.

Members of Set and Indices to Values

Create two vectors with values in common.

```
A = [5 3 4 2];
B = [2 4 4 4 6 8];
```

Determine which elements of A are also in B as well as their corresponding locations in B.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia =

     0     0     1     1


Locb =

     0     0     2     1
```

The lowest index to `A(3)` is `B(2)`, and `A(4)` is found in `B(1)`.

Rows of Another Table and Their Location

Create a table, A, of gender, age, and height for five people.

```
A = table(['M';'M';'F';'M';'F'],[27;52;31;46;35],[74;68;64;61;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty' 'Bob' 'Judy'})
```

```
A =

              Gender    Age    Height
              ______    ___    ______

    Ted       M         27     74
    Fred      M         52     68
    Betty     F         31     64
    Bob       M         46     61
    Judy      F         35     64
```

Create another table, B, with rows in common with A.

```
B = table(['M';'F';'F';'F'],[47;31;35;23],[68;64;62;58],...
```

```
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Joe' 'Meg' 'Beth' 'Amy'})


B =

            Gender    Age    Height
            ______    ___    ______

    Joe     M         47     68
    Meg     F         31     64
    Beth    F         35     62
    Amy     F         23     58
```

Determine which rows of `A` are also in `B`, as well as their corresponding locations in `B`.

```
[Lia,Locb] = ismember(A,B)


Lia =

     0
     0
     1
     0
     0


Locb =

     0
     0
     2
     0
     0
```

Two rows that have the same values, but different names, are considered equal. The same data for `Betty` is found in `B(2,:)`, which corresponds to `Meg`.

Rows That Belong to a Set

Create two matrices with a row in common.

```
A = [1 3 5 6; 2 4 6 8];
B = [2 4 6 8; 1 3 5 7; 2 4 6 8];
```

Determine which rows of A are also in B as well as their corresponding locations in B.

```
[Lia, Locb] = ismember(A,B, 'rows')
```

```
Lia =

     0
     1


Locb =

     0
     1
```

The lowest index to `A(2,:)` is `B(1,:)`.

Members of Set Containing NaN Values

Create two vectors containing NaN.

```
A = [5 NaN NaN];
B = [5 NaN NaN];
```

Determine which elements of A are also in B, as well as their corresponding locations in B.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia =

     1     0     0


Locb =

     1     0     0
```

`ismember` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Determine which strings of `A` are also in `B`.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia =

     0     1     0     1


Locb =

     0     2     0     4
```

`ismember` treats trailing white space in cell arrays of strings as distinct characters.

Members of Char and Cell Array of Strings

Create a character array, `A`, and a cell array of strings, `B`.

```
A = ['cat';'dog';'fox';'pig'];
B = {'dog','cat','fish','horse'};
```

Determine which strings of `A` are also in `B`.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia =

     1
     1
     0
```

```
     0


Locb =

     2
     1
     0
     0
```

Preserve Legacy Behavior of ismember

Use the `'legacy'` flag to preserve the behavior of `ismember` from R2012b and prior releases in your code.

Find the members of `B` with the current behavior.

```
A = [5 3 4 2];
B = [2 4 4 4 6 8];
[Lia1,Locb1] = ismember(A,B)
```

```
Lia1 =

     0     0     1     1


Locb1 =

     0     0     2     1
```

Find the members of `B`, and preserve the legacy behavior.

```
[Lia2,Locb2] = ismember(A,B,'legacy')
```

```
Lia2 =

     0     0     1     1


Locb2 =
```

```
     0     0     4     1
```

Input Arguments

A — Query array

numeric array | logical array | character array | categorical array | datetime arrays | duration arrays | cell array of strings | table

Query array, specified as a numeric array, logical array, character array, categorical array, datetime array, duration array, cell array of strings, or a table.

`A` must belong to the same class as `B` with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.
- Categorical arrays can combine with cell arrays of strings or single strings.
- Datetime arrays can combine with cell arrays of date strings or single date strings.

If `A` and `B` are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither `A` nor `B` are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names.

If you specify the `'rows'` option, `A` must have the same number of columns as `B`.

If `A` is a table, it must have the same variable names as `B`. Conversely, the row names do not matter. Two rows that have the same values, but different names, are considered equal.

If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

Furthermore, `A` can be an object with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `eq`
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class.

B — Set array

numeric array | logical array | character array | categorical array | datetime arrays | duration arrays | cell array of strings | table

Set array, specified as a numeric array, logical array, character array, categorical array, datetime array, duration array, cell array of strings, or a table.

`B` must belong to the same class as `A` with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.
- Categorical arrays can combine with cell arrays of strings or single strings.
- Datetime arrays can combine with cell arrays of date strings or single date strings.

If `A` and `B` are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither `A` nor `B` are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names.

If you specify the `'rows'` option, `B` must have the same number of columns as `A`.

If `B` is a table, it must have the same variable names as `A`. Conversely, the row names do not matter. Two rows that have the same values, but different names, are considered equal.

If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

Furthermore, `B` can be an object with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `eq`
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class.

Output Arguments

Lia — Logical index to A

vector | matrix | N-D array

Logical index to `A`, returned as a vector, matrix or N-D array containing `1` (`true`) wherever the values (or rows) in `A` are members of `B`. Elsewhere, it returns `0` (`false`).

`Lia` is an array of the same size as `A`, unless you specify the `'rows'` flag.

If the `'rows'` flag is specified or if `A` is a table, `Lia` is a column vector with the same number of rows as `A`.

`Locb` — Locations in `B`

vector | matrix | N-D array

Locations in `B`, returned as a vector, matrix, or N-D array. If the `'legacy'` flag is not specified, `Locb` contains the lowest indices to the values (or rows) in `B` that are found in `A`. `Locb` contains `0` wherever `A` is not a member of `B`.

`Locb` is an array of the same size as `A` unless you specify the `'rows'` flag.

If the `'rows'` flag is specified or if `A` is a table, `Locb` is a column vector with the same number of rows as `A`.

More About

Tips

- To find the rows from table `A` that are found in `B` with respect to a subset of variables, you can use column subscripting. For example, you can use `ismember(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.

See Also

`intersect` | `ismembertol` | `issorted` | `setdiff` | `setxor` | `sort` | `union` | `unique`

Introduced before R2006a

ismembertol

Set ismember within tolerance

`ismembertol` is similar to `ismember`. Whereas `ismember` performs exact comparisons, `ismembertol` performs comparisons using a tolerance.

Syntax

```
LIA = ismembertol(A,B,tol)
LIA = ismembertol(A,B)
[LIA,LocB] = ismembertol( ___ )
[ ___ ] = ismembertol( ___ ,Name,Value)
```

Description

`LIA = ismembertol(A,B,tol)` returns an array containing logical `1` (`true`) where the elements of `A` are within tolerance of the elements in `B`. Otherwise, the array contains logical `0` (`false`). Two values, `u` and `v`, are within tolerance if

```
abs(u-v) <= tol*max(abs([A(:);B(:)]))
```

That is, `ismembertol` scales the `tol` input based on the magnitude of the data.

`LIA = ismembertol(A,B)` uses a default tolerance of `1e-6` for single-precision inputs and `1e-12` for double-precision inputs.

`[LIA,LocB] = ismembertol( ___ )` also returns an array, `LocB`, that contains the index location in `B` for each element in `A` that is a member of `B`. You can use any of the input arguments in previous syntaxes.

`[ ___ ] = ismembertol( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments using any of the input or output argument combinations in previous syntaxes. For example, `ismembertol(A,B,'ByRows',true)` compares the rows of `A` and `B` and returns a logical column vector.

Examples

Set Members in Presence of Numerical Error

Create a vector, `x`. Obtain a second vector, `y`, by transforming and untransforming `x`. This transformation introduces round-off differences to `y`.

```
x = (1:6)'*pi;
y = 10.^log10(x);
```

Verify that `x` and `y` are not identical by taking the difference.

```
x-y
```

```
ans =

   1.0e-14 *

    0.0444
         0
         0
         0
         0
   -0.3553
```

Use `ismember` to find the elements of `x` that are in `y`. The `ismember` function performs exact comparisons and determines that some of the matrix elements in `x` are not members of `y`.

```
lia = ismember(x,y)
```

```
lia =

     0
     1
     1
     1
     1
     0
```

Use `ismembertol` to perform the comparison using a small tolerance. `ismembertol` treats elements that are within tolerance as equal and determines that all of the elements in `x` are members of `y`.

```
LIA = ismembertol(x,y)
```

```
LIA =

     1
     1
     1
     1
     1
     1
```

Determine Set Members by Rows

By default, `ismembertol` looks for *elements* that are within tolerance, but it also can find *rows* of a matrix that are within tolerance.

Create a numeric matrix, `A`. Obtain a second matrix, `B`, by transforming and untransforming `A`. This transformation introduces round-off differences to `B`.

```
A = [0.05 0.11 0.18; 0.18 0.21 0.29; 0.34 0.36 0.41; ...
    0.46 0.52 0.76; 0.82 0.91 1.00];
B = log10(10.^A);
```

Use `ismember` to find the rows of `A` that are in `B`. `ismember` performs exact comparisons and thus determines that most of the rows in `A` are not members of `B`, even though some of the rows differ by only a small amount.

```
lia = ismember(A,B,'rows')
```

```
lia =

     0
     0
     0
     0
     1
```

Use `ismembertol` to perform the row comparison using a small tolerance. `ismembertol` treats rows that are within tolerance as equal and thus determines that all of the rows in `A` are members of `B`.

```
LIA = ismembertol(A,B,'ByRows',true)
```

```
LIA =

     1
     1
     1
     1
     1
```

Average Similar Values in Vectors

Create two vectors of random numbers and determine which values in `A` are also members of `B`, using a tolerance. Specify `OutputAllIndices` as `true` to return all of the indices for the elements in `B` that are within tolerance of the corresponding elements in `A`.

```
rng(5)
A = rand(1,15);
B = rand(1,5);
[LIA,LocAllB] = ismembertol(A,B,0.2,'OutputAllIndices',true)
```

```
LIA =

  Columns 1 through 13

     1     0     1     0     1     1     1     1     1     1     0     1     1

  Columns 14 through 15

     1     0


LocAllB =

  Columns 1 through 6
```

```
    [2x1 double]    [0]    [2x1 double]    [0]    [3x1 double]    [2x1 double]

  Columns 7 through 12

    [4]    [3x1 double]    [3x1 double]    [2x1 double]    [0]    [2x1 double]

  Columns 13 through 15

    [4x1 double]    [2x1 double]    [0]
```

Find the average value of the elements in `B` that are within tolerance of the value `A(13)`. The cell `LocAllB{13}` contains all the indices for elements in `B` that are within tolerance of `A(13)`.

```
A(13)
allB = B(LocAllB{13})
aveB = mean(allB)


ans =

    0.4413


allB =

    0.2741    0.4142    0.2961    0.5798


aveB =

    0.3911
```

Specify Absolute Tolerance

By default, `ismembertol` uses a tolerance test of the form `abs(u-v) <= tol*DS`, where `DS` automatically scales based on the magnitude of the input data. You can specify a different `DS` value to use with the `DataScale` option. However, absolute tolerances (where `DS` is a scalar) do not scale based on the magnitude of the input data.

First, compare two small values that are a distance `eps` apart. Specify `tol` and `DS` to make the within tolerance equation `abs(u-v) <= 10^-6`.

```
x = 0.1;
ismembertol(x, exp(log(x)), 10^-6, 'DataScale', 1)
```

```
ans =

     1
```

Next, increase the magnitude of the values. The round-off error in the calculation `exp(log(x))` is proportional to the magnitude of the values, specifically to `eps(x)`. Even though the two large values are a distance `eps` from one another, `eps(x)` is now much larger. Therefore, `10^-6` is no longer a suitable tolerance.

```
x = 10^10;
ismembertol(x, exp(log(x)), 10^-6, 'DataScale', 1)
```

```
ans =

     0
```

Correct this issue by using the default (scaled) value of `DS`.

```
Y = [0.1 10^10];
ismembertol(Y, exp(log(Y)))
```

```
ans =

     1     1
```

Specify DataScale by Column

Create a set of random 2-D points, and then use `ismembertol` to group the points into vertical bands that have a similar (within-tolerance) x-coordinate to a small set of query points, `B`. Use these options with `ismembertol`:

- Specify `ByRows` as `true`, since the point coordinates are in the rows of `A` and `B`.
- Specify `OutputAllIndices` as `true` to return the indices for all points in `A` that have an x-coordinate within tolerance of the query points in `B`.

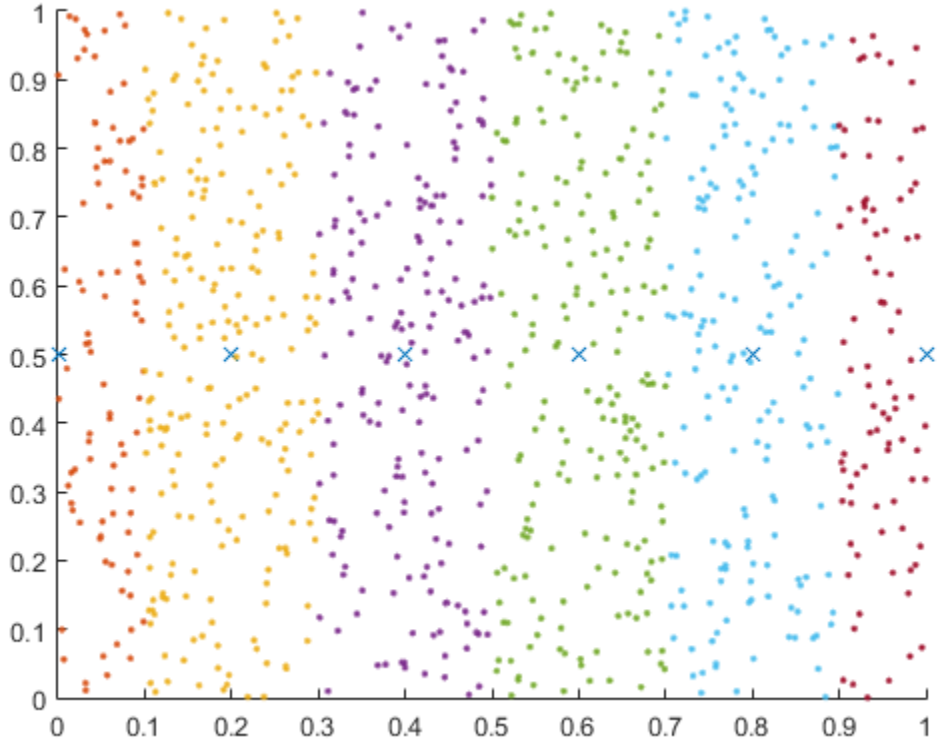
- Specify DataScale as `[1 Inf]` to use an absolute tolerance for the x-coordinate, while ignoring the y-coordinate.

```
A = rand(1000,2);
B = [(0:.2:1)',0.5*ones(6,1)];
[LIA,LocAllB] = ismembertol(B, A, 0.1, 'ByRows', true, ...
    'OutputAllIndices', true, 'DataScale', [1,Inf])
```

```
LIA =

     1
     1
     1
     1
     1
     1


LocAllB =

    [ 94x1 double]
    [223x1 double]
    [195x1 double]
    [212x1 double]
    [187x1 double]
    [ 89x1 double]
```

Plot the points in A that are within tolerance of each query point in B.

```
hold on
plot(B(:,1),B(:,2),'x')
for i=1:length(LocAllB)
    plot(A(LocAllB{i},1), A(LocAllB{i},2),'.')
end
```



- “Group Scattered Data Using a Tolerance”

Input Arguments

A — Query array

scalar | vector | matrix

Query array, specified as a scalar, vector, or matrix. Inputs `A` and `B` must be full.

If you specify the `ByRows` option, then `A` and `B` must have the same number of columns.

Data Types: `single` | `double`

`B` — Query array

scalar | vector | matrix

Query array, specified as a scalar, vector, or matrix. Inputs `A` and `B` must be full.

If you specify the `ByRows` option, then `A` and `B` must have the same number of columns.

Data Types: `single` | `double`

`tol` — Comparison tolerance

positive real scalar

Comparison tolerance, specified as a positive real scalar. `ismembertol` scales the `tol` input using the maximum absolute values in the input arrays `A` and `B`. Then `ismembertol` uses the resulting scaled comparison tolerance to determine which elements in `A` are also a member of `B`. If two elements are within tolerance of each other, then `ismembertol` considers them to be equal.

Two values, `u` and `v`, are within tolerance if `abs(u-v) <= tol*max(abs([A(:);B(:)]))`.

To specify an absolute tolerance, specify both `tol` and the `'DataScale'` Name-Value pair.

Example: `tol = 0.05`

Example: `tol = 1e-8`

Example: `tol = eps`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `LIA = ismembertol(A,B,'ByRows',true)`

`'OutputAllIndices'` — Output index type

`false` (default) | `true` | `0` | `1`

Output index type, specified as the comma-separated pair consisting of `'OutputAllIndices'` and either `false` (default), `true`, `0`, or `1`. `ismembertol` interprets numeric `0` as `false` and numeric `1` as `true`.

When `OutputAllIndices` is `true`, the `ismembertol` function returns the second output, `LocB`, as a cell array. The cell array contains the indices for *all* elements in `B` that are within tolerance of the corresponding value in `A`. That is, each cell in `LocB` corresponds to a value in `A`, and the values in each cell correspond to locations in `B`.

Example: `[LIA,LocAllB] = ismembertol(A,B,tol,'OutputAllIndices',true)`

`'ByRows'` — Row comparison toggle

`false` (default) | `true` | `0` | `1`

Row comparison toggle, specified as the comma-separated pair consisting of `'ByRows'` and either `false` (default), `true`, `0`, or `1`. `ismembertol` interprets numeric `0` as `false` and numeric `1` as `true`. Use this option to find rows in `A` and `B` that are within tolerance.

When `ByRows` is `true`:

- `ismembertol` compares the rows of `A` and `B` by considering each column separately. Thus, `A` and `B` must have the same number of columns.
- If the corresponding row in `A` is within tolerance of a row in `B`, then `LIA` contains logical `1` (`true`). Otherwise, it contains logical `0` (`false`).

Two rows, `u` and `v`, are within tolerance if `abs(u-v) <= tol*max(abs([A;B]))`.

Example: `LIA = ismembertol(A,B,tol,'ByRows',true)`

`'DataScale'` — Scale of data

scalar | vector

Scale of data, specified as the comma-separated pair consisting of `'DataScale'` and either a scalar or vector. Specify `DataScale` as a numeric scalar, `DS`, to change the tolerance test to be, `abs(u-v) <= tol*DS`.

When used together with the `ByRows` option, the `DataScale` value also can be a vector. In this case, each element of the vector specifies `DS` for a corresponding column in `A`. If a value in the `DataScale` vector is `Inf`, then `ismembertol` ignores the corresponding column in `A`.

Example: `LIA = ismembertol(A,B,'DataScale',1)`

Example: `[LIA,LocB] = ismembertol(A,B,'ByRows',true,'DataScale', [eps(1) eps(10) eps(100)])`

Data Types: `single` | `double`

Output Arguments

`LIA` — Logical index to `A`

vector | matrix

Logical index to `A`, returned as a vector or matrix containing logical `1` (`true`) wherever the elements (or rows) in `A` are members of `B` (within tolerance). Elsewhere, `LIA` contains logical `0` (`false`).

`LIA` is the same size as `A`, unless you specify the `ByRows` option. In that case, `LIA` is a column vector with the same number of rows as `A`.

`LocB` — Locations in `B`

vector | matrix | cell array

Locations in `B`, returned as a vector, matrix, or cell array. `LocB` contains the indices to the elements (or rows) in `B` that are found in `A` (within tolerance). `LocB` contains `0` wherever an element in `A` is not a member of `B`.

If `OutputAllIndices` is `true`, then `ismembertol` returns `LocB` as a cell array. The cell array contains the indices for *all* elements in `B` that are within tolerance of the corresponding value in `A`. That is, each cell in `LocB` corresponds to a value in `A`, and the values in each cell correspond to locations in `B`.

`LocB` is the same size as `A`, unless you specify the `ByRows` option. In that case, `LocB` is a column vector with the same number of rows as `A`.

See Also

`eps` | `ismember` | `unique` | `uniquetol`

Introduced in R2015a

ismethod

Determine if method of object

Syntax

```
tf = ismethod(obj,'methodName')
```

Description

`tf = ismethod(obj,'methodName')` returns logical `1` (`true`) if the specified `methodName` is a public method of object `obj`. Otherwise, returns logical `0` (`false`).

Examples

Determine if objects support equality testing:

```
if ismethod(obj1,'eq') && ismethod(obj2,'eq')
   tf = obj1 == obj2;
end
```

See Also

`methods` | `class` | `isprop` | `isobject`

Introduced before R2006a

ismethod (COM)

Determine whether input is COM object method

Syntax

```
tf = ismethod(h,'methodname')
```

Description

`tf = ismethod(h,'methodname')` returns logical `1` (`true`) if the specified `methodname` is a method you can call on COM object `h`. Otherwise, returns logical `0` (`false`).

COM functions are available on Microsoft Windows systems only.

Examples

Test members of an instance of a Microsoft Excel application.

```
h = actxserver('Excel.Application');
ismethod(h,'SaveWorkspace')
```

MATLAB returns `true`, `SaveWorkspace` is a method.

Try the same test on `UsableWidth`.

```
ismethod(h,'UsableWidth')
```

MATLAB returns `false`, `UsableWidth` is not a method; it is a property.

More About

- “Exploring Methods”

See Also

`methods` | `isprop` | `methodsview` | `isevent` | `isobject` | `class`

ismissing

Find table elements with missing values

Syntax

```
TF = ismissing(A)
TF = ismissing(A,id)
```

Description

`TF = ismissing(A)` returns a logical array, `TF`, that indicates which elements in the table `A` contain a missing value. The size of `TF` is the same as the size of `A`.

Indicators for missing data depend on the data type:

- `NaN` for numeric arrays
- `<undefined>` for categorical arrays
- empty string, `{''}`, for cell arrays of strings
- blank string, `[' ']`, for character arrays

`ismissing` ignores other data types.

`TF = ismissing(A,id)` treats the values in id as missing value indicators. You must include `NaN` (for floating-point variables), the empty string (for variables that are cell arrays of strings or character arrays), or the string `'<undefined>'` (for categorical variables) to have `ismissing` recognize them as missing value indicators.

Examples

Missing Values in Table with Various Data Types

Create a table with variables of different data types where each contains a missing value.

```
dblVar = [NaN;3;5;7;9];
singleVar = single([1;NaN;5;7;9]);
cellstrVar = {'one';'three';'';'seven';'nine'};
```

```
charVar = ['A';'C';'E';' ';'I'];
categoryVar = categorical({'red';'yellow';'blue';'violet';''});

A = table(dblVar,singleVar,cellstrVar,charVar,categoryVar)
```

```
A =

    dblVar    singleVar    cellstrVar    charVar    categoryVar
    ______    _________    __________    _______    ___________

    NaN         1          'one'         A          red
      3       NaN          'three'       C          yellow
      5         5          ''            E          blue
      7         7          'seven'                  violet
      9         9          'nine'        I          <undefined>
```

Find the table elements with missing values.

```
TF = ismissing(A)
```

```
TF =

     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

The size of `TF` is the same as the size of `A`.

Specify Indicators for Missing Values in Table

Create a table where `'NA'`, `''`-`99`, `NaN`, and `Inf` represent missing values.

```
dblVar = [NaN;3;Inf;7;9];
int8Var = int8([1;3;5;7;-99]);
cellstrVar = {'one';'three';'';'NA';'nine'};
charVar = ['A';'C';'E';' ';'I'];

A = table(dblVar,int8Var,cellstrVar,charVar)
```

```
A =

    dblVar    int8Var    cellstrVar    charVar
    ______    _______    __________    _______
```

```
    NaN         1        'one'         A
      3         3        'three'       C
    Inf         5        ''            E
      7         7        'NA'
      9       -99        'nine'        I
```

Find the missing values in the table `A`.

```
id = {'NA' '' -99 NaN Inf};
TF = ismissing(A,id)
```

```
TF =

     1     0     0     0
     0     0     0     0
     1     0     1     0
     0     0     1     1
     0     1     0     0
```

`ismissing` recognizes data in the each variable of the table, `A` as missing values. `ismissing` ignores trailing white space in character arrays. Therefore, since the empty string, `''`, is specified as a missing value indicator, `ismissing` identifies the empty string in the cell arrays of strings, `A.cellstrVar`, as well as the blank space in character array, `A.charVar` as missing values.

Input Arguments

`A` — Input table

table

Input table, specified as a table.

`id` — Missing value indicators

numeric vector | string | cell array containing numeric values and strings

Missing value indicators, specified as a numeric vector, string, or cell array containing numeric values and strings. Use strings to identify categories in categorical variables that you want to treat as missing values.

You must include `NaN` (for floating-point variables), the empty string (for variables that are cell arrays of strings or character arrays), or the string `'<undefined>'` (for categorical variables) to have `ismissing` recognize them as missing value indicators.

Example: `TF = ismissing(A,-99)` recognizes only `-99` as a missing value.

Data Types: `double` | `char` | `cell`

More About

Tips

- You can use `A(~any(TF,2),:)` to return only complete rows from `A` and `A(:,~any(TF,1))` to return only the variables from `A` with no missing values.
- Integer variables cannot store `NaN`; therefore, you must use a special integer value (otherwise unused) to indicate missing integer data. For example, `-99`.

Algorithms

`ismissing` treats leading and trailing white space differently for cell arrays of strings, character arrays, and categorical arrays.

- For cell arrays of strings, `ismissing` does not ignore white space. All strings must match exactly.
- For character arrays, `ismissing` ignores trailing white space. Therefore, when you specify the empty string, `''`, as a missing value indicator, `ismissing` identifies the empty string in cell arrays of strings, `''`, and the blank space in character arrays, `' '`, as missing values.
- For categorical arrays, `ismissing` ignores leading and trailing white space.

See Also

`isempty` | `isnan` | `isundefined` | `standardizeMissing` | `table`

isnan

Array elements that are NaN

Syntax

```
TF = isnan(A)
```

Description

`TF = isnan(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `NaN`s and logical 0 (`false`) where they are not. For a complex number `z`, `isnan(z)` returns `1` if either the real or imaginary part of `z` is `NaN`, and `0` if both the real and imaginary parts are finite or `Inf`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
A = [-2  -1  0  1  2];

isnan(1./A)
ans =
     0     0     0     0     0

isnan(0./A)
ans =
     0     0     1     0     0
```

See Also

`isfinite` | `nan` | `isinf` | `is*`

Introduced before R2006a

isnat

Determine NaT (Not-a-Time) elements

Syntax

```
tf = isnat(A)
```

Description

`tf = isnat(A)` returns an array the same size as the datetime array, `A`, containing logical 1 (`true`) where the elements of `A` are Not-a-Time (`NaT`s) and logical 0 (`false`) where they are not. `NaT` represents a datetime that is undefined.

Examples

Determine If Array Elements are NaT

Create a datetime array from numeric values containing `NaN`.

```
d = datetime(2014,[1 2 NaN 4],1)
```

```
d =

   01-Jan-2014   01-Feb-2014   NaT           01-Apr-2014
```

Determine if any elements of `d` are `NaT` (Not-a-Time).

```
isnat(d)
```

```
ans =

     0     0     1     0
```

Input Arguments

A — Input array

datetime array

Input array, specified as a datetime array.

See Also

`isfinite` | `isinf`

Introduced in R2014b

isnumeric

Determine if input is numeric array

Syntax

```
tf = isnumeric(A)
```

Description

`tf = isnumeric(A)` returns `true` if `A` is a numeric array and `false` otherwise.

A numeric array is any of the following numeric types and any subclasses of those types:

MATLAB Numeric Types

`single`	Single-precision floating-point array
`double`	Double-precision floating-point array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array

Examples

These examples show the values `isnumeric` returns when passed specific types:

```
% pi returns a numeric value
isnumeric(pi)
```

```
ans =
     1
% Complex numbers are numeric
isnumeric(3+7i)
ans =
     1
% Integers are numeric
isnumeric(uint8(1:255))
ans =
     1
% isnumeric returns a logical value
isnumeric(isnumeric(pi))
ans =
     0
```

See Also

isfloat | isnan | isreal | isinteger | isprime | isfinite | isinf | isa | is*

Introduced before R2006a

isobject

Determine if input is MATLAB object

Syntax

```
tf = isobject(A)
```

Description

`tf = isobject(A)` returns `true` if `A` is an object of a MATLAB class. Otherwise, it returns `false`.

Handle Graphics objects return `false`. Use `ishghandle` to test for Handle Graphics objects.

Instances of MATLAB fundamental classes return `false`. Use `isa` to test for any of these types. See "Fundamental MATLAB Classes" for more on these classes.

Examples

These examples show the values `isobject` returns when passed specific types:

Define the following MATLAB class:

```
classdef button < handle
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         obj.UiHandle = uicontrol('Position',pos,...
            'Style','pushbutton');
      end
   end
end
```

Determine which objects are MATLAB objects. For example:

```
h = button([20 20 60 60]);
isobject(h)
ans =
     1
isobject(h.UiHandle)
ans =
     0
```

Use `isjava` to test for Java objects in MATLAB, where it returns `false` for MATLAB objects:

```
isjava(h)
ans =
     0
```

Create an object that is a MATLAB numeric type:

```
a = pi;
isa(a,'double')
ans =
     1
isobject(a)
ans =
     0
```

See Also

`class` | `isa` | `is*`

Introduced before R2006a

isocaps

Compute isosurface end-cap geometry

Syntax

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

Description

`fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end-cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end-caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'enclose')` specifies whether the end-caps enclose data values above or below the value specified in `isovalue`. The string *`enclose`* can be either `above` (default) or `below`.

`fvc = isocaps(...,'whichplane')` specifies on which planes to draw the end-caps. Possible values for *`whichplane`* are `all` (default), `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, or `zmax`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end-caps in three arrays instead of the struct `fvc`.

`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

Examples

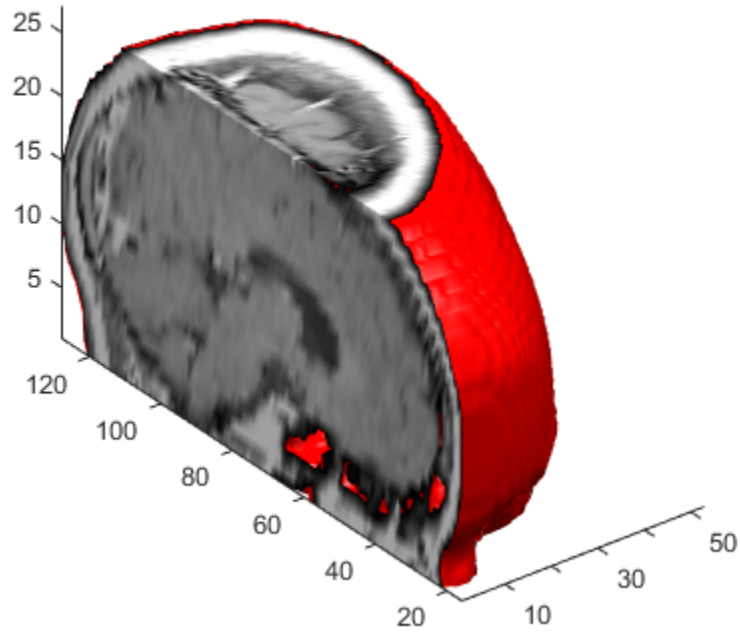
Compute Isosurface End-Cap Geometry

This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of isocaps to draw the end-caps on this cutaway volume.

The red isosurface shows the outline of the volume (skull) and the end-caps show what is inside of the volume.

The patch created from the end-cap data (`p2`) uses interpolated face coloring, which means the gray colormap and the light sources determine how it is colored. The isosurface patch (`p1`) used a flat red face color, which is affected by the lights, but does not use the colormap.

```
load mri
D = squeeze(D);
D(:,1:60,:) = [];
p1 = patch(isosurface(D, 5),'FaceColor','red',...
 'EdgeColor','none');
p2 = patch(isocaps(D, 5),'FaceColor','interp',...
 'EdgeColor','none');
view(3)
axis tight
daspect([1,1,.4])
colormap(gray(100))
camlight left
camlight
lighting gouraud
isonormals(D,p1)
```



More About

- "Isocaps Add Context to Visualizations"

See Also

`isosurface` | `isonormals` | `smooth3` | `subvolume` | `reducevolume` | `reducepatch`

Introduced before R2006a

isocolors

Calculate isosurface and patch colors

Syntax

```
nc = isocolors(X,Y,Z,C,vertices)
nc = isocolors(X,Y,Z,R,G,B,vertices)
nc = isocolors(C,vertices)
nc = isocolors(R,G,B,vertices)
nc = isocolors(...,PatchHandle)
isocolors(...,PatchHandle)
```

Description

`nc = isocolors(X,Y,Z,C,vertices)` computes the colors of isosurface (patch object) vertices (`vertices`) using color values `C`. Arrays `X`, `Y`, `Z` define the coordinates for the color data in `C` and must be monotonic vectors that represent a Cartesian, axis-aligned grid (as if produced by `meshgrid`). The colors are returned in `nc`. `C` must be 3-D (index colors).

`nc = isocolors(X,Y,Z,R,G,B,vertices)` uses `R`, `G`, `B` as the red, green, and blue color arrays (true color).

`nc = isocolors(C,vertices)`, and `nc = isocolors(R,G,B,vertices)` assume `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

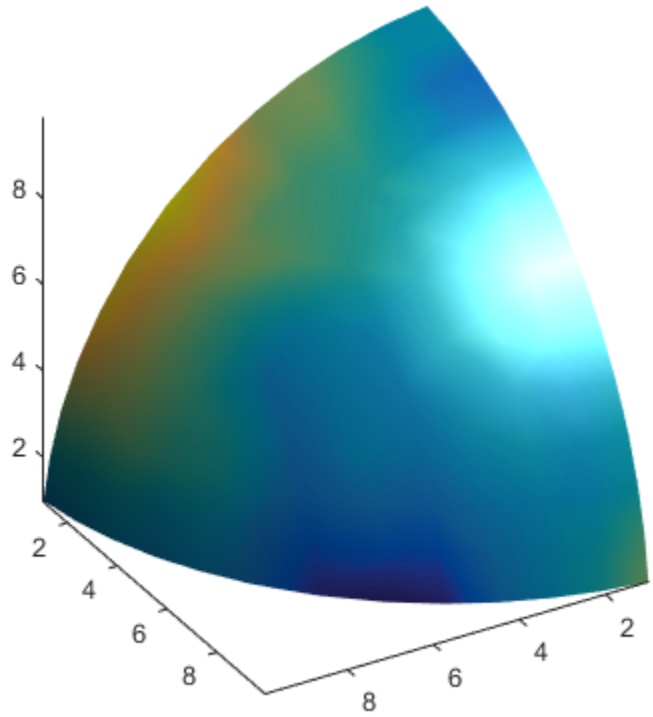
`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

Examples

Indexed Color Data

This example displays an isosurface and colors it with random data using indexed color.

```
[x,y,z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
cdata = smooth3(rand(size(data)),'box',7);
p = patch(isosurface(x,y,z,data,10));
isonormals(x,y,z,data,p)
isocolors(x,y,z,cdata,p)
p.FaceColor = 'interp';
p.EdgeColor = 'none';
view(150,30)
daspect([1 1 1])
axis tight
camlight
lighting gouraud
```

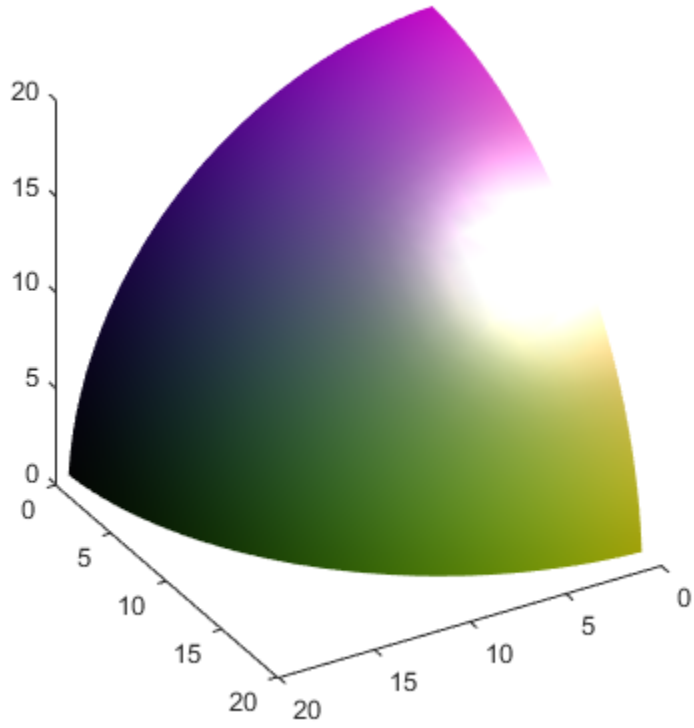


True Color Data

This example displays an isosurface and colors it with true color (RGB) data.

```
[x,y,z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p)
[r,g,b] = meshgrid(20:-1:1,1:20,1:20);
isocolors(x,y,z,r/20,g/20,b/20,p)
p.FaceColor = 'interp';
p.EdgeColor = 'none';
view(150,30)
daspect([1 1 1])
```

```
camlight
lighting gouraud
```

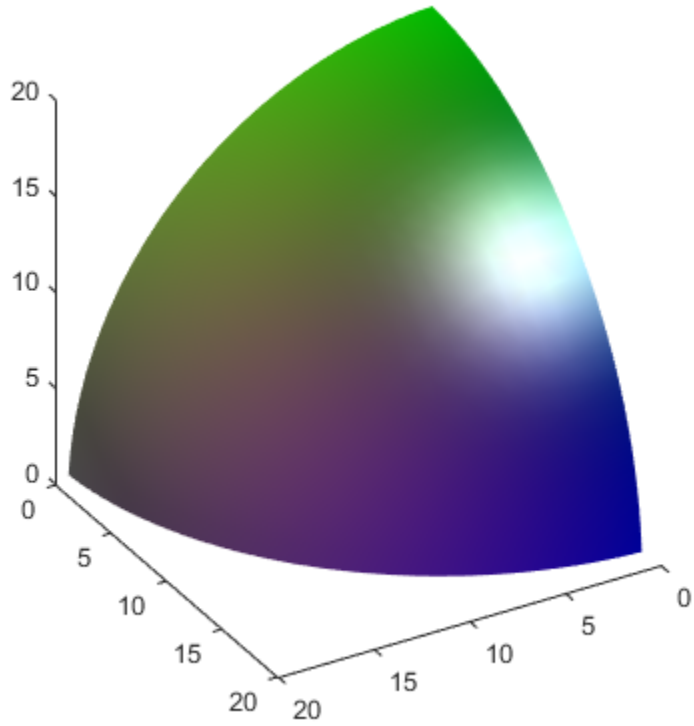


Modified True Color Data

This example uses `isocolors` to calculate the true color data using the isosurface's (patch object's) vertices, but then returns the color data in a variable (`c`) in order to modify the values. It then explicitly sets the isosurface's FaceVertexCData to the new data (`1-c`).

```
[x,y,z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(data,20));
isonormals(data,p)
```

```
[r,g,b] = meshgrid(20:-1:1,1:20,1:20);
c = isocolors(r/20,g/20,b/20,p);
p.FaceVertexCData = 1-c;
p.FaceColor = 'interp';
p.EdgeColor = 'none';
view(150,30)
daspect([1 1 1])
camlight
lighting gouraud
```



More About

- “Interpolating in Indexed Color Versus Truecolor”

See Also

isocaps | isonormals | isosurface | reducepatch | reducevolume | smooth3 | subvolume

Introduced before R2006a

isonormals

Compute normals of isosurface vertices

Syntax

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p)
n = isonormals(X,Y,Z,V,p)
n = isonormals(...,'negate')
isonormals(V,p)
isonormals(X,Y,Z,V,p)
```

Description

`n = isonormals(X,Y,Z,V,vertices)` computes the normals of the isosurface vertices from the vertex list, `vertices`, using the gradient of the data `V`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The computed normals are returned in `n`.

`n = isonormals(V,vertices)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`n = isonormals(V,p)` and `n = isonormals(X,Y,Z,V,p)` compute normals from the vertices of the patch identified by the handle `p`.

`n = isonormals(...,'negate')` negates (reverses the direction of) the normals.

`isonormals(V,p)` and `isonormals(X,Y,Z,V,p)` set the `VertexNormals` property of the patch identified by the handle `p` to the computed normals rather than returning the values.

Examples

Isosurface Using Different Types of Surface Normals

Compare the effect of different surface normals on the visual appearance of lit isosurfaces. In one case, the triangles used to draw the isosurface define the normals. In

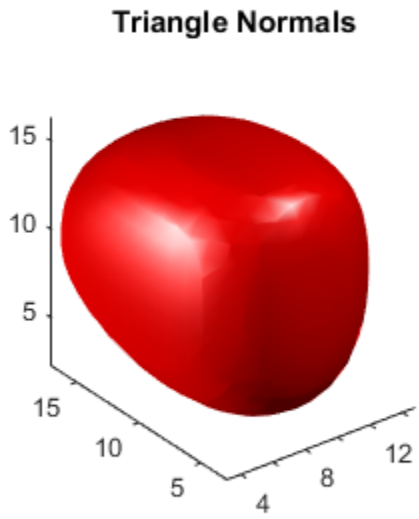
the other, the isonormals function uses the volume data to calculate the vertex normals based on the gradient of the data points. The latter approach generally produces a smoother-appearing isosurface.

Define a 3-D array of volume data.

```
data = cat(3,[0 .2 0; 0 .3 0; 0 0 0],...
    [.1 .2 0; 0 1 0; .2 .7 0],...
    [0 .4 .2; .2 .4 0;.1 .1 0]);
data = interp3(data,3,'cubic');
```

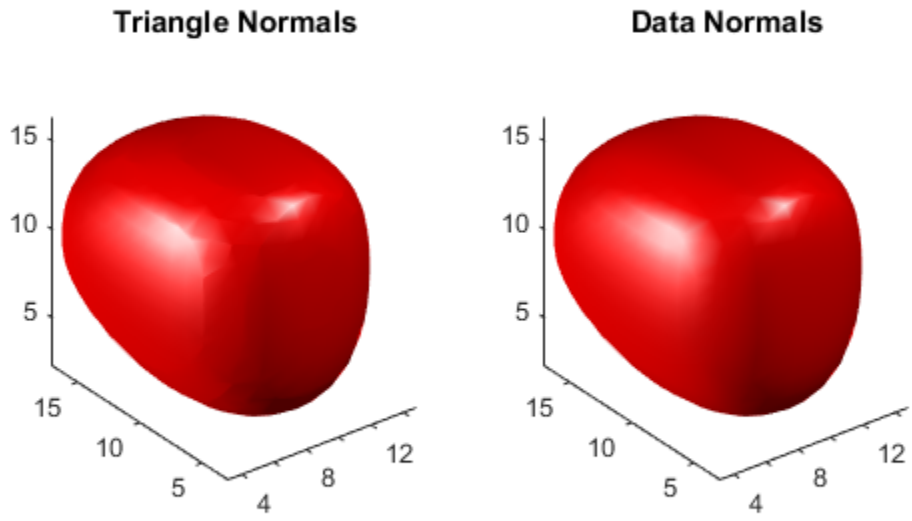
Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals.

```
figure
subplot(1,2,1)
fv = isosurface(data,.5);
p1 = patch(fv,'FaceColor','red','EdgeColor','none');
view(3)
daspect([1,1,1])
axis tight
camlight
camlight(-80,-10)
lighting gouraud
title('Triangle Normals')
```



Draw the same lit isosurface using normals calculated from the volume data.

```
subplot(1,2,2)
fv = isosurface(data,.5);
p2 = patch(fv,'FaceColor','red','EdgeColor','none');
isonormals(data,p2)
view(3)
daspect([1 1 1])
axis tight
camlight
camlight(-80,-10)
lighting gouraud
title('Data Normals')
```



These isosurfaces illustrate the difference between triangle and data normals.

See Also

`interp3` | `isosurface` | `isocaps` | `smooth3` | `subvolume` | `reducevolume` | `reducepatch`

Introduced before R2006a

isordinal

Determine whether input is ordinal categorical array

Syntax

```
tf = isordinal(A)
```

Description

`tf = isordinal(A)` returns logical `1` (`true`) if `A` is an ordinal categorical array. Otherwise, `isordinal` returns logical `0` (`false`).

If a categorical array is ordinal, you can use relational operations for inequality comparisons, such as greater and less than, in addition to tests for equality.

Examples

Determine Whether Categorical Array Is Ordinal

Create a categorical array containing the sizes of 10 objects. Use the names `small`, `medium`, and `large` for the values `'S'`, `'M'`, and `'L'`.

```
A = categorical({'M';'L';'S';'S';'M';'L';'M';'L';'M';'S'},...
    {'S','M','L'},{'small','medium','large'})
```

```
A =

     medium
     large
     small
     small
     medium
     large
     medium
     large
     medium
     small
```

Determine if the categories of A have a mathematical ordering.

```
isordinal(A)
```

```
ans =

     0
```

A is not ordinal. You must use the `'Ordinal',true` name-value pair argument in the function `categorical` to create an ordinal categorical array.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

More About

Tips

- To convert a categorical array, A, from nonordinal to ordinal, use `A = categorical(A,'Ordinal',true)`.
- To convert a categorical array, A, from ordinal to nonordinal, use `A = categorical(A,'Ordinal',false)`.
- "Ordinal Categorical Arrays"

See Also

`categorical` | `categories`

isosurface

Extract isosurface data from volume data

Syntax

```
fv = isosurface(X,Y,Z,V,isovalue)
fv = isosurface(V,isovalue)
fvc = isosurface(...,colors)
fv = isosurface(...,'noshare')
fv = isosurface(...,'verbose')
[f,v] = isosurface(...)
[f,v,c] = isosurface(...)
isosurface(...)
```

Description

`fv = isosurface(X,Y,Z,V,isovalue)` computes isosurface data from the volume data `V` at the isosurface value specified in `isovalue`. That is, the isosurface connects points that have the specified value much the way contour lines connect points of equal elevation.

The arrays `X`, `Y`, and `Z` represent a Cartesian, axis-aligned grid. `V` contains the corresponding values at these grid points. The coordinate arrays (`X`, `Y`, and `Z`) must be monotonic and conform to the format produced by `meshgrid`. `V` must be a 3D volume array of the same size as `X`, `Y`, and `Z`.

The `struct` `fv` contains the faces and vertices of the isosurface, which you can pass directly to the `patch` command.

`fv = isosurface(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isosurface(...,colors)` interpolates the array `colors` onto the scalar field and returns the interpolated values in the `facevertexcdata` field of the `fvc` structure. The size of the `colors` array must be the same as `V`. The `colors` argument enables you to control the color mapping of the isosurface with data different from that

used to calculate the isosurface (e.g., temperature data superimposed on a wind current isosurface).

`fv = isosurface(...,'noshare')` does not create shared vertices. This is faster, but produces a larger set of vertices.

`fv = isosurface(...,'verbose')` prints progress messages to the command window as the computation progresses.

`[f,v] = isosurface(...)` or `[f,v,c] = isosurface(...)` returns the faces and vertices (and `faceVertexcCData`) in separate arrays instead of a struct.

`isosurface(...)` with no output arguments, creates a patch in the current axes with the computed faces and vertices. If no current axes exists, a new axes is created with a 3-D view.

Special Case Behavior — isosurface Called with No Output Arguments

If there is no current axes and you call `isosurface` with without assigning output arguments, MATLAB creates a new axes, sets it to a 3-D view, and adds lighting to the isosurface graph.

Examples

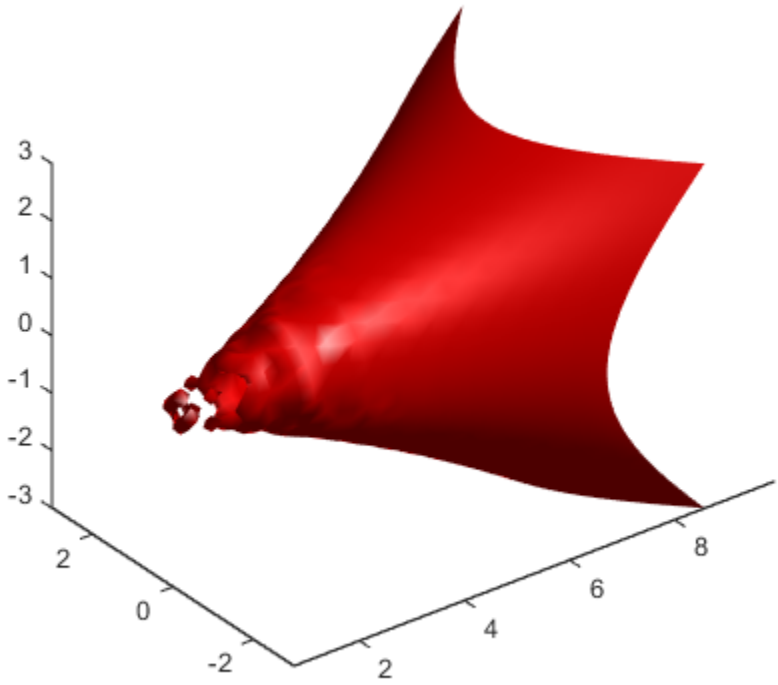
Example 1

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). The isosurface is drawn at the data value of -3. The statements that follow the `patch` command prepare the isosurface for lighting by

- Recalculating the isosurface normals based on the volume data (`isonormals`)
- Setting the face and edge color (`set`, `FaceColor`, `EdgeColor`)
- Specifying the view (`daspect`, `view`)
- Adding lights (`camlight`, `lighting`)

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
```

```
isonormals(x,y,z,v,p)
p.FaceColor = 'red';
p.EdgeColor = 'none';
daspect([1,1,1])
view(3); axis tight
camlight
lighting gouraud
```

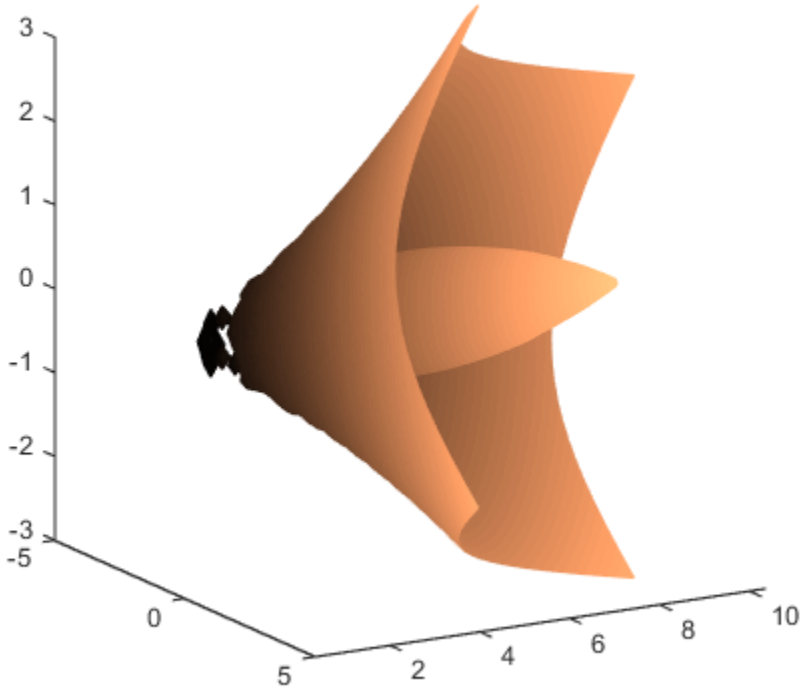


Example 2

Visualize the same flow data as above, but color-code the surface to indicate magnitude along the X-axis. Use a sixth argument to `isosurface`, which provides a means to overlay another data set by coloring the resulting isosurface. The `colors` variable is a

vector containing a scalar value for each vertex in the isosurface, to be portrayed with the current color map. In this case, it is one of the variables that define the surface, but it could be entirely independent. You can apply a different color scheme by changing the current figure color map.

```
[x,y,z,v] = flow;
[faces,verts,colors] = isosurface(x,y,z,v,-3,x);
patch('Vertices', verts, 'Faces', faces, ...
    'FaceVertexCData', colors, ...
    'FaceColor','interp', ...
    'edgecolor', 'interp');
view(30,-15);
axis vis3d;
colormap copper
```



More About

Tips

You can pass the `fv` structure created by `isosurface` directly to the `patch` command, but you cannot pass the individual faces and vertices arrays (`f`, `v`) to `patch` without specifying property names. For example,

```
patch(isosurface(X,Y,Z,V,isovalue))
```

or

```
[f,v] = isosurface(X,Y,Z,V,isovalue);
patch('Faces',f,'Vertices',v)
```

- "Connecting Equal Values with Isosurfaces"

See Also

`isonormals` | `shrinkfaces` | `smooth3` | `subvolume`

Introduced before R2006a

ispc

Determine if version is for Windows (PC) platform

Syntax

```
tf = ispc
```

Description

`tf = ispc` returns logical `1` (`true`) if the version of MATLAB software is for the Microsoft Windows platform. Otherwise, it returns logical `0` (`false`).

See Also

`computer` | `is*` | `ismac` | `isstudent` | `isunix`

Introduced before R2006a

ispref

Test for existence of preference

Syntax

```
ispref('group','pref')
ispref('group')
ispref('group',{'pref1','pref2',...'prefn'})
```

Description

`ispref('group','pref')` returns `1` if the preference specified by `group` and `pref` exists, and `0` otherwise.

`ispref('group')` returns `1` if the GROUP exists, and `0` otherwise.

`ispref('group',{'pref1','pref2',...'prefn'})` returns a logical array the same length as the cell array of preference names, containing `1` where each preference exists, and `0` elsewhere.

Examples

```
addpref('mytoolbox','version','1.0')
ispref('mytoolbox','version')

ans =
     1.0
```

See Also

`addpref` | `rmpref` | `getpref` | `setpref` | `uigetpref` | `uisetpref`

Introduced before R2006a

isprime

Determine which array elements are prime

Syntax

```
TF = isprime(X)
```

Description

`TF = isprime(X)` returns a logical array the same size as `X`. The value at `TF(i)` is `true` when `X(i)` is a prime number. Otherwise, the value is `false`.

Examples

Determine if Double Integer Values Are Prime

```
tf = isprime([2 3 0 6 10])
```

```
tf =

     1     1     0     0     0
```

2 and 3 are prime, but 0, 6, and 10 are not.

Determine if Unsigned Integer Values Are Prime

```
x = uint16([333 71 99]);
tf = isprime(x)
```

```
tf =

     0     1     0
```

71 is prime, but 333 and 99 are not.

Input Arguments

X — Input values

scalar, vector, or array of real, nonnegative integer values

Input values, specified as a scalar, vector, or array of real, nonnegative integer values.

Example: `17`

Example: `[1 2 3 4]`

Example: `int16([127 255 4095])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`is*` | `primes`

Introduced before R2006a

isprop

Determine if property of object

Syntax

```
tf = isprop(obj,'PropertyName')
```

Description

`tf = isprop(obj,'PropertyName')` returns `true` if the specified `PropertyName` is a property of object `obj`. Otherwise, `isprop` returns logical `false`.

If `obj` is an array, `isprop` returns a logical array the same size as `obj`. Each true element of `tf` corresponds to an element of `obj` that has the property, `PropertyName`.

Note: If `obj` is an empty object or an array of empty objects, `isprop` returns an empty logical array, even if `PropertyName` is a property of `obj`.

While `isprop` returns `true` if the class of an object defines a property of that name, classes can control access to property values by defining property attributes. Property access can be defined as:

- Readable and writable
- Read only
- Write only
- Accessible only to certain class methods

Therefore, `isprop` might indicate that a property exists, but you might not be able to access that property. For more information, see “Getting Information About Properties”.

Examples

This example uses `isprop` to determine if `XDataSource` is a property of object `h` before attempting to set the property value:

```
h = plot(1:10);
if isprop(h,'XDataSource')
   set(h,'XDataSource','x')
else
   error(['XDataSource not a property of class ',class(h)])
end
```

Since `XDataSource` is a property of `h`, its value is set to `'x'`.

See Also

`properties` | `ismethod`

Introduced before R2006a

isprop (COM)

Determine whether input is COM object property

Syntax

```
tf = isprop(h,'propertyname')
```

Description

`tf = isprop(h,'propertyname')` returns logical `1` (`true`) if the specified `name` is a property of COM object `h`. Otherwise, returns logical `0` (`false`).

COM functions are available on Microsoft Windows systems only.

Examples

Test a property of an instance of a Microsoft Excel application.

```
h = actxserver('Excel.Application');
isprop(h,'UsableWidth')
```

MATLAB returns `true`, `UsableWidth` is a property.

Try the same test on `SaveWorkspace`.

```
isprop(h,'SaveWorkspace')
```

MATLAB returns `false`. `SaveWorkspace` is not a property; it is a method.

More About

- “Exploring Properties”

See Also

`inspect` | `ismethod` | `isevent`

isprotected

Determine whether categories of categorical array are protected

Syntax

```
tf = isprotected(A)
```

Description

`tf = isprotected(A)` returns logical `1` (`true`) if the categories of `A` are protected. Otherwise, `isprotected` returns logical `0` (`false`).

- `true` — When you assign new values to `B`, the values must belong to one of the existing categories. Therefore, you only can combine arrays that have the same categories. To add new categories to `B`, you must use the `addcats` function.
- `false` — When you assign new values to `B`, the categories update automatically. Therefore, you can combine (nonordinal) categorical arrays that have different categories. The categories can update to include the categories from both arrays.

Examples

Determine Whether Categories Are Protected

Create a categorical array containing the sizes of 10 objects. Use the names `small`, `medium`, and `large` for the values `'S'`, `'M'`, and `'L'`.

```
valueset = {'S','M','L'};
catnames = {'small','medium','large'};

A = categorical({'M';'L';'S';'S';'M';'L';'M';'L';'M';'S'},...
    valueset,catnames,'Ordinal',true)
```

```
A =

     medium
     large
```

```
     small
     small
     medium
     large
     medium
     large
     medium
     small
```

A is a 10-by-1 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans =

    'small'
    'medium'
    'large'
```

Determine whether the categories of A are protected.

```
tf = isprotected(A)
```

```
tf =

     1
```

Since A is an ordinal categorical array, the categories are protected. If you try to add a new value that does not belong to one of the existing categories, for example `A(11) = 'xlarge'`, then MATLAB returns an error.

First, use `addcats` to add a new category for `xlarge`.

```
A = addcats(A,'xlarge','After','large');
```

Since A is protected, you can now add a value for `xlarge` since it has an existing category.

```
A(11) = 'xlarge'
```

```
A =

     medium
```

```
large
small
small
medium
large
medium
large
medium
small
xlarge
```

A is now a 11-by-1 categorical array with four categories, such that `small < medium < large < xlarge`.

- "Work with Protected Categorical Arrays"

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

The categories of ordinal categorical arrays are always protected.

See Also

`categorical` | `categories`

isreal

Determine whether array is real

Syntax

```
tf = isreal(A)
```

Description

`tf = isreal(A)` returns logical `1` (`true`) when `A` does not have an imaginary part. Otherwise, it returns logical `0` (`false`).

If `A` has a stored imaginary part with value `0`, then `isreal(A)` returns logical `0` (`false`).

Examples

Determine Whether Matrix Contains All Real Values

Define a 3-by-4 matrix, `A`.

```
A = [7 3+4i 2 5i;...
     2i 1+3i 12 345;...
     52 108 78 3];
```

Determine whether the array is real.

```
tf = isreal(A)
```

```
tf =

     0
```

Since `A` contains complex elements, `isreal` returns false.

Define Complex Number with Zero-Valued Imaginary Part

Use the `complex` function to create a scalar, `A`, with zero-valued imaginary part.

```
A = complex(12)

A =

  12.0000 + 0.0000i
```

Determine whether A is real.

```
tf = isreal(A)

tf =

     0
```

A is not real because it has an imaginary part, even though the value of the imaginary part is 0.

Determine whether A contains any elements with zero-valued imaginary part.

```
~any(imag(A))

ans =

     1
```

A contains elements with zero-valued imaginary part.

Computation Resulting in Zero-Valued Imaginary Part

Define two complex scalars, x and y.

```
x=3+4i;
y=5-4i;
```

Determine whether the addition of two complex scalars, x and y, is real.

```
A = x+y

A =

     8
```

MATLAB drops the zero imaginary part.

```
isreal(A)
```

```
ans =

     1
```

A is real since it does not have an imaginary part.

Find Real Elements in Cell Array

Create a cell array.

```
C{1,1} = pi;                % double
C{2,1} = 'John Doe';        % char array
C{3,1} = 2 + 4i;            % complex double
C{4,1} = ispc;              % logical
C{5,1} = magic(3);          % double array
C{6,1} = complex(5,0)       % complex double
```

```
C =

    [          3.1416]
    'John Doe'
    [2.0000 + 4.0000i]
    [               1]
    [3x3 double]
    [5.0000 + 0.0000i]
```

C is a 1-by-6 cell array.

Loop over the elements of a cell array to distinguish between real and complex elements.

```
for k = 1:6
x(k,1) = isreal(C{k,1});
end

x
```

```
x =

     1
     1
     0
     1
     1
     0
```

All but `C{3,1}` and `C{6,1}` are real arrays.

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

- For numeric data types, if `A` does not have an imaginary part, `isreal` returns `true`; if `A` does have an imaginary part `isreal` returns `false`.
- For `logical` and `char` data types, `isreal` always returns `true`.
- For `table`, `cell`, `struct`, `datetime`, `function_handle`, and `object` data types, `isreal` always returns `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `datetime` | `function_handle`
Complex Number Support: Yes

More About

Tips

- `isreal(complex(A))` always returns `false`, even when the imaginary part is all zeros.
- `~isreal(x)` detects arrays that have an imaginary part, even if it is all zeros.
- “Complex Numbers”

See Also

`complex` | `isa` | `isfinite` | `isinf` | `isnan` | `isnumeric` | `isprime`

Introduced before R2006a

isrow

Determine whether input is row vector

Syntax

```
isrow(V)
```

Description

`isrow(V)` returns logical 1 (`true`) if `size`(V) returns `[1 n]` with a nonnegative integer value `n`, and logical 0 (`false`) otherwise.

Examples

Determine if a vector is a row. This example is a column so `isrow` returns 0:

```
V = rand(5,1);
isrow(V)
ans =
     0
```

Transpose the vector to make it a row. `isrow` returns 1:

```
V1 = V';
isrow(V1)
ans =
     1
```

See Also

`iscolumn` | `ismatrix` | `isscalar` | `isvector`

isscalar

Determine whether input is scalar

Syntax

```
isscalar(A)
```

Description

`isscalar(A)` returns logical `1` (`true`) if `size(A)` returns [1 1], and logical `0` (`false`) otherwise.

Examples

Test matrix `A` and one element of the matrix:

```
A = rand(5);

isscalar(A)
ans =
     0

isscalar(A(3,2))
ans =
     1
```

See Also

`isvector` | `isrow` | `ismatrix` | `iscolumn` | `islogical` | `ischar` | `isa` | `is*`

Introduced before R2006a

issorted

Determine whether set elements are in sorted order

Syntax

```
TF = issorted(A)
TF = issorted(A, 'rows')
```

Description

`TF = issorted(A)` returns logical `1` (`true`) if the elements of `A` are in sorted order, and logical `0` (`false`) otherwise. Input `A` can be a vector or an N-by-1 or 1-by-N cell array of strings. `A` is considered to be sorted if `A` and the output of `sort(A)` are equal.

`TF = issorted(A, 'rows')` returns logical `1` (`true`) if the rows of two-dimensional matrix `A` are in sorted order, and logical `0` (`false`) otherwise. Matrix `A` is considered to be sorted if `A` and the output of `sortrows(A)` are equal.

Note Only the `issorted(A)` syntax supports `A` as a cell array of strings.

The `issorted(A,'rows')` syntax supports `A` as a categorical array, but not as a cell array of strings.

Examples

Example 1 — Using issorted on a vector

```
A = [5 12 33 39 78 90 95 107 128 131];

issorted(A)
ans =
     1
```

Example 2 — Using issorted on a matrix

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

issorted(A, 'rows')
ans =
     0

B = sortrows(A)
B =
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
    17    24     1     8    15
    23     5     7    14    16

issorted(B)
ans =
     1
```

Example 3 — Using issorted on a cell array

```
x = {'one'; 'two'; 'three'; 'four'; 'five'};
issorted(x)
ans =
     0

y = sort(x)
y =
    'five'
    'four'
    'one'
    'three'
    'two'

issorted(y)
```

More About

Tips

For character arrays, `issorted` uses ASCII, rather than alphabetical, order.

You cannot use `issorted` on arrays of greater than two dimensions.

See Also

`sort` | `sortrows` | `unique` | `ismember` | `intersect` | `union` | `setdiff` | `setxor` | `is*`

Introduced before R2006a

isspace

Array elements that are space characters

Syntax

```
tf = isspace('str')
```

Description

`tf = isspace('str')` returns an array the same size as `'str'` containing logical `1` (`true`) where the elements of `str` are Unicode-represented whitespace characters, and logical `0` (`false`) where they are not.

Whitespace characters for which `isspace` returns `true` include tab, line feed, vertical tab, form feed, carriage return, and space, in addition to a number of other Unicode characters. To see all characters for which `isspace` returns `true`, enter the following command, and then look up the returned decimal codes in a Unicode reference:

```
find(isspace(char(1):char(intmax('uint16'))))
```

Examples

```
isspace('  Find spa ces ')
  Columns 1 through 13
     1   1   0   0   0   0   1   0   0   0   1   0   0
  Columns 14 through 15
     0   1
```

See Also

`isletter` | `ischar` | `isstrprop` | `strings` | `isa` | `is*`

Introduced before R2006a

issparse

Determine whether input is sparse

Syntax

```
TF = issparse(S)
```

Description

`TF = issparse(S)` returns logical 1 (`true`) if the storage class of `S` is sparse and logical 0 (`false`) otherwise.

See Also

`is*` | `sparse` | `full`

Introduced before R2006a

isstr

Determine whether input is character array

Note: `isstr` is not recommended. Use `ischar` instead.

See Also

`ischar` | `isa` | `is*`

Introduced before R2006a

isstrprop

Determine whether string is of specified category

Syntax

```
tf = isstrprop('str', 'category')
```

Description

`tf = isstrprop('str', 'category')` returns a logical array the same size as `str` containing logical `1` (`true`) where the elements of `str` belong to the specified *`category`*, and logical `0` (`false`) where they do not.

The `str` input can be a character array, cell array, or any MATLAB numeric type. If `str` is a cell array, then the return value is a cell array of the same shape as `str`.

The *`category`* input can be any of the strings shown in the left column below:

Category	Description
`alpha`	`True` for those elements of `str` that are alphabetic
`alphanum`	`True` for those elements of `str` that are alphanumeric
`cntrl`	`True` for those elements of `str` that are control characters (for example, `char(0:20)`)
`digit`	`True` for those elements of `str` that are numeric digits

`graphic`	True for those elements of str that are graphic characters. These are all values that represent any characters except for the following:
	
	`unassigned, space, line separator,`
	`paragraph separator, control characters,`
	`Unicode format control characters,`
	`private user-defined characters,`
	`Unicode surrogate characters,`
	`Unicode other characters`

`lower`	`True` for those elements of `str` that are lowercase letters

Category	Description
`print`	`True` for those elements of `str` that are graphic characters, plus `char(32)`
`punct`	`True` for those elements of `str` that are punctuation characters
`wspace`	`True` for those elements of `str` that are white-space characters. This range includes the ANSI® C definition of white space, `{' ','\t','\n','\r','\v','\f'}`, in addition to a number of other Unicode characters.
`upper`	`True` for those elements of `str` that are uppercase letters
`xdigit`	`True` for those elements of `str` that are valid hexadecimal digits

Examples

Test for alphabetic characters in a string:

```
A = isstrprop('abc123def', 'alpha')
A =
   1 1 1 0 0 0 1 1 1
```

Test for numeric digits in a string:

```
A = isstrprop('abc123def', 'digit')
A =
   0 0 0 1 1 1 0 0 0
```

Test for hexadecimal digits in a string:

```
A = isstrprop('abcd1234efgh', 'xdigit')
A =
   1 1 1 1 1 1 1 1 1 1 0 0
```

Test for numeric digits in a character array:

```
A = isstrprop(char([97 98 99 49 50 51 101 102 103]), ...
              'digit')
A =
   0 0 0 1 1 1 0 0 0
```

Test for alphabetic characters in a two-dimensional cell array:

```
A = isstrprop({'abc123def';'456ghi789'}, 'alpha')
```

```
A =
   [1x9 logical]
   [1x9 logical]

A{:,:}
ans =
   1 1 1 0 0 0 1 1 1
   0 0 0 1 1 1 0 0 0
```

Test for white-space characters in a string:

```
A = isstrprop(sprintf('a bc\n'), 'wspace')
A =
   0 1 0 0 1
```

More About

Tips

Numbers of type `double` are converted to `int32` according to MATLAB rules of double-to-integer conversion. Numbers of type `int64` and `uint64` bigger than `int32(inf)` saturate to `int32(inf)`.

MATLAB classifies the elements of the `str` input according to the Unicode definition of the specified category. If the numeric value of an element in the input array falls within the range that defines a Unicode character category, then this element is classified as being of that category. The set of Unicode character codes includes the set of ASCII character codes, but also covers a large number of languages beyond the scope of the ASCII set. The classification of characters is dependent on the global location of the platform on which MATLAB is installed.

Whitespace characters for which the `wspace` option returns `true` include tab, line feed, vertical tab, form feed, carriage return, and space, in addition to a number of other Unicode characters. To see all characters for which the `wspace` option returns `true`, enter the following command, and then look up the returned decimal codes in a Unicode reference:

```
find(isstrprop(char(1):char(intmax('uint16')), 'wspace'))
```

See Also

`strings` | `ischar` | `isletter` | `isspace` | `iscellstr` | `isnumeric` | `isa` | `is*`

Introduced before R2006a

isstruct

Determine whether input is structure array

Syntax

```
tf = isstruct(A)
```

Description

`tf = isstruct(A)` returns logical `1` (`true`) if `A` is a MATLAB structure and logical `0` (`false`) otherwise.

Examples

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];

isstruct(patient)

ans =

     1
```

More About

- dynamic field names

See Also

`struct` | `istable` | `iscell` | `ischar` | `isfield` | `isobject` | `isnumeric` | `islogical` | `isa` | `is*`

Introduced before R2006a

isstudent

Determine if version is Student Version

Syntax

```
tf = isstudent
```

Description

`tf = isstudent` returns logical `1` (`true`) if the version of MATLAB software is the Student Version, and returns logical `0` (`false`) for commercial versions.

See Also

`ver` | `version` | `license` | `ispc` | `isunix` | `is*`

Introduced before R2006a

issymmetric

Determine if matrix is symmetric or skew-symmetric

Syntax

```
tf = issymmetric(A)
tf = issymmetric(A,skewOption)
```

Description

`tf = issymmetric(A)` returns logical `1` (`true`) if square matrix `A` is symmetric; otherwise, it returns logical `0` (`false`).

`tf = issymmetric(A,skewOption)` specifies the type of the test. Specify `skewOption` as `'skew'` to determine if `A` is skew-symmetric.

Examples

Test if Hermitian Matrix Is Symmetric

Create a 3-by-3 matrix.

```
A = [1 0 1i; 0 1 0;-1i 0 1]
```

```
A =

   1.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 1.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 - 1.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i
```

The matrix is Hermitian and has a real-valued diagonal.

Test whether the matrix is symmetric.

```
tf = issymmetric(A)
```

```
tf =

     0
```

The result is logical `0` (`false`) because `A` is not symmetric. In this case, `A` is equal to its complex conjugate transpose, `A'`, but not its nonconjugate transpose, `A.'`.

Change the element in `A(3,1)` to be `1i`.

```
A(3,1) = 1i;
```

Determine whether the modified matrix is symmetric.

```
tf = issymmetric(A)
```

```
tf =

     1
```

The matrix, `A`, is now symmetric because it is equal to its nonconjugate transpose, `A.'`.

Test if Matrix Is Skew-Symmetric

Create a 4-by-4 matrix.

```
A = [0 1 -2 5; -1 0 3 -4; 2 -3 0 6; -5 4 -6 0]
```

```
A =

     0     1    -2     5
    -1     0     3    -4
     2    -3     0     6
    -5     4    -6     0
```

The matrix is real and has a diagonal of zeros.

Specify `skewOption` as `'skew'` to determine whether the matrix is skew-symmetric.

```
tf = issymmetric(A,'skew')
```

```
tf =

     1
```

The matrix, A, is skew-symmetric since it is equal to the negation of its nonconjugate transpose, `-A.'`.

Input Arguments

A — Input matrix

numeric matrix

Input matrix, specified as a numeric matrix. If A is not square, then `issymmetric` returns logical `0` (`false`).

Data Types: `single` | `double`
Complex Number Support: Yes

skewOption — Test type

`'nonskew'` (default) | `'skew'`

Test type, specified as `'nonskew'` (default) or `'skew'`. Specify `'skew'` to test whether A is skew-symmetric. Specifying `issymmetric(A,'nonskew')` is equivalent to `issymmetric(A)`.

Data Types: `char`

More About

Symmetric Matrix

- A square matrix, A, is symmetric if it is equal to its nonconjugate transpose, `A = A.'`.

  In terms of the matrix elements, this means that

  $$a_{i,j} = a_{j,i} .$$

- Since real matrices are unaffected by complex conjugation, a real matrix that is symmetric is also Hermitian. For example, the matrix

  $$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

is both symmetric and Hermitian.

Skew-Symmetric Matrix

- A square matrix, `A`, is skew-symmetric if it is equal to the negation of its nonconjugate transpose, `A = -A.'`.

  In terms of the matrix elements, this means that

  $$a_{i,j} = -a_{j,i} .$$

- Since real matrices are unaffected by complex conjugation, a real matrix that is skew-symmetric is also skew-Hermitian. For example, the matrix

  $$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

  is both skew-symmetric and skew-Hermitian.

See Also

`ctranspose` | `ishermitian` | `isreal` | `transpose`

Introduced in R2014a

isTiled

Class: Tiff

Determine if tiled image

Syntax

```
tf = isTiled(tiffobj)
```

Description

`tf = isTiled(tiffobj)` returns `true` if the image has a tiled organization and `false` if the image has a stripped organization.

Examples

Determine if Image Has Tiled Organization

Open a `Tiff` object and check if the image in the TIFF file has a tiled or stripped organization.

```
t = Tiff('example.tif','r');
tf = isTiled(t)
```

```
tf =

     1
```

The image has a tiled organization.

Close the `Tiff` object.

```
close(t)
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFIsTiled` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

istable

Determine whether input is table

Syntax

```
tf = istable(T)
```

Description

`tf = istable(T)` returns logical `1` (`true`) if `T` is a table. Otherwise, it returns logical `0` (`false`) otherwise.

Examples

Determine if Workspace Variable Is Table

Create a workspace variable, `T`.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T =

    Var1          Var2          Var3    Var4
    ____    ________________    ____    _____

    M       45          45    'NY'    true
    F       41          32    'CA'    false
    M       40          34    'MA'    false
```

Verify that the workspace variable, `T`, is a table.

```
istable(T)
```

```
ans =

     1
```

T is a table.

Determine if Subset of Table Is Table

Create a table, T.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T =

    Var1          Var2          Var3    Var4
    ____    ______________    ____    _____

    M       45            45    'NY'    true
    F       41            32    'CA'    false
    M       40            34    'MA'    false
```

Determine if the subset of table T that contains only the second and fourth variables is a table.

```
istable(T{:,[2 4]})
```

```
ans =

     0
```

Conversely, accessing data with curly braces, `T{:,[2 4]}`, returns a matrix and not a table.

Input Arguments

T — Input variable

workspace variable

Input variable, specified as a workspace variable. T can be any data type.

See Also

`iscell` | `islogical` | `isnumeric` | `isobject` | `isstruct` | `table`

istril

Determine if matrix is lower triangular

Syntax

```
tf = istril(A)
```

Description

`tf = istril(A)` returns logical `1` (`true`) if `A` is a lower triangular matrix; otherwise, it returns logical `0` (`false`).

Examples

Test Lower Triangular Matrix

Create a 5-by-5 matrix.

```
D = tril(magic(5))
```

```
D =

    17     0     0     0     0
    23     5     0     0     0
     4     6    13     0     0
    10    12    19    21     0
    11    18    25     2     9
```

Test `D` to see if it is lower triangular.

```
istril(D)
```

```
ans =

     1
```

The result is logical 1 (`true`) because all elements above the main diagonal are zero.

Test Matrix of Zeros

Create a 5-by-5 matrix of zeros.

```
Z = zeros(5);
```

Test `Z` to see if it is lower triangular.

```
istril(Z)
```

```
ans =

     1
```

The result is logical 1 (`true`) because a lower triangular matrix can have any number of zeros on its main diagonal.

Input Arguments

A — Input array

numeric array

Input array, specified as a numeric array. `istril` returns logical `0` (`false`) if `A` has more than two dimensions.

Data Types: `single | double`
Complex Number Support: Yes

More About

Lower Triangular Matrix

A matrix is lower triangular if all elements above the main diagonal are zero. Any number of the elements on the main diagonal can also be zero.

For example, the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -2 & -2 & 1 & 0 \\ -3 & -3 & -3 & 1 \end{pmatrix}$$

is lower triangular. A diagonal matrix is both upper and lower triangular.

Tips

- Use the `tril` function to produce lower triangular matrices for which `istril` returns logical `1` (`true`).
- The functions `isdiag`, `istriu`, and `istril` are special cases of the function `isbanded`, which can perform all of the same tests with suitably defined upper and lower bandwidths. For example, `istril(A) == isbanded(A,size(A,1),0)`.

See Also

`diag` | `isdiag` | `istriu` | `tril` | `triu`

Introduced in R2014a

istriu

Determine if matrix is upper triangular

Syntax

```
tf = istriu(A)
```

Description

`tf = istriu(A)` returns logical `1` (`true`) if `A` is an upper triangular matrix; otherwise, it returns logical `0` (`false`).

Examples

Test Upper Triangular Matrix

Create a 5-by-5 matrix.

```
A = triu(magic(5))
```

```
A =

    17    24     1     8    15
     0     5     7    14    16
     0     0    13    20    22
     0     0     0    21     3
     0     0     0     0     9
```

Test `A` to see if it is upper triangular.

```
istriu(A)
```

```
ans =

     1
```

The result is logical `1` (`true`) because all elements below the main diagonal are zero.

Test Matrix of Zeros

Create a 5-by-5 matrix of zeros.

```
Z = zeros(5);
```

Test `Z` to see if it is upper triangular.

```
istriu(Z)
```

```
ans =

     1
```

The result is logical `1` (`true`) because an upper triangular matrix can have any number of zeros on the main diagonal.

Input Arguments

A — Input array

numeric array

Input array, specified as a numeric array. `istriu` returns logical `0` (`false`) if `A` has more than two dimensions.

Data Types: `single | double`
Complex Number Support: Yes

More About

Upper Triangular Matrix

A matrix is upper triangular if all elements below the main diagonal are zero. Any number of the elements on the main diagonal can also be zero.

For example, the matrix

$$A = \begin{pmatrix} 1 & -1 & -1 & -1 \\ 0 & 1 & -2 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is upper triangular. A diagonal matrix is both upper and lower triangular.

Tips

- Use the `triu` function to produce upper triangular matrices for which `istriu` returns logical `1` (`true`).
- The functions `isdiag`, `istriu`, and `istril` are special cases of the function `isbanded`, which can perform all of the same tests with suitably defined upper and lower bandwidths. For example, `istriu(A) == isbanded(A,0,size(A,2))`.

See Also

`diag` | `isdiag` | `istril` | `tril` | `triu`

Introduced in R2014a

isundefined

Find undefined elements in categorical array

Syntax

```
TF = isundefined(A)
```

Description

`TF = isundefined(A)` returns a logical array, `TF`, that indicates which elements in the categorical array, `A`, contain undefined values. `isundefined` returns logical `1` (`true`) for undefined elements; otherwise it returns logical `0` (`false`). The size of `TF` is the same as the size of `A`.

Any elements in `A` without a corresponding category are undefined. Undefined values are similar to `NaN` in numeric arrays.

Examples

Categorical Array with Undefined Values

Create a categorical array, `A`, from numeric values where `1`, `2`, and `3` represent red, green, and blue respectively.

```
A = categorical([4 1; 2 3; 2 1; 3 4; 1 1],1:3,{'red','green','blue'})
```

```
A =

     <undefined>      red
     green            blue
     green            red
     blue             <undefined>
     red              red
```

`A` is a 5-by-2 categorical array with three categories: `red`, `green`, and `blue`. Array elements corresponding to the numeric value `4` in the input array to the `categorical`

function do not have a corresponding category. Therefore, they are undefined in the output categorical array, A.

Find undefined elements in A.

```
TF = isundefined(A)
```

```
TF =

     1     0
     0     0
     0     0
     0     1
     0     0
```

A(1,1) and A(4,2) are undefined.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

See Also

ismember

isunix

Determine if version is for Linux or Mac platforms

Syntax

```
tf = isunix
```

Description

`tf = isunix` returns logical `1` (`true`) if the version of MATLAB is for Linux or Apple Mac OS X platforms. Otherwise, it returns logical `0` (`false`).

See Also

`computer` | `is*` | `ismac` | `ispc` | `isstudent`

Introduced before R2006a

isvalid (serial)

Determine whether serial port objects are valid

Syntax

```
out = isvalid(obj)
```

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a `0` where the elements of the serial port object, `obj` are invalid serial port objects and a `1` where the elements of `obj` are valid serial port objects.

Examples

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM1');
```

`s2` becomes invalid after it is deleted.

```
delete(s2)
```

`isvalid` verifies that `s1` is valid and `s2` is invalid.

```
sarray = [s1 s2];
isvalid(sarray)

ans =
     1     0
```

More About

Tips

`obj` becomes invalid after it is removed from memory with the `delete` function. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command.

See Also

`clear` | `delete`

Introduced before R2006a

isvarname

Determine whether input is valid variable name

Syntax

```
tf = isvarname('str')
isvarname str
```

Description

`tf = isvarname('str')` returns logical `1` (`true`) if the string `str` is a valid MATLAB variable name and logical `0` (`false`) otherwise. A valid variable name is a character string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

MATLAB keywords are not valid variable names. Type the command `iskeyword` with no input arguments to see a list of MATLAB keywords.

`isvarname str` uses the MATLAB command format.

Examples

This variable name is valid:

```
isvarname foo
ans =
   1
```

This one is not because it starts with a number:

```
isvarname 8th_column
ans =
   0
```

If you are building strings from various pieces, place the construction in parentheses.

```
d = date;
```

```
isvarname(['Monday_', d(1:2)])
ans =
   1
```

See Also

`matlab.lang.makeValidName` | `matlab.lang.makeUniqueStrings` | `iskeyword` | `namelengthmax` | `is*`

Introduced before R2006a

isvector

Determine whether input is vector

Syntax

```
isvector(A)
```

Description

`isvector(A)` returns logical `1` (`true`) if size(A) returns `[1 n]` or `[n 1]` with a nonnegative integer value `n`, and logical `0` (`false`) otherwise.

Examples

Test matrix `A` and its row and column vectors:

```
A = rand(5);

isvector(A)
ans =
     0

isvector(A(3, :))
ans =
     1

isvector(A(:, 2))
ans =
     1
```

See Also

`iscolumn` | `isrow` | `ismatrix` | `isscalar` | `isempty` | `isnumeric` | `islogical` | `ischar` | `isa` | `is*`

Introduced before R2006a

isweekend

Determine weekend elements

Syntax

```
tf = isweekend(t)
```

Description

`tf = isweekend(t)` returns an array the same size as `t` containing logical 1 (`true`) where the corresponding element of `t` is a datetime that occurs on a weekend day, and logical 0 (`false`) otherwise. Weekend days are Saturday and Sunday.

Examples

Determine If Date Occurs During Weekend

```
t = datetime(2014,5,2:5,'Format','eee dd-MMM-yyyy')
```

```
t =

   Fri 02-May-2014   Sat 03-May-2014   Sun 04-May-2014   Mon 05-May-2014
```

```
tf = isweekend(t)
```

```
tf =

     0     1     1     0
```

May 3 and May 4, 2014 are days that fall on a weekend.

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

`day` | `isdst`

Introduced in R2014b

i

Imaginary unit

Syntax

```
1j
z = a + bj
z = x + 1j*y
```

Description

`1j` returns the basic imaginary unit. `j` is equivalent to `sqrt(-1)`.

You can use `j` to enter complex numbers. You also can use the character `i` as the imaginary unit. To create a complex number without using `i` and `j`, use the `complex` function.

`z = a + bj` returns a complex numerical constant, `z`.

`z = x + 1j*y` returns a complex array, `z`.

Examples

Complex Scalar

Create a complex scalar and use the character, `j`, without a multiplication sign as a suffix in forming a complex numerical constant.

```
z = 1+2j
```

```
z =

   1.0000 + 2.0000i
```

Complex Vector

Create a complex vector from two 4-by-1 vectors of real numbers.

```
x = [1:4]';
y = [8:-2:2]';

z = x+1j*y
```

```
z =

   1.0000 + 8.0000i
   2.0000 + 6.0000i
   3.0000 + 4.0000i
   4.0000 + 2.0000i
```

`z` is a 4-by-1 complex vector.

Complex Exponential

Create a complex scalar representing a complex vector with radius, `r`, and angle from the origin, `theta`.

```
r = 4;
theta = pi/4;

z = r*exp(1j*theta)
```

```
z =

   2.8284 + 2.8284i
```

Input Arguments

a — Real component of complex scalar

scalar

Real component of a complex scalar, specified as a scalar.

Data Types: `single` | `double`

b — Imaginary component of complex scalar

scalar

Imaginary component of a complex scalar, specified as a scalar.

If `b` is `double`, you can use the character, `j`, without a multiplication sign as a suffix in forming the complex numerical constant.

Example: `7j`

If `b` is `single`, you must use a multiplication sign when forming the complex numerical constant.

Example: `single(7)*j`

Data Types: `single` | `double`

x — Real component of complex array

scalar | vector | matrix | multidimensional array

Real component of a complex array, specified as a scalar, vector, matrix, or mulitdimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

y — Imaginary component of complex array

scalar | vector | matrix | multidimensional array

Imaginary component of a complex array, specified as a scalar, vector, matrix, or mulitdimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

Output Arguments

z — Complex array

scalar | vector | matrix | multidimensional array

Complex array, returned as a scalar, vector, matrix, or multidimensional array.

The size of `z` is the same as the input arguments.

`z` is `single` if at least one input argument is `single`. Otherwise, `z` is `double`.

More About

Tips

- For speed and improved robustness in complex arithmetic, use `1i` and `1j` instead of `i` and `j`.
- Since `j` is a function, it can be overridden and used as a variable. However, it is best to avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.
- Use the `complex` function to create a complex output in the following cases:
  - When the names `i` and `j` might be used for other variables (and do not equal `sqrt(-1)`)
  - When the inputs are not `double` or `single`
  - When the imaginary component is all zeros
- "Complex Numbers"

See Also

`complex` | `conj` | `i` | `imag` | `real`

Introduced before R2006a

javaaddpath

Add entries to dynamic Java class path

Syntax

```
javaaddpath(dpath)
javaaddpath(dpath,'-end')
```

Description

`javaaddpath(dpath)` adds one or more folders or Java Archive (JAR) files to the beginning of the current dynamic class path. Use the dynamic path when developing and debugging your own Java classes.

`javaaddpath(dpath,'-end')` adds files or folders to the end of the path.

Examples

Add Folder to Dynamic Class Path

Display the current dynamic path.

```
javaclasspath('-dynamic')
```

```
  DYNAMIC JAVA PATH

    <empty>
```

The output reflects your configuration.

Add the current folder.

```
javaaddpath(pwd)
```

Display the dynamic path.

```
p = javaclasspath
```

```
p =
    'c:\work\Java'
```

The output reflects your current folder.

Append URL to Dynamic Class Path

```
javaaddpath('http://www.example.com','-end')
p = javaclasspath
```

```
p =
    'c:\work\Java'
    'http://www.example.com'
```

Input Arguments

`dpath` — Folder or JAR file

string | cell array of strings

Folder or JAR file, specified as a string or cell array of strings, to add to the dynamic path. When you add a folder to the path, MATLAB includes all files in that folder as part of the path.

Data Types: `char` | `cell`

Limitations

- MATLAB does not support JAR files generated by the MATLAB Compiler SDK product.

More About

Tips

- MATLAB provides the dynamic path as a convenience for when you develop your own Java classes. Although the dynamic path offers flexibility in changing the path, Java classes on the dynamic path might load more slowly than classes on the static path. Also, classes on the dynamic path might not behave the same as classes on the static path. If your class does not behave as expected, use the static path.

After you develop and debug a Java class, add the class to the static path.

- MATLAB calls the `clear java` command whenever you change the dynamic path. This command clears the definitions of all Java classes defined by files on the dynamic class path, removes all global variables and variables from the base workspace, and removes all compiled scripts, functions, and MEX-functions from memory.
- To add folders to the static path, which MATLAB loads at startup, create a `javaclasspath.txt` file, as described in "Static Path".
- "Bringing Java Classes into MATLAB Workspace"

See Also

`clear` | `javaclasspath` | `javarmpath`

Introduced before R2006a

javaArray

Construct Java array object

Syntax

```
ObjArr = javaArray(PackageName.ClassName,x1,...,xN)
```

Description

`ObjArr = javaArray(PackageName.ClassName,x1,...,xN)` constructs an empty Java array object for objects of the specified PackageName.ClassName class.

Examples

Create 4-By-5 Array

Create 4-by-5 array of `java.lang.Double` type.

```
x1 = 4; x2 = 5;
dblArray = javaArray ('java.lang.Double',x1,x2);
```

Fill in values.

```
for m = 1:x1
    for n = 1:x2
        dblArray(m,n) = java.lang.Double((m*10) + n);
    end
end
```

Display results.

```
dblArray
```

```
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
```

```
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Input Arguments

`PackageName.ClassName` — Name of Java class

string

Name of Java class, including package name, specified as a string.

Data Types: `char`

`x1,...,xN` — Dimensions of the array

integer

Dimensions of the array, specified as integer.

Data Types: `double`

Output Arguments

`ObjArr` — Java array

Java array

Java array with dimensions x1,...,xN.

More About

Java Array Object

A Java array object is an object with Java dimensionality.

Tips

- The array created by `javaArray` is equivalent to the array created by the following Java code:

  ```
  A = new PackageName.ClassName[x1]...[xN];
  ```

- To create an array of primitive Java types, create an array of the equivalent MATLAB type, shown in the Conversion of MATLAB Types to Java Types table. See "Conversion of MATLAB Argument Data".
- "Working with Java Arrays"

See Also

`class` | `isjava` | `javaMethodEDT` | `javaObjectEDT` | `methodsview`

Introduced before R2006a

javachk

Error message based on Java feature support

Syntax

```
MSG = javachk(feature)
javachk(feature,component)
```

Description

`MSG = javachk(feature)` returns a generic error message if the specified Java feature is not available in the current MATLAB session.

`javachk(feature,component)` also names the specified component in the error message.

Examples

Generate Error

```
if isempty(javachk('jvm'))
   scalar = java.lang.Double(5);
end
% Check that JVM is available & JavaFigures are supported
error(javachk('jvm'))
error(javachk('awt'))
```

Generate Error in User-Defined Script

If you write a script, `myFile`, that displays a Java Frame and want it to error gracefully if a frame cannot be displayed, do the following:

```
error(javachk('awt','myFile'));
myFrame = java.awt.Frame;
myFrame.setVisible(1);
```

If the script cannot display a frame, it displays this error:

```
myFile is not supported on this platform.
```

Input Arguments

`feature` — Java feature

`'awt' | 'desktop' | 'jvm' | 'swing'`

Java feature, specified as one of these values:

`'awt'`	UI components in the Java Abstract Window Toolkit (AWT) are available.
`'desktop'`	MATLAB interactive desktop is running.
`'jvm'`	Java Virtual Machine software (JVM™) is running.
`'swing'`	Swing components (Java lightweight UI components in the Java Foundation Classes) are available.

`component` — Identifier

string

Identifier, specified as a string, to display in the error message.

Data Types: `char`

Output Arguments

`MSG` — Error message

structure

Error message, returned as a structure with these fields:

If it is available, `javachk` returns an error structure with empty `message` and `identifier` fields.

`message` — Message

string | empty

Message, specified as a string.

`identifier` — Identifier
string | empty

Identifier, specified as a string.

See Also

`error` | `usejava`

Introduced before R2006a

javaclasspath

Return Java class path or specify dynamic path

Syntax

```
javaclasspath
javaclasspath('-dynamic')
javaclasspath('-static')

dpath = javaclasspath
spath = javaclasspath('-static')
jpath = javaclasspath('-all')

javaclasspath(dpath)
javaclasspath(dpath1,dpath2)

javaclasspath(statusmsg)
```

Description

`javaclasspath` displays the static and dynamic segments of the Java class path.

`javaclasspath('-dynamic')` displays the dynamic path.

`javaclasspath('-static')` displays the static path.

`dpath = javaclasspath` returns the dynamic path, dpath.

`spath = javaclasspath('-static')` returns the static path, spath.

`jpath = javaclasspath('-all')` returns the entire path, `jpath`. The returned cell array contains first the static segment of the path, and then the dynamic segment.

`javaclasspath(dpath)` changes the dynamic path to dpath. Use this syntax to reload Java classes you are actively developing and debugging.

`javaclasspath(dpath1,dpath2)` changes the dynamic path to the concatenation of paths dpath1,dpath2.

`javaclasspath(statusmsg)` enables or disables the display of status messages.

Examples

Modify Path Using Cell Array

Create a cell array with two path values.

```
dpath = {'http://domain.com','http://some.domain.com/jarfile.jar'};
```

Set message flag to display class-loading messages.

```
javaclasspath('-v1')
```

Modify path.

```
javaclasspath(dpath)
```

```
Loading following class path(s) from local file system:
* http://domain.com
* http://some.domain.com/jarfile.jar
```

Display updated dynamic path.

```
javaclasspath('-dynamic')
```

```
  DYNAMIC JAVA PATH

 http://domain.com
 http://some.domain.com/jarfile.jar
```

MATLAB adds folders from `dpath` to the existing path.

Capture Contents of Dynamic Path

Create a cell array, `p`, with the entries of the dynamic path.

```
javaclasspath('-v0') %Suppress display of class-loading messages
p = javaclasspath
```

```
p =

     {}
```

If there are no entries on the dynamic path, MATLAB creates an empty cell array.

Input Arguments

`dpath` — Path entries

string | cell array of strings

Path entries, specified as a string or cell array of strings, to specify for the dynamic path. Converts relative paths to absolute paths.

Example: javaclasspath('http://domain.com')

Data Types: `char` | `cell`

`dpath1,dpath2` — Path entries

string | cell array of strings

Path entries, specified as a string or cell array of strings, concatenated, to specify for the dynamic path.

Data Types: `char` | `cell`

`statusmsg` — Message flag

'-v0' (default) | '-v1'

Message flag, specified as one of these values:

`'-v0'`	Does not display status messages while loading the Java path from the file system.
`'-v1'`	Displays status messages.

Controls status message display from the `javaclasspath`, `javaaddpath`, and `javarmpath` functions.

Output Arguments

`dpath` — Dynamic path entries

cell array of strings

Dynamic path entries for the current path, returned as a cell array of strings. If no path entries are defined, `dpath` is an empty cell array.

`spath` — Static path entries

cell array of strings

Static path entries for the current path, returned as a cell array of strings. If no path entries are defined, `spath` is an empty cell array.

`jpath` — All path entries

cell array of strings

All path entries, returned as a cell array of strings. If no path entries are defined, `jpath` is an empty cell array.

More About

Static Path

The static path is a segment of the Java path which is loaded at the start of each MATLAB session from the MATLAB built-in Java path and the file `javaclasspath.txt`.

The static Java path offers better Java class-loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `javaclasspath.txt` and restart MATLAB. For more information, see "Static Path".

Dynamic Path

The dynamic path is a segment of the Java class path.

MATLAB provides the dynamic path as a convenience for when you develop your own Java classes. You can load the dynamic path any time during a MATLAB session using the `javaclasspath` function. Although the dynamic path offers flexibility in changing the path, Java classes on the dynamic path might load more slowly than classes on the static path. Also, classes on the dynamic path might not behave the same as classes on the static path. If your class does not behave as expected, use the static path.

After you develop and debug a Java class, add the class to the static path.

You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB. For more information, see "Dynamic Path".

Tips

- MATLAB searches the static path before the dynamic path.
- Java classes on the static path should not have dependencies on classes on the dynamic path. Such dependencies produce run-time errors.
- MATLAB calls the `clear java` command whenever you change the dynamic path. This command clears the definitions of all Java classes defined by files on the dynamic class path, removes all global variables and variables from the base workspace, and removes all compiled scripts, functions, and MEX-functions from memory.
- MATLAB displays a warning if you add an entry to the dynamic path that is already specified on the static path.
- "Java Class Path"

See Also

`clear` | `javaaddpath` | `javarmpath`

Introduced before R2006a

matlab.exception.JavaException class

Package: matlab.exception

Capture error information for Java exception

Description

Process information from a `matlab.exception.JavaException` object to handle Java errors thrown from Java methods called from MATLAB. This class is derived from `MException`.

Construction

`e = matlab.exception.JavaException(msgID,errMsg,excObj)` constructs instance e of `matlab.exception.JavaException` class.

Input Arguments

`msgID`

message identifier

`errMsg`

error message string

`excObj`

`java.lang.Throwable` object that caused the exception

Output Arguments

`e`

Instance of `matlab.exception.JavaException` class

Properties

`ExceptionObject`

Java exception object that caused the error.

Tips

- You do not typically construct a `matlab.exception.JavaException` object explicitly. MATLAB automatically constructs a `JavaException` object whenever Java throws an exception. The `JavaException` object wraps the original Java exception.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

```
try
    java.lang.Class.forName('foo');
catch e
    e.message
    if(isa(e, 'matlab.exception.JavaException'))
        ex = e.ExceptionObject;
        assert(isJava(ex));
        ex.printStackTrace;
    end
end
```

More About

- " Capture Information About Exceptions"
- "Throw an Exception"

Introduced in R2012b

javaMethod

Call Java method

Syntax

```
javaMethod(MethodName,JavaObj,x1,...,xN)
javaMethod(StaticMethodName,ClassName,x1,...,xN)
```

Description

`javaMethod(MethodName,JavaObj,x1,...,xN)` calls the method in the class of the Java object array with the signature matching the arguments x1,...,xN.

`javaMethod(StaticMethodName,ClassName,x1,...,xN)` calls the static method in class ClassName.

Examples

Call Method on Java Object

Create a `java.util.Date` object, `myDate`, and change the month to `3`.

```
myDate = java.util.Date;
javaMethod('setMonth',myDate,3);
```

Call Static Method

Call `java.lang.Double` static method, `isNaN`, to test variable `num`.

```
num = 2.2;
if javaMethod('isNaN','java.lang.Double',num)
    disp('This is not a number')
end
```

Since `num` contains a number, no message is displayed.

Call Method Specified at Runtime

This example, searching for a text pattern in a string, uses variables for the pattern and for the search method. These variables could be set at runtime from user input.

Choose method, `startsWith`, and identify pattern, `str`.

```
fnc = 'startsWith';
str = java.lang.String('Four score');
```

Identify text to search.

```
gAddress = java.lang.String('Four score and seven years ago');
```

Search `gAddress`.

```
javaMethod(fnc,gAddress,str)
```

```
ans =
     1
```

`gAddress` starts with the words `Four score`.

Input Arguments

`MethodName` — Name of nonstatic Java method

string

Name of nonstatic Java method, specified as a string.

Data Types: `char`

`JavaObj` — Array

Java object

Array, specified as a Java object of the class containing the method.

`x1,...,xN` — Java method input arguments

any type

Java method input arguments, 1 through N (if any), required by MethodName or StaticMethodName, specified by any type. The argument type is specified by the method argument list.

`StaticMethodName` — Name of static Java method
string

Name of static Java method, specified as a string.

Data Types: `char`

`ClassName` — Name of Java class
string

Name of Java class, specified as a string, containing StaticMethodName.

Data Types: `char`

More About

Tips

- In most cases, use either MATLAB or Java syntax to call methods on Java objects:

  ```
  % MATLAB syntax
  method(object,arg1,...,argn)

  /* Java syntax */
  object.method(arg1,...,argn)
  ```

- Use `javaMethod` to call methods having names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)

  This is the only way you can call such a method in MATLAB. For example, if you have the following function:

  ```
  javaMethod('DataDefinitionAndDataManipulationTransactions',T);
  ```

- Use `javaMethod` when you want to specify the method name as a variable, to be invoked at runtime. When calling a static method, you also can use a variable in place of the class name argument. For example, see “Call Method Specified at Runtime” on page 1-4325.

- “Calling Syntax”

See Also

`import` | `isjava` | `javaArray` | `javaMethodEDT` | `javaObject` | `methods`

Introduced before R2006a

javaMethodEDT

Call Java method from Event Dispatch Thread (EDT)

Syntax

```
javaMethodEDT(MethodName,JavaObj,x1,...,xN)
javaMethodEDT(StaticMethodName,ClassName,x1,...,xN)
```

Description

`javaMethodEDT(MethodName,JavaObj,x1,...,xN)` calls the method in the class of the Java object array with the signature matching the arguments x1,...,xN from the Event Dispatch Thread (EDT)

`javaMethodEDT(StaticMethodName,ClassName,x1,...,xN)` calls the static method in class ClassName.

Examples

Call Method from EDT

Create a `java.util.Vector` object, `v`, and add a `string` element.

```
v = java.util.Vector;
javaMethodEDT('add',v,'string');
```

Input Arguments

`MethodName` — Name of nonstatic Java method

string

Name of nonstatic Java method, specified as a string.

Data Types: `char`

`JavaObj` — Array

Java object

Array, specified as a Java object of the class containing the method.

`x1,...,xN` — Java method input arguments

any type

Java method input arguments, 1 through N (if any), required by MethodName or StaticMethodName, specified by any type. The argument type is specified by the method argument list.

`StaticMethodName` — Name of static Java method

string

Name of static Java method, specified as a string.

Data Types: `char`

`ClassName` — Name of Java class

string

Name of Java class, specified as a string, containing StaticMethodName.

Data Types: `char`

More About

EDT

The EDT is the Event Dispatch Thread, used in Java.

See Also

`import` | `isjava` | `javaMethod` | `javaObjectEDT` | `methods`

Introduced in R2009a

javaObject

Call Java constructor

Syntax

```
JavaObj = javaObject(ClassName,x1,...,xN)
```

Description

`JavaObj = javaObject(ClassName,x1,...,xN)` returns Java object array, `JavaObj`, created by the Java constructor for the class with the argument list matching x1,...,xN.

Examples

Create Java Object

Create a Java object, `strObj`, of class `java.lang.String`.

```
strObj = javaObject('java.lang.String','hello');
```

Input Arguments

`ClassName` — Name of Java class

string

Name of Java class, specified as a string.

Data Types: `char`

`x1,...,xN` — Java constructor input arguments

any type

Java constructor input arguments, 1 through N (if any), required by ClassName, specified by any type. The argument type is specified by the class constructor argument list.

More About

- "Using the javaObjectEDT Function"

See Also

`import` | `javaArray` | `javaMethod` | `javaObjectEDT` | `methods`

Introduced before R2006a

javaObjectEDT

Call Java constructor on Event Dispatch Thread (EDT)

Syntax

```
JavaObj = javaObjectEDT(ClassName,x1,...,xN)
```

Description

`JavaObj = javaObjectEDT(ClassName,x1,...,xN)` returns Java object array, `JavaObj`, created from the EDT by the Java constructor for the class with the signature matching the arguments x1,...,xN.

Examples

Construct Java Object Array from the EDT

```
f = javaObjectEDT('javax.swing.JFrame','New Title');
```

Call Method on Java Object

Create a `JOptionPane` on the EDT.

```
optPane = javaObjectEDT('javax.swing.JOptionPane');
```

Call the `createDialog` method on the EDT.

```
dlg = optPane.createDialog([],'Sample Dialog');
```

Input Arguments

`ClassName` — Name of Java class

string

Name of Java class, specified as a string.

Data Types: `char`

`x1, . . . , xN` — Java constructor input arguments

any type

Java constructor input arguments, 1 through N (if any), required by ClassName, specified by any type. The argument type is specified by the class constructor argument list.

More About

EDT

The EDT is the Event Dispatch Thread, used in Java.

Tips

- MATLAB calls methods on JavaObj from the EDT.
- Static methods on the specified class or Java object run on the MATLAB thread unless called using the `javaMethodEDT` function.

See Also

`import` | `javaMethodEDT` | `javaObject` | `methods`

Introduced in R2009a

javarmpath

Remove entries from dynamic Java class path

Syntax

```
javarmpath(dpath1,...,dpathN)
```

Description

`javarmpath(dpath1,...,dpathN)` removes one or more files or folders from the current dynamic class path.

Examples

Remove Folder from Dynamic Path

In order to preserve the state of the dynamic path on your system, this example first adds folders to the path, then removes one of these folders.

Create a variable that points to the MATLAB examples folder.

```
expath = fullfile(matlabroot,'extern','examples')
```

```
expath =

C:\Program Files\MATLAB\R2012b\extern\examples
```

The path reflects the folder to your MATLAB installation.

Add two folders to the path.

```
javaclasspath({...
   expath,...
   'http://www.example.com'})
javaclasspath('-dynamic')
```

```
  DYNAMIC JAVA PATH
```

```
C:\Program Files\MATLAB\R2012b\extern\examples
http://www.example.com
```

The output displays these new folders on your existing path.

Remove one folder.

```
javarmpath(expath)
javaclasspath('-dynamic')

  DYNAMIC JAVA PATH

 http://www.example.com
```

The path no longer contains the examples folder.

Input Arguments

`dpath1,...,dpathN` — Folders or JAR files

string

Folders or JAR files, specified as strings, to remove from path.

Data Types: `char`

More About

Tips

- MATLAB calls the `clear java` command whenever you change the dynamic path. This command clears the definitions of all Java classes defined by files on the dynamic class path, removes all global variables and variables from the base workspace, and removes all compiled scripts, functions, and MEX-functions from memory.
- “Bringing Java Classes into MATLAB Workspace”

See Also

Functions

`clear` | `javaaddpath` | `javaclasspath`

Introduced before R2006a

join

Merge two tables by matching up rows using key variables

Syntax

```
C = join(A,B)
C = join(A,B,Name,Value)
[C,ib] = join(___)
```

Description

`C = join(A,B)` merges tables `A` and `B` by matching up rows using all the variables with the same name as key variables. The key values must be common to both `A` and `B`, except for order.

`join` retains all the variables from `A` and appends the corresponding contents from the nonkey variables of `B`.

`C = join(A,B,Name,Value)` joins the tables with additional options specified by one or more Name,Value pair arguments.

For example, you can specify the variables to use as key variables.

`[C,ib] = join(___)` also returns an index vector, ib, such that each element of `ib` identifies the row in `B` that corresponds to that row in the output table, `C`. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Append Values from One Table to Another

Create a table, `A`.

```
A = table({'John','Jane','Jim','Jerry','Jill'}',[1;2;1;2;1],...
    'VariableNames',{'Employee' 'Department'})
```

```
A =
```

```
    Employee    Department
    ________    __________

    'John'      1
    'Jane'      2
    'Jim'       1
    'Jerry'     2
    'Jill'      1
```

Create a table, `B`, with a variable in common with `A`.

```
B = table([1 2]',{'Mary' 'Mike'}',...
    'VariableNames',{'Department' 'Manager'})
```

```
B =

    Department    Manager
    __________    _______

    1             'Mary'
    2             'Mike'
```

Create a new table, `C`, containing data from tables `A` and `B`. Use the `join` function to repeat and append `Manager` data from table `B` to the data from table `A` based on the key variable, `Department`.

```
C = join(A,B)
```

```
C =

    Employee    Department    Manager
    ________    __________    _______

    'John'      1             'Mary'
    'Jane'      2             'Mike'
    'Jim'       1             'Mary'
    'Jerry'     2             'Mike'
    'Jill'      1             'Mary'
```

Merge Tables with One Variable in Common

Create a table, `A`.

```
A = table([5;12;23;2;6],...
```

```
    {'cereal';'pizza';'salmon';'cookies';'pizza'},...
    'VariableNames',{'Age','FavoriteFood'},...
    'RowNames',{'Amy','Bobby','Holly','Harry','Sally'})
```

```
A =

              Age    FavoriteFood
              ___    ____________

    Amy        5     'cereal'
    Bobby     12     'pizza'
    Holly     23     'salmon'
    Harry      2     'cookies'
    Sally      6     'pizza'
```

Create a table, `B`, with one variable in common with `A`.

```
B = table({'cereal';'cookies';'pizza';'salmon';'cake'},...
    [110;160;140;367;243],...
    {'B';'D';'B-';'A';'C-'},...
    'VariableNames',{'FavoriteFood','Calories','NutritionGrade'})
```

```
B =

    FavoriteFood    Calories    NutritionGrade
    ____________    ________    ______________

    'cereal'        110         'B'
    'cookies'       160         'D'
    'pizza'         140         'B-'
    'salmon'        367         'A'
    'cake'          243         'C-'
```

Create a new table, `C`, with data from tables `A` and `B`. Use `FavoriteFood` as a key variable to the `join` function.

```
C = join(A,B)
```

```
C =

              Age    FavoriteFood    Calories    NutritionGrade
              ___    ____________    ________    ______________

    Amy        5     'cereal'        110         'B'
    Bobby     12     'pizza'         140         'B-'
```

```
    Holly    23     'salmon'        367         'A'
    Harry     2     'cookies'       160         'D'
    Sally     6     'pizza'         140         'B-'
```

Table `C` does not include information from the last row of table `B` about `'cake'` since there is no corresponding entry in table `A`.

Merge Tables by Specifying One Key Variable

Create a table, `A`.

```
A = table([10;4;2;3;7],[5;4;9;6;1],[10;3;8;8;4])
```

```
A =

    Var1    Var2    Var3
    ____    ____    ____

    10      5       10
     4      4        3
     2      9        8
     3      6        8
     7      1        4
```

Create a table, `B`, giving `Var2` the same contents as `Var2` from table `A`.

```
B = table([6;1;1;6;8],[5;4;9;6;1])
```

```
B =

    Var1    Var2
    ____    ____

    6       5
    1       4
    1       9
    6       6
    8       1
```

Create a new table, `C`, containing data from tables `A` and `B`. Use `Var1` in table `A` and `Var2` in table `B` as key variables to the `join` function.

```
C = join(A,B,'Keys','Var2')
```

```
C =
```

```
    Var1_A    Var2    Var3    Var1_B
    ______    ____    ____    ______

    10         5       10       6
     4         4        3       1
     2         9        8       1
     3         6        8       6
     7         1        4       8
```

`join` adds a unique suffix to the nonkey variable `Var1` to distinguish the data from tables `A` and `B`.

Keep One Copy of Nonkey Variables

Create a new table with data from tables `A` and `B`. If any nonkey variables have the same name in both tables, keep only the copy from table `A`.

Create a table, `A`.

```
A = table([10;4;2;3;7],[5;4;9;6;1])
```

```
A =

    Var1    Var2
    ____    ____

    10       5
     4       4
     2       9
     3       6
     7       1
```

Create a table, `B`, giving `Var2` the same contents as `Var2` from table `A`.

```
B = table([6;1;1;6;8],[5;4;9;6;1],[10;3;8;8;4])
```

```
B =

    Var1    Var2    Var3
    ____    ____    ____

    6       5       10
    1       4        3
```

```
    1        9        8
    6        6        8
    8        1        4
```

Create a new table, `C`, with data from tables `A` and `B`. Use `Var2` as a key variable to the `join` function and keep only the copy of `Var1` from table `A`.

```
C = join(A,B,'Keys','Var2','KeepOneCopy','Var1')
```

```
C =

    Var1    Var2    Var3
    ____    ____    ____

    10      5       10
     4      4        3
     2      9        8
     3      6        8
     7      1        4
```

`C` does not contain the `Var1` data from table `B`.

Merge Tables Using Row Names as Keys

Create a table, `A`.

```
A = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
    'VariableNames',{'Gender' 'Age'},...
    'RowNames',{'Smith' 'Johnson' 'Williams' 'Jones' 'Brown'})
```

```
A =

                Gender    Age
                ______    ___

    Smith       M         38
    Johnson     M         43
    Williams    F         38
    Jones       F         40
    Brown       F         49
```

Create a table, `B`, such that there is a one-to-one correspondence between the rows of `A` and the rows of `B`.

```
B = table([64;69;67;71;64],...
```

```
    [119;163;133;176;131],...
    [122 80; 109 77; 117 75; 124 93; 125 83],...
    'VariableNames',{'Height' 'Weight' 'BloodPressure'},...
    'RowNames',{'Brown' 'Johnson' 'Jones' 'Smith' 'Williams'})
```

```
B =

                Height    Weight     BloodPressure
                ______    ______    _______________

    Brown       64        119       122          80
    Johnson     69        163       109          77
    Jones       67        133       117          75
    Smith       71        176       124          93
    Williams    64        131       125          83
```

Create a new table, `C`, with data from tables `A` and `B`. Use the row names as keys to the `join` function.

```
C = join(A,B,'Keys','RowNames')
```

```
C =

                Gender    Age    Height    Weight     BloodPressure
                ______    ___    ______    ______    _______________

    Smith       M         38     71        176       124          93
    Johnson     M         43     69        163       109          77
    Williams    F         38     64        131       125          83
    Jones       F         40     67        133       117          75
    Brown       F         49     64        119       122          80
```

The rows of `C` are in the same order as `A`.

Merge Tables Using Left and Right Keys

Create a table, `A`.

```
A = table([10;4;2;3;7],[5;4;9;6;1],[10;3;8;8;4])
```

```
A =

    Var1    Var2    Var3
    ____    ____    ____
```

```
    10       5        10
     4       4         3
     2       9         8
     3       6         8
     7       1         4
```

Create a table, `B`, giving `Var2` the same contents as `Var1` from table `A`, except for order.

```
B = table([6;1;1;6;8],[2;3;4;7;10])
```

```
B =

    Var1    Var2
    ____    ____

    6        2
    1        3
    1        4
    6        7
    8       10
```

Create a new table, `C`, containing data from tables `A` and `B`. Use `Var1` from table `A` with `Var2` from table `B` as key variables to the `join` function.

```
[C,ib] = join(A,B,'LeftKeys',1,'RightKeys',2)
```

```
C =

    Var1_A    Var2    Var3    Var1_B
    ______    ____    ____    ______

    10         5      10       8
     4         4       3       1
     2         9       8       6
     3         6       8       1
     7         1       4       6


ib =

     5
     3
     1
     2
     4
```

`C` is the horizontal concatenation of `A` and `B(ib,2)`.

Input Arguments

`A,B` — Input tables

tables

Input tables, specified as tables. For all key variables, each row of `A` must match exactly one row in `B`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Keys',2` uses the second variable in `A` and the second variable in `B` as key variables.

`'Keys'` — Variables to use as keys

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector | `'RowNames'`

Variables to use as keys, specified as the comma-separated pair consisting of `'Keys'` and a positive integer, vector of positive integers, variable name, cell array of variable names, logical vector, or `'RowNames'`.

If you specify the string `'RowNames'`, then `join` uses the row names of `A` and row names of `B` as keys. In this case, there must be a row in `B` for every row in `A`.

You cannot use the `'Keys'` name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

Example: `'Keys',[1 3]` uses the first and third variables from `A` and `B` as a key variables.

`'LeftKeys'` — Variables to use as keys in `A`

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys in A, specified as the comma-separated pair consisting of 'LeftKeys' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You must use the 'LeftKeys' name-value pair argument in conjunction with the 'RightKeys' name-value pair argument. 'LeftKeys' and 'RightKeys' both must specify the same number of key variables. join pairs key values based on their order.

Example: 'LeftKeys',1 uses only the first variable in A as a key variable.

'RightKeys' — Variables to use as keys in B

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys in B, specified as the comma-separated pair consisting of 'RightKeys' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You must use the 'RightKeys' name-value pair argument in conjunction with the 'LeftKeys' name-value pair argument. 'LeftKeys' and 'RightKeys' both must specify the same number of key variables. join pairs values in A and B based on their order.

Example: 'RightKeys',3 uses only the third variable in B as a key variable.

'LeftVariables' — Variables from A to include in C

positive integer | vector of positive integers | variable name | cell array containing one or more variable names | logical vector

Variables from A to include in C, specified as the comma-separated pair consisting of 'LeftVariables' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You can use 'LeftVariables'to include or exclude key variables, as well as nonkey variables from the output, C.

By default, join includes all variables from A.

'RightVariables' — Variables from B to include in C

positive integer | vector of positive integers | variable name | cell array containing one or more variable names | logical vector

Variables from B to include in C, specified as the comma-separated pair consisting of 'RightVariables' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You can use 'RightVariables' to include or exclude key variables, as well as nonkey variables from the output, C.

By default, join includes all the variables from B except the key variables.

'KeepOneCopy' — Variables for which join retains only the copy from A
variable name | cell array containing one or more variable names

Variables for which join retains only the copy from A, specified as the comma-separated pair consisting of 'KeepOneCopy' and a variable name or a cell array containing one or more variable names.

Key variables appear once in C, but if nonkey variables with identical names occur in A and B, then join retains both copies in C by default. You must use the 'KeepOneCopy' name-value pair to retain only the copy from A.

Example: 'KeepOneCopy',Var2 keeps only the copy from A of the nonkey variable Var2.

Output Arguments

C — Merged data from A and B
table

Merged data from A and B, returned as a table. The table, C, contains one row for each row in A, appearing in the same order. If A and B contain nonkey variables with the same name, join adds a unique suffix to the corresponding variable names in C, unless you specify the 'KeepOneCopy' name-value pair argument.

join creates C by horizontally concatenating A(:,LeftVars) and B(ib,RightVars). By default, LeftVars is all the variables of A, and RightVars is all the nonkey variables from B. Otherwise, LefttVars consists of the variables specified by the 'LeftVariables' name-value pair argument, and RightVars consists of the variables specified by the 'RightVariables' name-value pair argument.

You can store additional metadata such as descriptions, variable units, variable names, and row names in the output table, C. For more information, see Table Properties.

`ib` — Index to `B`

column vector

Index to `B`, returned as a column vector. Each element of `ib` identifies the row in `B` that corresponds to that row in the output table, `C`.

More About

Key Variable

Variable used to match and combine data between the input tables, `A` and `B`.

Key Value

Entry in a key variable of `A`.

Algorithms

The `join` function first finds one or more key variables. Then, `join` uses the key variables to find the row in input table `B` that matches each row in input table `A`, and combines those rows to create a row in output table `C`.

- If there is a one-to-one mapping between key values in `A` and `B`, then `join` sorts the data in `B` and appends it to table `A`.
- If there is a many-to-one mapping between key values in `A` and `B`, then `join` sorts and repeats the data in `B` before appending it to table `A`.
- If there is data in a key variable of `B` that does not map to a key value in `A`, then `join` does not include that data in the output table, `C`.

See Also

`innerjoin` | `outerjoin`

juliandate

Convert MATLAB datetime to Julian date

Syntax

```
d = juliandate(t)
d = juliandate(t,dateType)
```

Description

`d = juliandate(t)` returns a `double` array containing "Julian date" on page 1-4350 equivalent to the datetime values in `t`. If `t` has a time zone, then `juliandate` computes `d` with respect to UTC. `juliandate` ignores leap seconds unless the time zone of `t` is `UTCLeapSeconds`.

`d = juliandate(t,dateType)` returns the type of Julian dates specified by `dateType`. For example, you can convert datetime values to modified Julian dates.

Examples

Convert Datetime Array to Julian Dates

Create a `datetime` array. Then, convert the dates to Julian dates.

```
t = datetime('today') + calmonths(1:3)
```

```
t =

   23-Mar-2015   23-Apr-2015   23-May-2015
```

```
d = juliandate(t)
```

```
d =
```

```
1.0e+06 *

 2.4571    2.4571    2.4572
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`dateType` — Type of Julian date values

`'juliandate'` (default) | `'modifiedjuliandate'`

Type of Julian date values, specified as either `'juliandate'` or `'modifiedjuliandate'`.

- If `dateType` is `'juliandate'`, then `juliandate` converts the datetime values in `t` to the equivalent Julian dates. A Julian date is the number of days and fractional days since noon on November 24, 4712 BCE in the proleptic Gregorian calendar, or January 1, 4713 BCE in the proleptic Julian calendar.
- If `dateType` is `'modifiedjuliandate'`, then `juliandate` converts the datetime values in `t` to the equivalent modified Julian dates. A modified Julian date is the number of days and fractional days since November 17, 1858 00:00:00.

More About

Julian date

A Julian date is the number of days and fractional days since noon on November 24, 4712 BCE in the proleptic Gregorian calendar, or January 1, 4713 BCE in the proleptic Julian calendar.

See Also

`datenum` | `datetime` | `exceltime` | `posixtime` | `yyyymmdd`

Introduced in R2014b

keyboard

Input from keyboard

Syntax

```
keyboard
```

Description

`keyboard` , when placed in a program `.m` file, stops execution of the file and gives control to the keyboard. The special status is indicated by a `K` appearing before the prompt. You can examine or change variables; all MATLAB commands are valid. This keyboard mode is useful for debugging your functions.

To terminate the keyboard mode, type `dbcont`, and then press **Enter**. To terminate keyboard mode and exit the function, type `dbquit`, and press **Enter**.

See Also

`dbstop` | `input` | `quit` | `pause` | `dbcont`

Introduced before R2006a

keys

Class: containers.Map
Package: containers

Identify keys of `containers.Map` object

Syntax

```
keySet = keys(mapObj)
```

Description

`keySet = keys(mapObj)` returns cell array `keySet`, which contains all of the keys in `mapObj`.

Input Arguments

`mapObj`

Object of class `containers.Map`.

Output Arguments

`keySet`

1-by-`n` cell array, where `n` is the number of keys in `mapObj`.

Examples

Get the Keys in a Map

Create a map, and view the keys in the map:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
mapObj = containers.Map(myKeys,myValues);

keySet = keys(mapObj)
```

This code returns 1-by-3 cell array `keySet`:

```
keySet =
    'a'    'b'    'c'
```

See Also

`isKey` | `values` | `containers.Map` | `remove`

kron

Kronecker tensor product

Syntax

```
K = kron(A,B)
```

Description

`K = kron(A,B)` returns the Kronecker tensor product of matrices `A` and `B`. If `A` is an `m`-by-`n` matrix and `B` is a `p`-by-`q` matrix, then `kron(A,B)` is an `m*p`-by-`n*q` matrix formed by taking all possible products between the elements of `A` and the matrix `B`.

Examples

Block Diagonal Matrix

Create a block diagonal matrix.

Create a 4-by-4 identity matrix and a 2-by-2 matrix that you want to be repeated along the diagonal.

```
A = eye(4);
B = [1 -1;-1 1];
```

Use `kron` to find the Kronecker tensor product.

```
K = kron(A,B)
```

```
K =

     1    -1     0     0     0     0     0     0
    -1     1     0     0     0     0     0     0
     0     0     1    -1     0     0     0     0
     0     0    -1     1     0     0     0     0
     0     0     0     0     1    -1     0     0
```

```
     0     0     0     0    -1     1     0     0
     0     0     0     0     0     0     1    -1
     0     0     0     0     0     0    -1     1
```

The result is an 8-by-8 block diagonal matrix.

Repeat Matrix Elements

Expand the size of a matrix by repeating elements.

Create a 2-by-2 matrix of ones and a 2-by-3 matrix whose elements you want to repeat.

```
A = [1 2 3; 4 5 6];
B = ones(2);
```

Calculate the Kronecker tensor product using `kron`.

```
K = kron(A,B)
```

```
K =

     1     1     2     2     3     3
     1     1     2     2     3     3
     4     4     5     5     6     6
     4     4     5     5     6     6
```

The result is a 4-by-6 block matrix.

Sparse Laplacian Operator Matrix

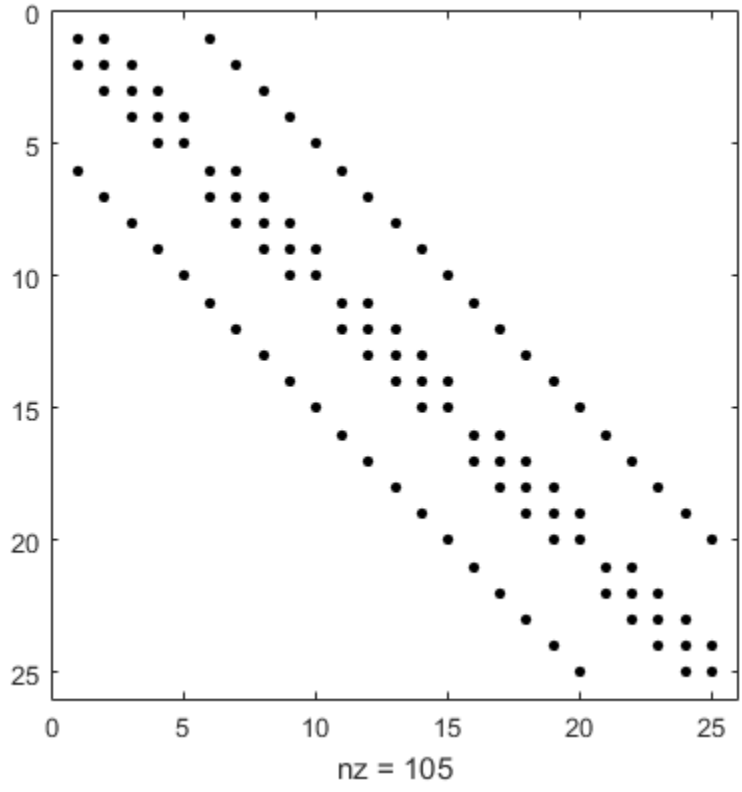
This example visualizes a sparse Laplacian operator matrix.

The matrix representation of the discrete Laplacian operator on a two-dimensional, `n`-by-`n` grid is a `n*n`-by- `n*n` sparse matrix. There are at most five nonzero elements in each row or column. You can generate the matrix as the Kronecker product of one-dimensional difference operators. In this example `n = 5`.

```
n = 5;
I = speye(n,n);
E = sparse(2:n,1:n-1,1,n,n);
D = E+E'-2*I;
A = kron(D,I)+kron(I,D);
```

Visualize the sparsity pattern with `spy`.

```
spy(A,'k')
```



Input Arguments

A,B — Input matrices

scalars | vectors | matrices

Input matrices, specified as scalars, vectors, or matrices. If either `A` or `B` is sparse, then `kron` multiplies only nonzero elements and the result is also sparse.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Complex Number Support: Yes

More About

Kronecker Tensor Product

If A is an m-by-n matrix and B is a p-by-q matrix, then the Kronecker tensor product of A and B is a large matrix formed by multiplying B by each element of A

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}.$$

For example, two simple 2-by-2 matrices produce

$$A = \begin{bmatrix} 1 & -2 \\ -1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & -3 \\ 2 & 3 \end{bmatrix}$$

$$A \otimes B = \begin{bmatrix} 1\cdot 4 & 1\cdot -3 & -2\cdot 4 & -2\cdot -3 \\ 1\cdot 2 & 1\cdot 3 & -2\cdot 2 & -2\cdot 3 \\ -1\cdot 4 & -1\cdot -3 & 0\cdot 4 & 0\cdot -3 \\ -1\cdot 2 & -1\cdot 3 & 0\cdot 2 & 0\cdot 3 \end{bmatrix} = \begin{bmatrix} 4 & -3 & -8 & 6 \\ 2 & 3 & -4 & -6 \\ -4 & 3 & 0 & 0 \\ -2 & -3 & 0 & 0 \end{bmatrix}.$$

See Also

cross | dot | hankel | toeplitz

Introduced before R2006a

Using KeyValueStore Objects

Store key-value pairs for use with mapreduce

The `mapreduce` function automatically creates a `KeyValueStore` object during execution and uses it to store key-value pairs added by the map and reduce functions. Although you never need to explicitly create a `KeyValueStore` object to use `mapreduce`, you do need to use the `add` and `addmulti` object functions to interact with this object in the map and reduce functions.

Examples

Add Key-Value Pair to KeyValueStore in Map Function

The following map function uses the `add` function to add key-value pairs one at a time to an intermediate `KeyValueStore` object (named `intermKVStore`).

```
function MeanDistMapFun(data, info, intermKVStore)
    distances = data.Distance(~isnan(data.Distance));
    sumLenKey = 'sumAndLength';
    sumLenValue = [sum(distances), length(distances)];
    add(intermKVStore, sumLenKey, sumLenValue);
end
```

Add Multiple Key-Value Pairs to KeyValueStore in Map Function

The following map function uses `addmulti` to add several key-value pairs to an intermediate `KeyValueStore` object (named `intermKVStore`). Note that this map function collects multiple keys in the `intermKeys` variable, and multiple values in the `intermVals` variable. This prepares a single call to `addmulti` to add all of the key-value pairs at once. It is a best practice to use a single call to `addmulti` rather than using `add` in a loop.

```
function meanArrivalDelayByDayMapper(data, ~, intermKVStore)
% Mapper function for the MeanByGroupMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% Data is an n-by-2 table: first column is the DayOfWeek and the second
% is the ArrDelay. Remove missing values first.
delays = data.ArrDelay;
```

```
day = data.DayOfWeek;
notNaN =~isnan(delays);
day = day(notNaN);
delays = delays(notNaN);

% find the unique days in this chunk
[intermKeys,~,idx] = unique(day, 'stable');

% group delays by idx and apply @grpstatsfun function to each group
intermVals = accumarray(idx,delays,size(intermKeys),@countsum);
addmulti(intermKVStore,intermKeys,intermVals);

function out = countsum(x)
n = length(x); % count
s = sum(x); % mean
out = {[n, s]};
```

Object Functions

`add` `addmulti`

Create Object

The `mapreduce` function automatically creates `KeyValueStore` objects during execution.

See Also

`mapreduce`

More About

- “Getting Started with MapReduce”

lastDirectory

Class: Tiff

Determine if current IFD is last in file

Syntax

```
tf = lastDirectory(tiffobj)
```

Description

`tf = lastDirectory(tiffobj)` returns `true` if the current image file directory (IFD) is the last IFD in the TIFF file; otherwise, `false`. If the file contains only one image, the current IFD is the last.

Examples

Determine if Current Directory is the Last Directory

Open a `Tiff` object and determine if the current directory is the last directory in the file.

```
t = Tiff('example.tif','r');
tf = lastDirectory(t)
```

```
tf =

     0
```

The current directory is not the last directory in the file.

Close the `Tiff` object.

```
close(t)
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFLastDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.setDirectory`

Chart Line Properties

Control chart line appearance and behavior

Chart line properties control the appearance and behavior of a chart line object. By changing property values, you can modify certain aspects of the line.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = plot(1:10);
c = h.Color;
h.Color = 'red';
```

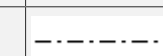
If you are using an earlier release, use the `get` and `set` functions instead.

Line

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the `Color` as `'none'`, then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the chart line object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

`6` (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Data

`XData` — x values

vector

x values, specified as a vector.

- For 2-D line plots, if you do not specify the *x* values, then MATLAB uses the indices of `YData` as the *x* values for the plot. `XData` and `YData` must have equal lengths.

- For 3-D line plots, if you do not specify the *x* values, then MATLAB uses the indices of `ZData` as the *x* values for the plot. `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1:10]`

`YData` — y values

vector

y values, specified as a vector. For 2-D line plots, `XData` and `YData` must have equal lengths. For 3-D line plots, `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1:10]`

`ZData` — z values

vector

z values for the 3-D line plot, specified as a vector. `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1:10]`

`XDataSource` — Variable linked to `XData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZDataSource` — Variable linked to `ZData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `ZData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

`XDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData` (or `ZData` for 3-D plots).
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

Visibility

`Visible` — Visibility of chart line

`'on'` (default) | `'off'`

Visibility of chart line, specified as one of these values:

- `'on'` — Display the chart line.
- `'off'` — Hide the chart line without deleting it. You still can access the properties of an invisible chart line object.

`Clipping` — Clipping of chart line to axes limits

`'on'` (default) | `'off'`

Clipping of chart line to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the chart line that are outside the axes limits.
- `'off'` — Display the entire chart line, even if parts of it appear outside the axes limits. Parts of the chart line might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the chart line that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.

- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object
`'line'`

Type of graphics object, returned as `'line'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — User-specified tag
`''` (default) | string

Tag to associate with the chart line, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with chart line
`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the chart line object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the chart line.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the chart line object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the chart line from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:

- `'on'` — Include the chart line object in the legend as one entry (default).
- `'off'` — Do not include the chart line object in the legend.
- `'children'` — Include only children of the chart line object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of chart line

axes object | group object | transform object

Parent of chart line, specified as an axes, group, or transform object.

`Children` — Children of chart line

empty `GraphicsPlaceholder` array

The chart line has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of chart line object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The chart line object handle is always visible.
- `'off'` — The chart line object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The chart line object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the chart line at the command-line, but allows callback functions to access it.

If the chart line object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle

properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the chart line. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The chart line object — You can access properties of the chart line object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the chart line. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the chart line when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the chart line.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the chart line that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the chart line responds to the click or if an ancestor does.

- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'` and you can click a part of the chart line that has no color. The HitTest property determines if the chart line responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the chart line passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the chart line. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the chart line that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the chart line object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the ButtonDownFcn callback of the chart line is the running callback, then the Interruptible property determines if it another callback can interrupt it:

- 'on' — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a drawnow, figure, getframe, waitfor, or pause command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- 'off' — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

BusyAction — Callback queuing
'queue' (default) | 'cancel'

Callback queuing specified as 'queue' or 'cancel'. The BusyAction property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the ButtonDownFcn callback of the chart line tries to interrupt a running callback that cannot be interrupted, then the BusyAction property determines if it is discarded or put in the queue. Specify the BusyAction property as one of these values:

- 'queue' — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.

- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the chart line. Setting the `CreateFcn` property on an existing chart line has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during chart line creation. MATLAB executes the callback after creating the chart line and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The chart line object — You can access properties of the chart line object from within the callback function. You also can access the chart line object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the chart line. MATLAB executes the callback before destroying the chart line so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The chart line object — You can access properties of the chart line object from within the callback function. You also can access the chart line object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of chart line

`'off'` (default) | `'on'`

Deletion status of chart line, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the chart line begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the chart line no longer exists.

Check the value of the BeingDeleted property to verify that the chart line is not about to be deleted before querying or modifying it.

See Also

`loglog` | `plot` | `plot3` | `plotyy` | `semilogx` | `semilogy`

More About

- "Access Property Values"

- “Graphics Object Properties”

lasterr

Last error message

Note: `lasterr` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See " Capture Information About Exceptions" in the Programming Fundamentals documentation.

Syntax

```
msgstr = lasterr
[msgstr, msgid] = lasterr
lasterr('new_msgstr')
lasterr('new_msgstr', 'new_msgid')
[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')
```

Description

`msgstr = lasterr` returns the last error message generated by the MATLAB software.

`[msgstr, msgid] = lasterr` returns the last error in `msgstr` and its message identifier in `msgid`. If the error was not defined with an identifier, `lasterr` returns an empty string for `msgid`. See "Message Identifiers" in the MATLAB Programming Fundamentals documentation for more information on the `msgid` argument and how to use it.

`lasterr('new_msgstr')` sets the last error message to a new string, `new_msgstr`, so that subsequent invocations of `lasterr` return the new error message string. You can also set the last error to an empty string with `lasterr('')`.

`lasterr('new_msgstr', 'new_msgid')` sets the last error message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lasterr` return the new error message and message identifier.

`[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')` returns the last error message and its identifier, also changing these values so that subsequent

invocations of `lasterr` return the message and identifier strings specified by `new_msgstr` and `new_msgid` respectively.

Examples

Example 1

Here is a function that examines the `lasterr` string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply:

```
function matrix_multiply(A, B)
try
   A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
   else
      if(strfind(errmsg, 'not defined for variables of class'))
         disp('** Both arguments must be double matrices')
      end
   end
end
```

If you call this function with matrices that are incompatible for matrix multiplication (e.g., the column dimension of `A` is not equal to the row dimension of `B`), MATLAB catches the error and uses `lasterr` to determine its source:

```
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];

matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

Example 2

Specify a message identifier and error message string with `error`:

```
error('MyToolbox:angleTooLarge', ...
      'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterr` to determine the message identifier and error message string for the failing operation:

```
[errmsg, msgid] = lasterr
errmsg =
   The angle specified must be less than 90 degrees.
msgid =
   MyToolbox:angleTooLarge
```

See Also

`error` | `lasterror` | `rethrow` | `warning` | `lastwarn`

Introduced before R2006a

lasterror

Last error message and related information

Note: `lasterror` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See " Capture Information About Exceptions" in the Programming Fundamentals documentation.

Syntax

```
s = lasterror
s = lasterror(err)
s = lasterror('reset')
```

Description

`s = lasterror` returns a structure `s` containing information about the most recent error issued by the MATLAB software. The return structure contains the following fields:

Fieldname	Description
`message`	Character array containing the text of the error message.
`identifier`	Character array containing the message identifier of the error message. If the last error issued by MATLAB had no message identifier, then the `identifier` field is an empty character array.
`stack`	Structure providing information on the location of the error. The structure has fields `file`, `name`, and `line`, and is the same as the structure returned by the `dbstack` function. If `lasterror` returns no stack information, `stack` is a 0-by-1 structure having the same three fields.

Note The `lasterror` return structure might contain additional fields in future versions of MATLAB.

The fields of the structure returned in `stack` are

Fieldname	Description
`file`	Name of the file in which the function generating the error appears. This field is the empty string if there is no file.
`name`	Name of the function in which the error occurred. If this is the primary function in the file, and the function name differs from the file name, `name` is set to the file name.
`line`	Line number of the file at which the error occurred.

See "Message Identifiers" in the MATLAB Programming Fundamentals documentation for more information on the syntax and usage of message identifiers.

`s = lasterror(err)` sets the last error information to the error message and identifier specified in the structure `err`. Subsequent invocations of `lasterror` return this new error information. The optional return structure `s` contains information on the previous error.

`s = lasterror('reset')` sets the last error information to the default state. In this state, the `message` and `identifier` fields of the return structure are empty strings, and the `stack` field is a 0-by-1 structure.

Examples

Example 1

Save the following MATLAB code in a file called `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.
check_inputs(x)
y = sum(x)/length(x);      % The actual computation

function check_inputs(x)
[m,n] = size(x);
if (~((m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('AVG:NotAVector', 'Input must be a vector.')
end
```

Now run the function. Because this function requires vector input, passing a scalar value to it forces an error. The error occurs in subroutine `check_inputs`:

```
average(200)
Error using average>check_inputs (line 11)
Input must be a vector.

Error in average (line 5)
check_inputs(x)
```

Get the three fields from `lasterror`:

```
err = lasterror
err =
       message: [1x61 char]
    identifier: 'AVG:NotAVector'
         stack: [2x1 struct]
```

Display the text of the error message:

```
msg = err.message
msg =
    Error using average>check_inputs (line 11)
    Input must be a vector.
```

Display the fields containing the `stack` information. `err.stack` is a 2-by-1 structure because it provides information on the failing subroutine `check_inputs` and also the outer, primary function `average`:

```
st1 = err.stack(1,1)
st1 =
    file: 'd:\matlab_test\average.m'
    name: 'check_inputs'
    line: 11

st2 = err.stack(2,1)
st2 =
    file: 'd:\matlab_test\average.m'
    name: 'average'
    line: 5
```

Note As a rule, the name of your primary function should be the same as the name of the file that contains that function. If these names differ, MATLAB uses the file name in the `name` field of the `stack` structure.

Example 2

`lasterror` is often used in conjunction with the `MException.rethrow` function in `try, catch` statements. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(lasterror)
end
```

More About

Tips

MathWorks is gradually transitioning MATLAB error handling to an object-oriented scheme that is based on the `MException` class. Although support for `lasterror` is expected to continue, using the static `MException.last` method of `MException` is preferable.

Warning `lasterror` and `MException.last` are not guaranteed to always return identical results. For example, `MException.last` updates its error status only on uncaught errors, where `lasterror` can update its error status on any error, whether it is caught or not.

See Also

`MException.rethrow` | `MException` | `MException.last` | `assert` | `dbstack` | `error` | `lastwarn` | `try, catch`

Introduced before R2006a

lastwarn

Last warning message

Syntax

```
msgstr = lastwarn
[msgstr, msgid] = lastwarn
lastwarn('new_msgstr')
lastwarn('new_msgstr', 'new_msgid')
[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')
```

Description

`msgstr = lastwarn` returns the last warning message generated by MATLAB, regardless of its display state.

`[msgstr, msgid] = lastwarn` returns the last warning in `msgstr` and its message identifier in `msgid`. If the warning was not defined with an identifier, `lastwarn` returns an empty string for `msgid`.

`lastwarn('new_msgstr')` sets the last warning message to a new string, `new_msgstr`, so that subsequent invocations of `lastwarn` return the new warning message string. You can also set the last warning to an empty string with `lastwarn('')`.

`lastwarn('new_msgstr', 'new_msgid')` sets the last warning message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lastwarn` return the new warning message and message identifier.

`[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')` returns the last warning message and its identifier, also changing these values so that subsequent invocations of `lastwarn` return the message and identifier strings specified by `new_msgstr` and `new_msgid`, respectively.

Examples

Write a short function that generates a warning message. At the start of the function, enable any warnings that have a message identifier called `TestEnv:InvalidInput`:

```
function myfun(p1)
warning on TestEnv:InvalidInput;

exceedMax = find(p1 > 5000);
if any(exceedMax)
    warning('TestEnv:InvalidInput', ...
        '%d values in the "%s" array exceed the maximum.', ...
        length(exceedMax), inputname(1))
end
```

Pass an invalid value to the function:

```
dataIn = magic(10) - 2;

myfun(dataIn)
Warning: 2 values in the "dataIn" array exceed the maximum.
> In myfun at 4
```

Use `lastwarn` to determine the message identifier and error message string for the operation:

```
[warnmsg, msgid] = lastwarn
warnmsg =
   2 values in the "dataIn" array exceed the maximum.
msgid =
   TestEnv:InvalidInput
```

More About

Tips

`lastwarn` does not return warnings that are reported during the parsing of MATLAB commands. (Warning messages that include the failing file name and line number are parse-time warnings.)

See Also

`warning` | `error`

Introduced before R2006a

lcm

Least common multiple

Syntax

```
L = lcm(A,B)
```

Description

`L = lcm(A,B)` returns the least common multiples of the elements of `A` and `B`.

Examples

Least Common Multiples of Double Array and a Scalar

```
A = [5 17; 10 60];
B = 45;
L = lcm(A,B)
```

```
L =

    45   765
    90   180
```

Least Common Multiples of Unsigned Integers

```
A = uint16([255 511 15]);
B = uint16([15 127 1023]);
L = lcm(A,B)
```

```
L =

    255  64897   5115
```

Input Arguments

`A,B` — Input values

scalars, vectors, or arrays of real, positive integer values

Input values, specified as scalars, vectors, or arrays of real, positive integer values. A and B can be any numeric type, and they can be of different types within certain limitations:

- If A or B is of type `single`, then the other can be of type `single` or `double`.
- If A or B belongs to an integer class, then the other must belong to the same class or it must be a `double` scalar value.

A and B must be the same size or one must be a scalar.

Example: `[20 3 13],[10 6 7]`

Example: `int16([100 30 200]),int16([20 15 9])`

Example: `int16([100 30 200]),20`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Output Arguments

L — Least common multiple

real, positive integer values

Least common multiple, returned as an array of real positive integer values. L is the same size as A and B, and it has the same type as A and B. If A and B are of different types, then L is returned as the nondouble type.

See Also

`gcd`

Introduced before R2006a

ldl

Block LDL' factorization for Hermitian indefinite matrices

Syntax

```
L = ldl(A)
[L,D] = ldl(A)
[L,D,P] = ldl(A)
[L,D,p] = ldl(A,'vector')
[U,D,P] = ldl(A,'upper')
[U,D,p] = ldl(A,'upper','vector')
[L,D,P,S] = ldl(A)
[L,D,P,S] = LDL(A,THRESH)
[U,D,p,S] = LDL(A,THRESH,'upper','vector')
```

Description

`L = ldl(A)` returns only the permuted lower triangular matrix `L` as in the two-output form. The permutation information is lost, as is the block diagonal factor `D`. By default, `ldl` references only the diagonal and lower triangle of `A`, and assumes that the upper triangle is the complex conjugate transpose of the lower triangle. Therefore `[L,D,P] = ldl(TRIL(A))` and `[L,D,P] = ldl(A)`both return the exact same factors. Note, this syntax is not valid for sparse `A`.

`[L,D] = ldl(A)` stores a block diagonal matrix `D` and a permuted lower triangular matrix in `L` such that `A = L*D*L'`. The block diagonal matrix `D` has 1-by-1 and 2-by-2 blocks on its diagonal. Note, this syntax is not valid for sparse `A`.

`[L,D,P] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, and permutation matrix `P` such that `P'*A*P = L*D*L'`. This is equivalent to `[L,D,P] = ldl(A,'matrix')`.

`[L,D,p] = ldl(A,'vector')` returns the permutation information as a vector, `p`, instead of a matrix. The `p` output is a row vector such that `A(p,p) = L*D*L'`.

`[U,D,P] = ldl(A,'upper')` references only the diagonal and upper triangle of `A` and assumes that the lower triangle is the complex conjugate transpose of the upper triangle.

This syntax returns a unit upper triangular matrix `U` such that `P'*A*P = U'*D*U` (assuming that `A` is Hermitian, and not just upper triangular). Similarly, `[L,D,P] = ldl(A,'lower')` gives the default behavior.

`[U,D,p] = ldl(A,'upper','vector')` returns the permutation information as a vector, `p`, as does `[L,D,p] = ldl(A,'lower','vector')`. `A` must be a full matrix.

`[L,D,P,S] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, permutation matrix `P`, and scaling matrix `S` such that `P'*S*A*S*P = L*D*L'`. This syntax is only available for real sparse matrices, and only the lower triangle of `A` is referenced. `ldl` uses MA57 for sparse real symmetric `A`.

`[L,D,P,S] = LDL(A,THRESH)` uses `THRESH` as the pivot tolerance in MA57. `THRESH` must be a double scalar lying in the interval `[0, 0.5]`. The default value for `THRESH` is `0.01`. Using smaller values of `THRESH` may give faster factorization times and fewer entries, but may also result in a less stable factorization. This syntax is available only for real sparse matrices.

`[U,D,p,S] = LDL(A,THRESH,'upper','vector')` sets the pivot tolerance and returns upper triangular `U` and permutation vector `p` as described above.

Examples

These examples illustrate the use of the various forms of the `ldl` function, including the one-, two-, and three-output form, and the use of the `vector` and `upper` options. The topics covered are:

Before running any of these examples, you will need to generate the following positive definite and indefinite Hermitian matrices:

```
A = full(delsq(numgrid('L', 10)));
```

```
B = gallery('uniformdata',10,0);
M = [eye(10) B; B' zeros(10)];
```

The structure of M here is very common in optimization and fluid-flow problems, and M is in fact indefinite. Note that the positive definite matrix A must be full, as `ldl` does not accept sparse arguments.

Example 1 — Two-Output Form of ldl

The two-output form of `ldl` returns L and D such that `A-(L*D*L')` is small, L is permuted unit lower triangular, and D is a block 2-by-2 diagonal. Note also that, because A is positive definite, the diagonal of D is all positive:

```
[LA,DA] = ldl(A);
fprintf(1, ...
'The factorization error ||A - LA*DA*LA''|| is %g\n', ...
norm(A - LA*DA*LA'));
neginds = find(diag(DA) < 0)
```

Given a b, solve Ax=b using LA, DA:

```
bA = sum(A,2);
x = LA'\(DA\(LA\bA));
fprintf(...
'The absolute error norm ||x - ones(size(bA))|| is %g\n', ...
norm(x - ones(size(bA))));
```

Example 2 — Three Output Form of ldl

The three output form returns the permutation matrix as well, so that L is in fact unit lower triangular:

```
[Lm, Dm, Pm] = ldl(M);
fprintf(1, ...
'The error norm ||Pm''*M*Pm - Lm*Dm*Lm''|| is %g\n', ...
norm(Pm'*M*Pm - Lm*Dm*Lm'));
fprintf(1, ...
'The difference between Lm and tril(Lm) is %g\n', ...
norm(Lm - tril(Lm)));
```

Given b, solve Mx=b using Lm, Dm, and Pm:

```
bM = sum(M,2);
```

```
x = Pm*(Lm'\(Dm\(Lm\(Pm'*bM))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

Example 3 — The Structure of D

D is a block diagonal matrix with 1-by-1 blocks and 2-by-2 blocks. That makes it a special case of a tridiagonal matrix. When the input matrix is positive definite, D is almost always diagonal (depending on how definite the matrix is). When the matrix is indefinite however, D may be diagonal or it may express the block structure. For example, with A as above, DA is diagonal. But if you shift A just a bit, you end up with an indefinite matrix, and then you can compute a D that has the block structure.

```
figure; spy(DA); title('Structure of D from ldl(A)');
[Las, Das] = ldl(A - 4*eye(size(A)));
figure; spy(Das);
title('Structure of D from ldl(A - 4*eye(size(A)))');
```

Example 4 — Using the 'vector' Option

Like the `lu` function, `ldl` accepts an argument that determines whether the function returns a permutation vector or permutation matrix. `ldl` returns the latter by default. When you select `'vector'`, the function executes faster and uses less memory. For this reason, specifying the `'vector'` option is recommended. Another thing to note is that indexing is typically faster than multiplying for this kind of operation:

```
[Lm, Dm, pm] = ldl(M, 'vector');
fprintf(1, 'The error norm ||M(pm,pm) - Lm*Dm*Lm''|| is %g\n', ...
  norm(M(pm,pm) - Lm*Dm*Lm'));

% Solve a system with this kind of factorization.
clear x;
x(pm,:) = Lm'\(Dm\(Lm\(bM(pm,:))));
fprintf('The absolute error norm ||x - ones(size(b))|| is %g\n', ...
  norm(x - ones(size(bM))));
```

Example 5 — Using the 'upper' Option

Like the `chol` function, `ldl` accepts an argument that determines which triangle of the input matrix is referenced, and also whether `ldl` returns a lower (`L`) or upper (`L'`) triangular factor. For dense matrices, there are no real savings with using the upper triangular version instead of the lower triangular version:

```
Ml = tril(M);
[Lml, Dml, Pml] = ldl(Ml, 'lower'); % 'lower' is default behavior.
fprintf(1, ...
'The difference between Lml and Lm is %g\n', norm(Lml - Lm));
[Umu, Dmu, pmu] = ldl(triu(M), 'upper', 'vector');
fprintf(1, ...
'The difference between Umu and Lm'' is %g\n', norm(Umu - Lm'));

% Solve a system using this factorization.
clear x;
x(pm,:) = Umu\(Dmu\(Umu'\(bM(pmu,:))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

When specifying both the `'upper'` and `'vector'` options, `'upper'` must precede `'vector'` in the argument list.

Example 6 — linsolve and the Hermitian indefinite solver

When using the `linsolve` function, you may experience better performance by exploiting the knowledge that a system has a symmetric matrix. The matrices used in the examples above are a bit small to see this so, for this example, generate a larger matrix. The matrix here is symmetric positive definite, and below we will see that with each bit of knowledge about the matrix, there is a corresponding speedup. That is, the symmetric solver is faster than the general solver while the symmetric positive definite solver is faster than the symmetric solver:

```
Abig = full(delsq(numgrid('L', 30)));
bbig = sum(Abig, 2);
LSopts.POSDEF = false;
LSopts.SYM = false;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.SYM = true;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.POSDEF = true;
tic; linsolve(Abig, bbig, LSopts); toc;
```

More About

Algorithms

`ldl` uses the MA57 routines in the Harwell Subroutine Library (HSL) for real sparse matrices.

References

[1] Ashcraft, C., R.G. Grimes, and J.G. Lewis. “Accurate Symmetric Indefinite Linear Equations Solvers.” *SIAM J. Matrix Anal. Appl.* Vol. 20. Number 2, 1998, pp. 513–561.

[2] Duff, I. S. "MA57 — A new code for the solution of sparse symmetric definite and indefinite systems." Technical Report RAL-TR-2002-024, Rutherford Appleton Laboratory, 2002.

See Also

`chol` | `lu` | `qr`

ldivide, .\

Left array division

Syntax

```
x = B.\A
x = ldivide(B,A)
```

Description

`x = B.\A` divides each element of A by the corresponding element of B.

- If A and B are arrays, then they must be the same size.
- If either A or B is a scalar, then MATLAB expands the scalar value into an appropriately sized array.

`x = ldivide(B,A)` is an alternative way to divide A by B, but is rarely used. It enables operator overloading for classes.

Examples

Divide Two Numeric Arrays

```
A = ones(2, 3);
B = [1 2 3; 4 5 6];
x = B.\A
```

```
x =

    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
```

Divide a Scalar by a Numeric Array

```
C = 2;
D = [1 2 3; 4 5 6];
x = D.\C
```

```
x =

    2.0000    1.0000    0.6667
    0.5000    0.4000    0.3333
```

Input Arguments

A — Numerator

scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix or multidimensional array. If B is an integer data type, then A must be the same integer type or a scalar double.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`
Complex Number Support: Yes

B — Denominator

scalar | vector | matrix | multidimensional array

Denominator, specified as a scalar, vector, matrix or multidimensional array. If A is an integer data type, then B must be the same integer type or a scalar double.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`
Complex Number Support: Yes

Output Arguments

x — Solution

scalar | vector | matrix | multidimensional array

Solution, returned as a scalar, vector, matrix or multidimensional array. If either A or B are integer data types, then x is that same integer data type.

More About

Tips

- MATLAB does not support complex integer division.

- “Array vs. Matrix Operations”
- “Operator Precedence”

See Also

`mldivide` | `mrdivide` | `rdivide`

Introduced before R2006a

le, <=

Determine less than or equal to

Syntax

```
A <= B
le(A,B)
```

Description

`A <= B` returns a logical array with elements set to logical 1 (`true`) where `A` is less than or equal to `B`; otherwise, it returns logical `0` (`false`).

The test compares only the real part of numeric arrays. `le` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`le(A,B)` is an alternate way to execute `A <= B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Find which vector elements are less than or equal to a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are less than or equal to `12`.

```
A <= 12
```

```
ans =

     1     1     0     1     1     1     1     0
```

The result is a vector with values of logical 1 (`true`) where the elements of A satisfy the expression.

Use the vector of logical values as an index to view the values in A that are less than or equal to 12.

```
A(A <= 12)
```

```
ans =

     1    12     7     9    11     2
```

The result is a subset of the elements in A.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values less than or equal to 9 with the value 10.

```
A(A <= 9) = 10
```

```
A =

    16    10    10    13
    10    11    10    10
    10    10    10    12
    10    14    15    10
```

The result is a new matrix whose smallest element is 10.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
```

```
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)

A =

     large       medium      small
     medium      small       large
```

The array has three categories: `'small'`, `'medium'`, and `'large'`.

Find all values less than or equal to the category `'medium'`.

```
A <= 'medium'

ans =

     0     1     1
     1     1     0
```

A value of logical `1` (`true`) indicates a value less than or equal to the category `'medium'`.

Compare the rows of `A`.

```
A(1,:) <= A(2,:)

ans =

     0     0     1
```

The function returns logical `1` (`true`) where the first row has a category value less than or equal to the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are less than or equal to `3`.

```
A(A <= 3)

ans =

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i
```

`le` compares only the real part of the elements in `A`.

Use `abs` to find which elements are within a radius of `3` from the origin.

```
A(abs(A) <= 3)
```

```
ans =

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 - 2.0000i
```

The result has one less element. The element `1.0000 + 3.0000i` is not within a radius of `3` from the origin.

Test Duration Values

Create a `duration` array.

```
d = hours(21:25) + minutes(75)
```

```
d =

   22.25 hrs   23.25 hrs   24.25 hrs   25.25 hrs   26.25 hrs
```

Test the array for elements that are less than or equal to one standard day.

```
d <= 1
```

```
ans =

     1     1     0     0     0
```

Input Arguments

A — Left array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Left array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

B — Right array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Right array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

More About

Tips

- Some floating-point numbers cannot be represented exactly in binary form. This leads to small differences in results that the `<=` operator reflects. For more information, see "Avoiding Common Problems with Floating-Point Arithmetic".
- "Ordinal Categorical Arrays"

See Also

`eq | ge | gt | lt | ne`

Introduced before R2006a

legend

Add legend to graph

Syntax

```
legend('string1',...,'stringN')
legend('string1',...,'stringN','Location',Value)
legend(___,'Orientation',Value)

legend(strings)
legend(strings,Name,Value)

legend(entries,___)

legend('boxon')
legend('boxoff')

legend('hide')
legend('show')
legend('toggle')
legend('off')

legend(ax,___)

h = legend(___)
[h,icons,plots,str] = legend(___)
```

Description

`legend('string1',...,'stringN')` creates a legend in the current axes using the specified strings to label each set of data. The legend shows an icon of the associated object next to each string.

`legend('string1',...,'stringN','Location',Value)` specifies the legend location. Set `Value` to a location string such as `'northoutside'`. For a list of strings, see the Location property.

`legend( ___ ,'Orientation',Value)` specifies the legend orientation. This option can be used with either of the input argument combinations in the previous syntaxes. Set `Value` as either `'vertical'` or `'horizontal'`. For more information, see the Orientation property.

`legend(strings)` creates a legend in the current axes and uses the entries in `strings` to label each set of data. Specify `strings` as either a cell array of strings or a matrix of strings.

`legend(strings,Name,Value)` specifies legend properties. Use this syntax if you want to specify `Name,Value` pairs other than Location and Orientation when creating a legend. For example, `'FontSize',12` sets the font size to 12 points.

`legend(entries, ___ )` displays legend entries for the objects in the graph specified by the vector `entries`. Use this syntax if you want to display legend entries for only some of the objects in the graph. The option, `entries`, can precede any of the input argument combinations in the previous syntaxes.

`legend('boxon')` displays the box around the legend.

`legend('boxoff')` removes the box around the legend.

`legend('hide')` hides the legend associated with the current axes. Hiding the legend does not delete it.

`legend('show')` makes the legend in the current axes visible. If a legend does not exist, then MATLAB creates one using the default strings.

`legend('toggle')` toggles the legend visibility. If a legend does not exist, then MATLAB creates one using the default strings.

`legend('off')` deletes the legend from the current axes.

`legend(ax, ___ )` uses the axes specified by `ax` instead of the current axes (`gca`). The option, `ax`, can precede any of the previous input argument combinations.

`h = legend( ___ )` returns the legend object. Use `h` to query and set properties of the legend after it is created.

Note: Starting in R2014b, the `legend` function returns a legend object. In previous releases it returns an axes object.

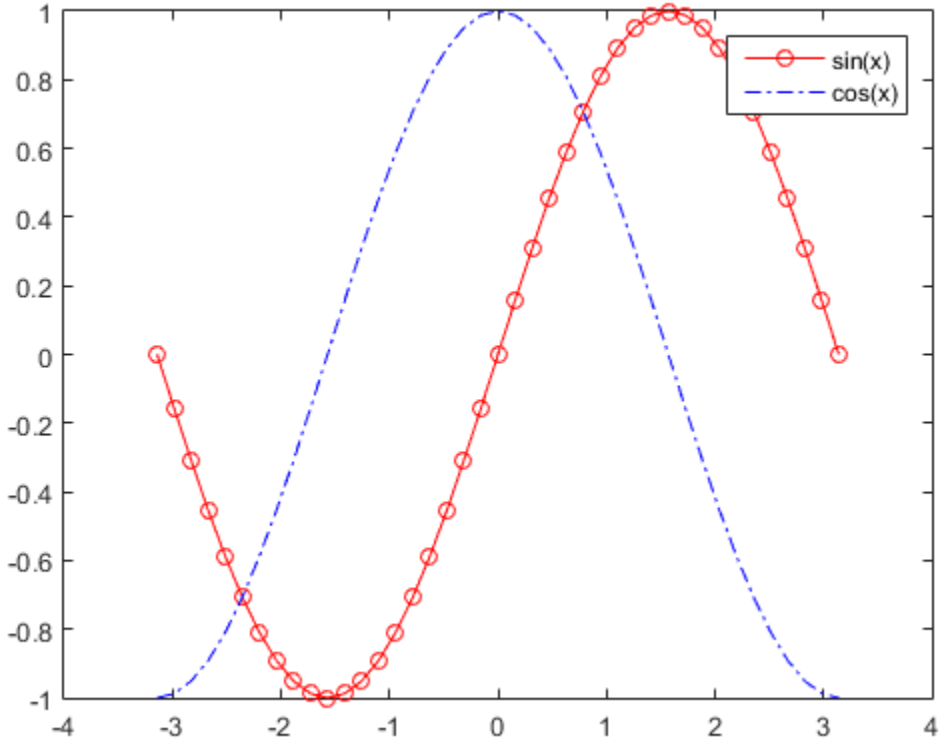
`[h,icons,plots,str] = legend( ___ )` additionally returns the objects used to create the legend icons, the objects plotted in the graph, and an array of text strings.

Examples

Add Legend to Graph

Create a graph of sine and cosine. Add a legend to the graph indicating the two functions.

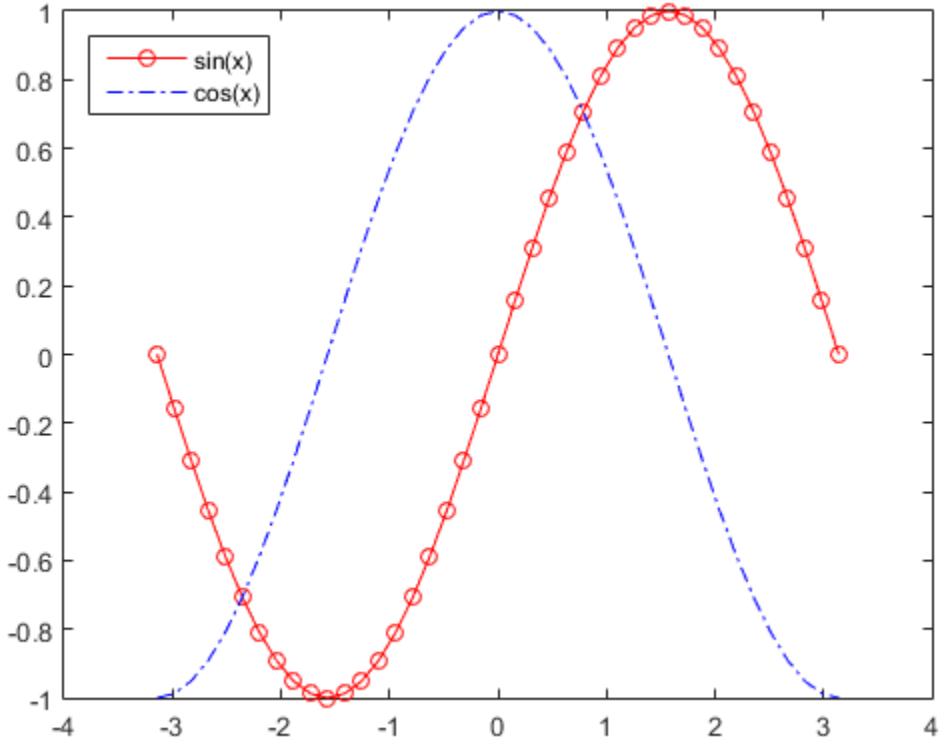
```
x = -pi:pi/20:pi;
y1 = sin(x);
y2 = cos(x);

figure
plot(x,y1,'-ro',x,y2,'-.b')
legend('sin(x)','cos(x)')
```



Specify Legend Location

Create a graph of sine and cosine. Add a legend to the top left corner of the graph by specifying the `'Location'` property as `'northwest'`.

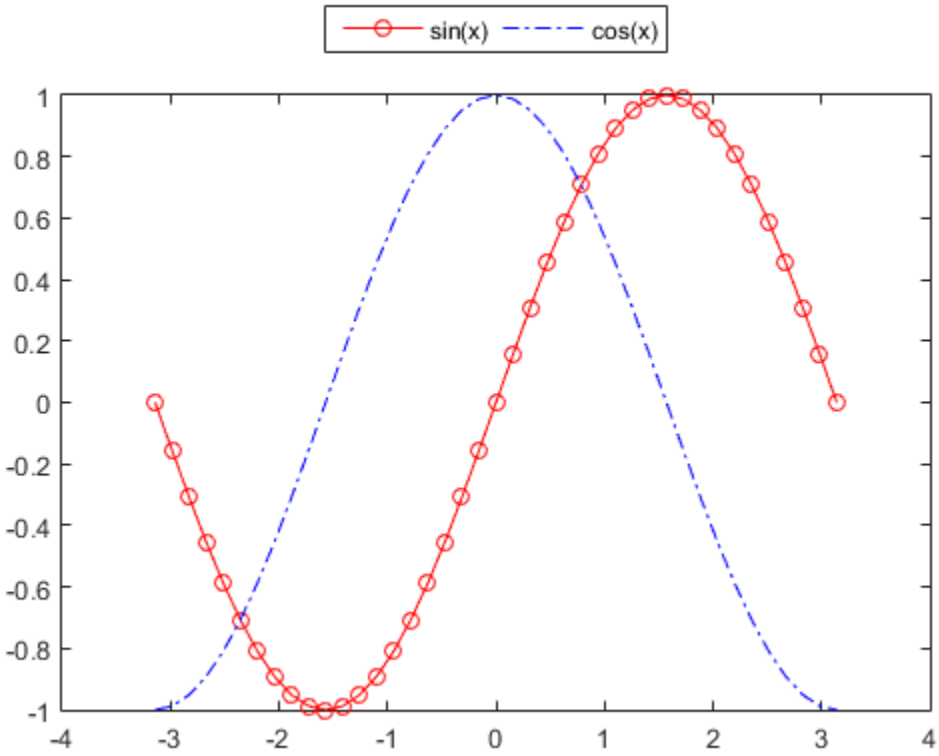
```
x = -pi:pi/20:pi;
y1 = sin(x);
y2 = cos(x);

figure
plot(x,y1,'-ro',x,y2,'-.b')
legend('sin(x)','cos(x)','Location','northwest')
```



Add Horizontal Legend Above Graph

Create a graph of sine and cosine. Add a horizontal legend above the axes.

```
x = -pi:pi/20:pi;
y1 = sin(x);
y2 = cos(x);

figure
plot(x,y1,'-ro',x,y2,'-.b')
legend('sin(x)','cos(x)','Location','northoutside','Orientation','horizontal')
```



Change Legend Font Size

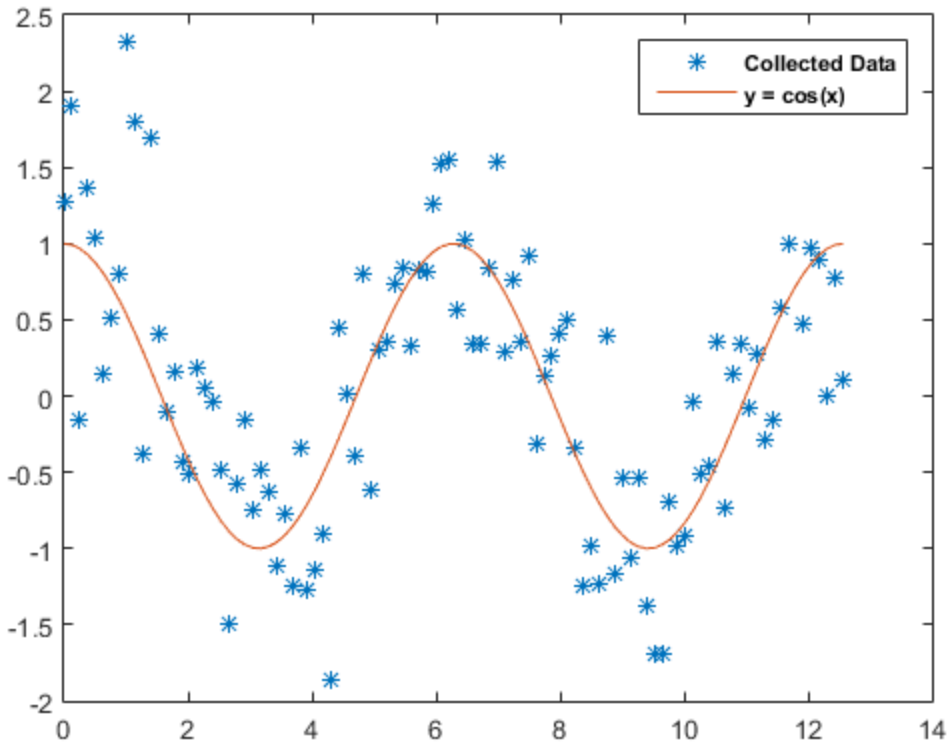
Define the data, `x`, `ydata`, and `ycos`.

```
rng default % initialize random number generator
x = linspace(0,4*pi,100);
ydata = cos(x) + 0.5*randn(1,100);
ycos = cos(x);
```

Plot `ydata` and `ycos`. Add a legend to the graph using a small, bold font. To specify `Name,Value` pairs when creating a legend, you must specify the legend description strings as a cell array by using curly braces, `{}`.

```
figure
```

```
plot(x,ydata,'*',x,ycos)
legend({'Collected Data','y = cos(x)'},'FontSize',8,'FontWeight','bold')
```



Display Subset of Lines in Legend

Define the data.

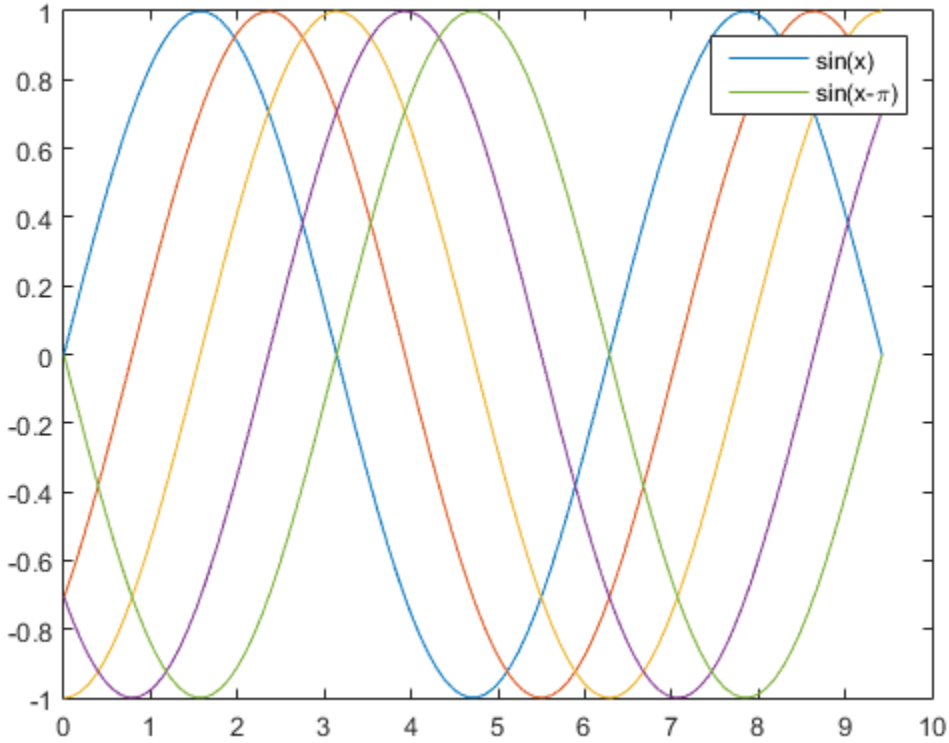
```
x = linspace(0,3*pi);
y1 = sin(x);
y2 = sin(x - pi/4);
y3 = sin(x - pi/2);
y4 = sin(x - 3*pi/4);
y5 = sin(x - pi);
```

Plot the data and return the charting line handles.

```
figure
hLine1 = plot(x,y1);
hold all
hLine2 = plot(x,y2);
hLine3 = plot(x,y3);
hLine4 = plot(x,y4);
hLine5 = plot(x,y5);
hold off
```

Add a legend for the first and last lines by specifying their handles, `hLine1` and `hLine5`.

```
legend([hLine1,hLine5],'sin(x)','sin(x-\pi)')
```

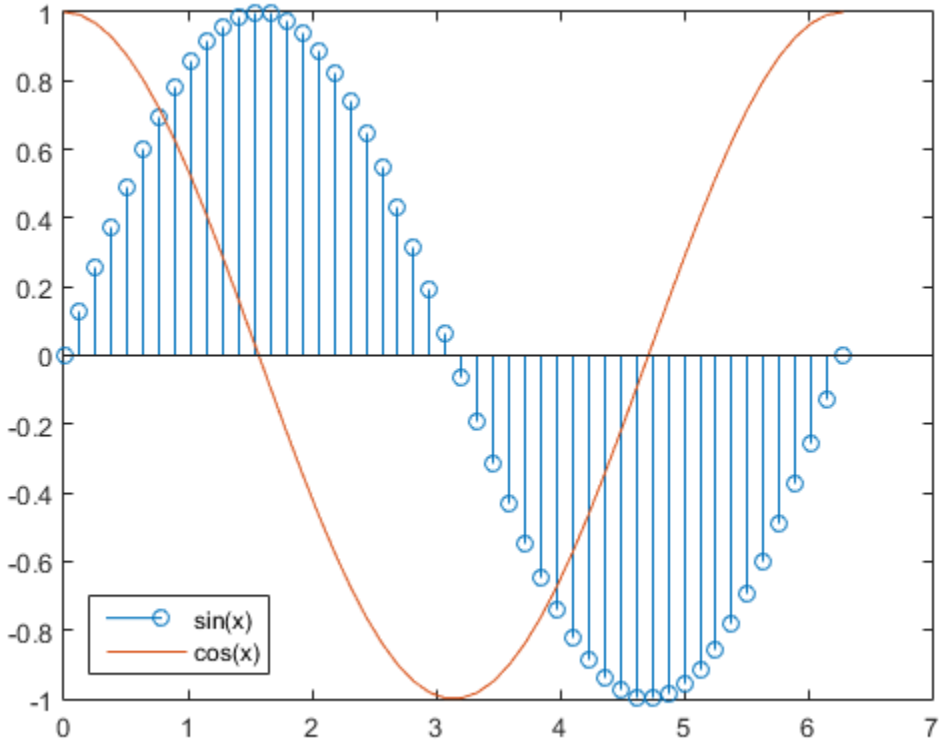


Specify Legend Location and Remove Border

Create a plot of sine and cosine. Add a legend to the bottom left of the axes.

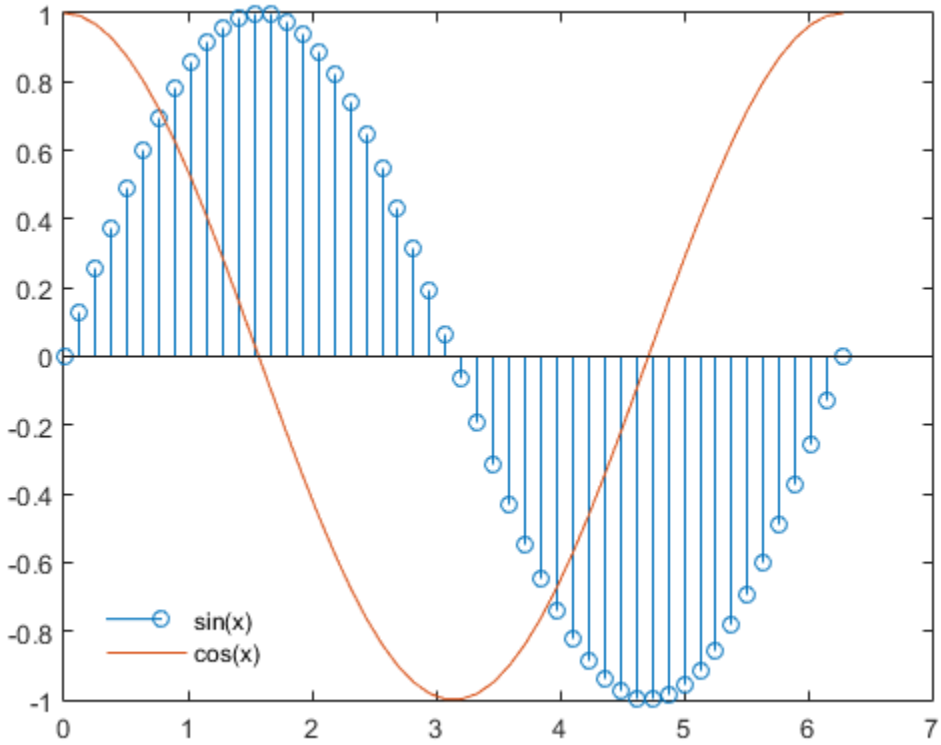
```
x = linspace(0,2*pi,50);
ysin = sin(x);
ycos = cos(x);

figure
stem(x,ysin)
hold all
plot(x,ycos)
hold off
```

```
legend('sin(x)','cos(x)','Location','southwest')
```



Remove the legend border.

```
legend('boxoff')
```



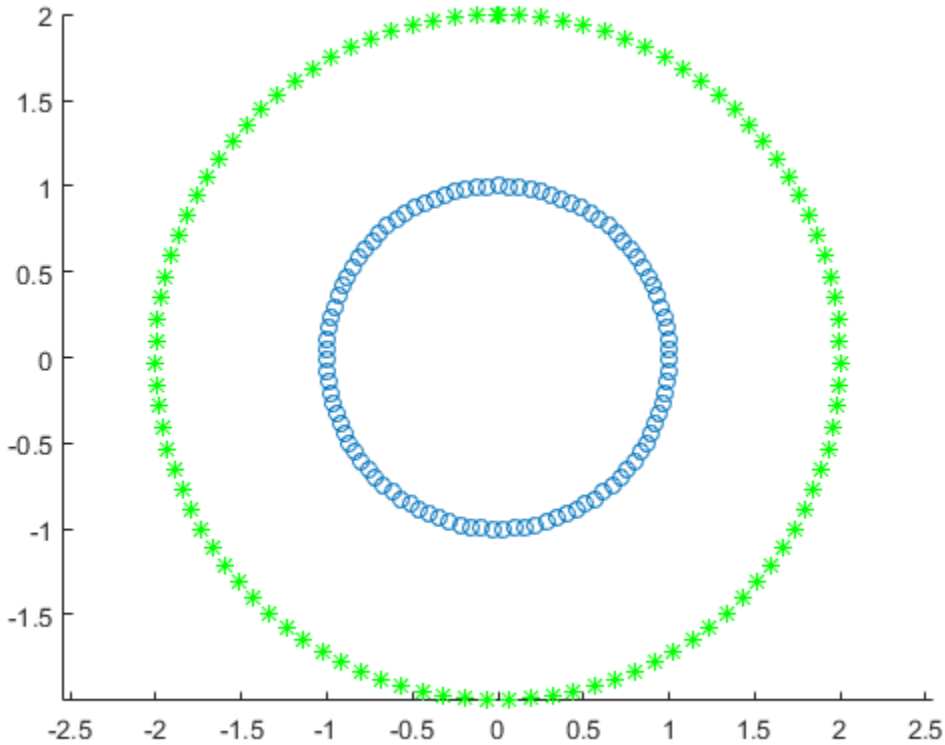
Specify Custom Legend Size and Location

Define the data.

```
theta = linspace(0,2*pi,100);
x1 = sin(theta);
y1 = cos(theta);

x2 = 2*sin(theta);
y2 = 2*cos(theta);
```

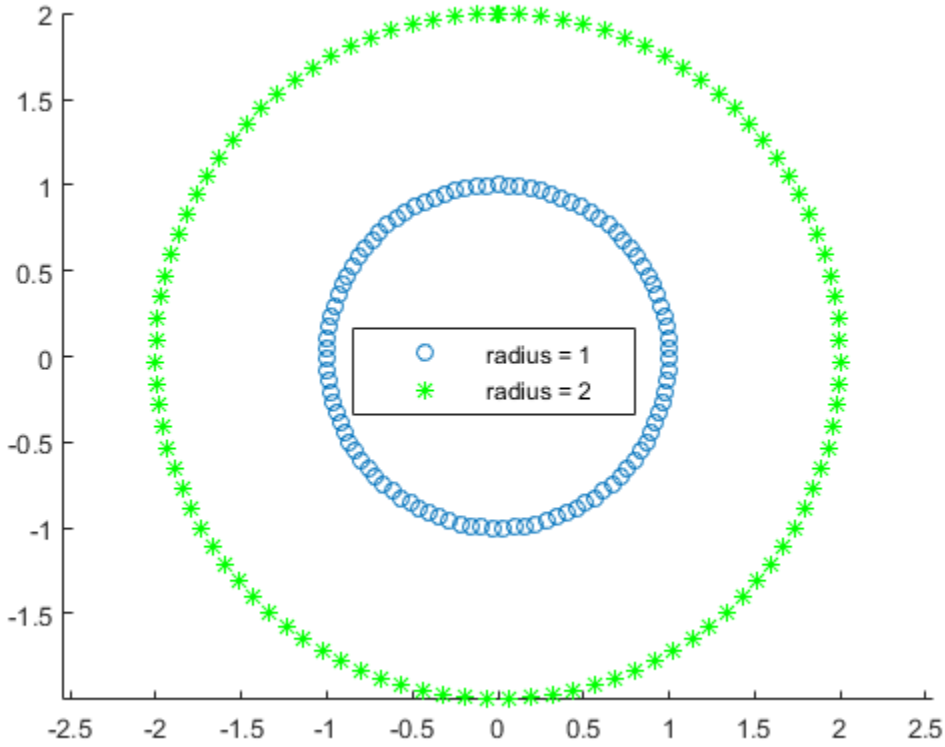
Create a scatter plot of two circles. Use `axis equal` to set equal scaling along both the *x*-axis and *y*-axis.

```
figure
scatter(x1,y1)
hold on
scatter(x2,y2,'*g')
hold off
axis equal
```



Add a legend to the center of the graph using the `Position` property. To set the `Position` property when creating a legend, you must define the legend strings as a cell array by using curly braces, `{}`.

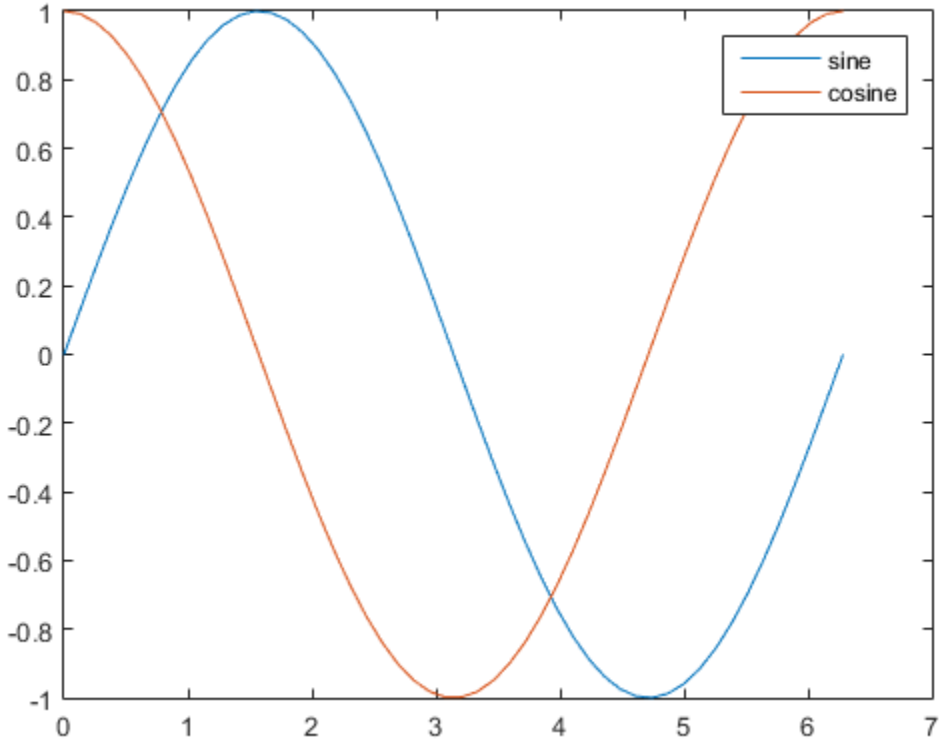
```
legend({'radius = 1','radius = 2'},'Position',[0.39,0.45,0.25,0.1])
```



Delete Legend

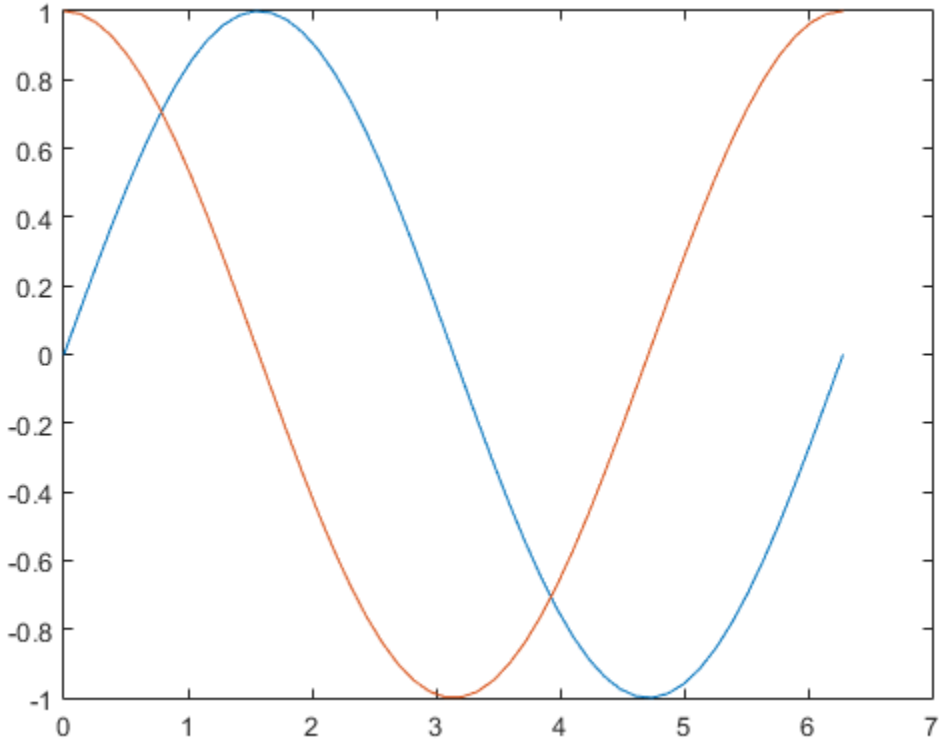
Create a plot of sine and cosine and add a legend.

```
x = linspace(0,2*pi,50);
ysin = sin(x);
ycos = cos(x);

figure
plot(x,ysin,x,ycos)
legend('sine','cosine')
```



Delete the legend.

```
legend('off')
```

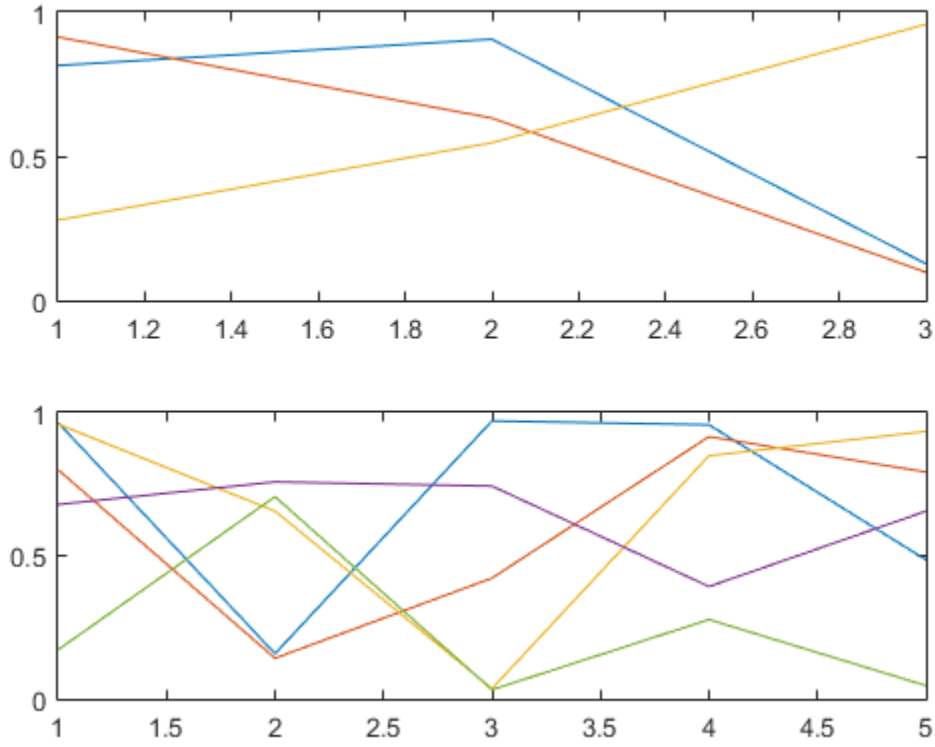


Specify Axes for Legend

Create a figure with two subplots and return the two axes handles, `ax1` and `ax2`. Plot random data in each subplot.

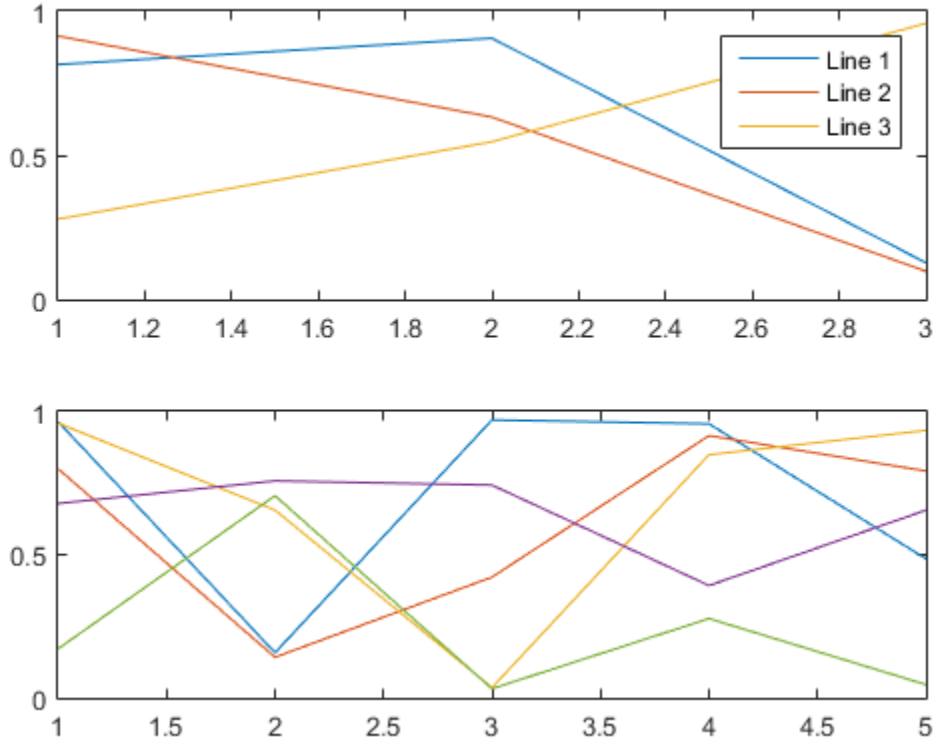
```
rng default % initialize random number generator
y1 = rand(3);
y2 = rand(5);

figure
ax1 = subplot(2,1,1); % upper subplot
plot(y1)

ax2 = subplot(2,1,2); % lower subplot
```

```
plot(y2)
```



Add a legend to the upper subplot by passing its axes handle to `legend`.

```
legend(ax1,'Line 1','Line 2','Line 3')
```



- “Include Subset of Objects in Graph Legend”
- “Specify Legend Descriptions During Line Creation”

Input Arguments

`'string1',...,'stringN'` — Description for each object

comma-separated list of strings

Description for each object, specified as a comma-separated list of strings.

MATLAB sets the `String` property of the legend to the specified strings. MATLAB also sets the `DisplayName` properties of the objects to the specified strings.

If you do not specify legend strings, then `legend` uses the `DisplayName` properties of the objects instead. If an object does not have a `DisplayName` property, or if you have not defined a value for this property, then `legend` uses a string of the form `'data1'`, `'data2'`, and so on.

Example: `'Sine Function','Cosine Function'`

`strings` — Description for each object

cell array of strings | matrix of strings

Description for each object, specified as a cell array of strings or a matrix of strings. If `strings` is a matrix, then the `legend` function uses each row of the matrix to label a set of data.

MATLAB sets the `String` property of the legend to the specified strings. MATLAB also sets the `DisplayName` properties of the objects to the specified strings.

If you do not specify legend strings, then `legend` uses the `DisplayName` properties of the objects instead. If an object does not have a `DisplayName` property, or if you have not defined a value for this property, then `legend` uses a string of the form `'data1'`, `'data2'`, and so on.

Example: `{'Sine Function','Cosine Function'}`

`entries` — List of graphics objects to include in legend

vector

List of graphics objects to include in the legend, specified as a vector. Specifying objects is useful if you do not want to add a description for every object in the graph.

These are the graphics objects that can be included in the legend:

- `Animated line` objects
- `Area` objects
- `Bar series` objects
- `Chart line` objects
- `Chart surface` objects
- `Contour` objects
- `Errorbar series` objects
- `Group` objects

- `Histogram` objects
- `Patch` objects
- `Primitive line` objects
- `Primitive surface` objects
- `Quiver series` objects
- `Scatter series` objects
- `Stair` objects
- `Stem series` objects
- `Transform` objects

ax — Axes object

axes object

Axes object. If you do not specify an axes, then the `legend` function adds a legend to the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are only a subset. For a complete list see Legend Properties.

Example: `'FontSize',12,'TextColor','blue'`

'Location' — Location with respect to axes

supported location string

Location with respect to the axes, specified as one of the location strings listed in this table.

Supported String	Resulting Location
`'north'`	Inside top of axes
`'south'`	Inside bottom of axes
`'east'`	Inside right of axes
`'west'`	Inside left of axes

Supported String	Resulting Location
'northeast'	Inside top-right of axes (default for 2-D axes)
'northwest'	Inside top-left of axes
'southeast'	Inside bottom-right of axes
'southwest'	Inside bottom-left of axes
'northoutside'	Above the axes
'southoutside'	Below the axes
'eastoutside'	To the right of the axes
'westoutside'	To the left of the axes
'northeastoutside'	Outside top-right corner of the axes (default for 3-D axes)
'northwestoutside'	Outside top-left corner of the axes
'southeastoutside'	Outside bottom-right corner of the axes
'southwestoutside'	Outside bottom-left corner of the axes
'best'	Inside axes where least conflict with data in plot
'bestoutside'	To the right of the axes
'none'	Determined by Position property

To display the legend in a location that is not listed in the table, use the `Position` property to specify a custom location. If you set the `Position` property, then MATLAB automatically sets the `Location` property to `'none'`.

Example: `'northoutside'`

`'Orientation'` — Orientation

`'vertical'` (default) | `'horizontal'`

Orientation, specified as one of these values:

- `'vertical'` — Stacks the legend entries vertically. This is the default.
- `'horizontal'` — Lists the legend entries side-by-side.

Example: `'horizontal'`

`'Box'` — Box outline

`'on'` (default) | `'off'`

Box outline, specified as one of these values:

- `'on'` — Displays the box around the legend. This is the default.
- `'off'` — Removes the box around the legend.

Example: `'off'`

`'String'` — Legend entry descriptions

cell array of strings

Legend entry descriptions, specified as a cell array of strings. If you specify strings as input to `legend`, then MATLAB sets the `String` property to the specified strings and uses these strings in the legend display. If you do not specify strings, then MATLAB uses the `DisplayName` properties of the objects, or a string of the form `'data1'`, `'data2'`, and so on. The `String` property reflects the entries in the legend.

Example: `{'Sine','Cosine','Tangent'}`

`'Interpreter'` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	°
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

`'TextColor'` — Text color
`[0,0,0]` or `'black'` (default) | RGB triplet | supported color string

Text color, specified as an RGB triplet or a supported color string. The default color is black with an RGB triplet value of `[0,0,0]`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]

Long Name	Short Name	RGB Triplet
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0,0,1]

Example: 'blue'

'FontSize' — Font size

9 (default) | scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default value is 9 points. If you change the axes font size, then MATLAB automatically sets the legend font size to 90% of the axes font size. If you manually set the legend font size, then changing the axes font size does not affect the legend.

Example: 12

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Output Arguments

h — Legend object

legend object

Legend object. Use h to modify the legend after it has been created.

icons — Objects used to create legend icons and descriptions

text, patch, and line objects

Objects used to create the legend icons and descriptions, returned as text, patch, and line object.

plots — Objects plotted in graph

vector of objects

Objects plotted in the graph, returned as a vector of objects.

`str` — Text strings used in legend

cell array

Text strings used in the legend, returned as a cell array.

More About

Tips

- An axes can have only one legend.
- To use legend keywords, such as `'Location'` or `'off'` as legend descriptions, you must pass the string in a cell array to `legend`. For example, `l = legend({'Location'})` displays a legend with the entry `Location`.
- To label more than 20 objects in the legend, you must specify a string for each object. Otherwise, `legend` depicts only the first 20 objects in the graph.
- `legend` associates strings with objects in the same order that the objects are listed in the axes `Children` property.
- “Control Legend Content”

See Also

Functions

`hold` | `LineSpec` | `plot`

Properties

Legend Properties

Introduced before R2006a

Legend Properties

Control legend appearance and behavior

Legend properties control the appearance and behavior of a legend object. By changing property values, you can modify certain aspects of the legend. Use dot notation to refer to a particular object and property:

```
h = legend('show');
c = h.TextColor;
h.TextColor = 'red';
```

Appearance

`TextColor` — Text color

`[0,0,0]` or `'black'` (default) | RGB triplet | supported color string

Text color, specified as an RGB triplet or a supported color string. The default color is black with an RGB triplet value of `[0,0,0]`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0,0,1]`

Example: `'blue'`

`Color` — Background color

`[1,1,1]` or `'white'` (default) | RGB triplet | supported color string

Background color, specified as an RGB triplet or a supported color string. The default color is white with an RGB triplet value of `[1,1,1]`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0,0,1]`

Example: `'b'`

`Box` — Box outline

`'on'` (default) | `'off'`

Box outline, specified as one of these values:

- `'on'` — Displays the box around the legend. This is the default.
- `'off'` — Removes the box around the legend.

Example: `'off'`

`EdgeColor` — Box outline color

`[0.15 0.15 0.15]` (default) | RGB triplet | supported color string

Box outline color, specified as an RGB triplet or a supported color string. The default color is dark gray with an RGB triplet value of `[0.15,0.15,0.15]`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0,1,0]`

Example: `'g'`

`LineWidth` — Width of box outline

`0.5` (default) | positive value

Width of box outline, specified as a positive value in point units. One point equals 1/72 inches.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Location and Size

`Location` — Location with respect to axes

supported location string

Location with respect to the axes, specified as one of the location strings listed in this table.

Supported String	Resulting Location
'north'	Inside top of axes
'south'	Inside bottom of axes
'east'	Inside right of axes
'west'	Inside left of axes
'northeast'	Inside top-right of axes (default for 2-D axes)
'northwest'	Inside top-left of axes
'southeast'	Inside bottom-right of axes
'southwest'	Inside bottom-left of axes
'northoutside'	Above the axes
'southoutside'	Below the axes
'eastoutside'	To the right of the axes
'westoutside'	To the left of the axes
'northeastoutside'	Outside top-right corner of the axes (default for 3-D axes)
'northwestoutside'	Outside top-left corner of the axes
'southeastoutside'	Outside bottom-right corner of the axes
'southwestoutside'	Outside bottom-left corner of the axes
'best'	Inside axes where least conflict with data in plot
'bestoutside'	To the right of the axes
'none'	Determined by Position property

To display the legend in a location that is not listed in the table, use the `Position` property to specify a custom location. If you set the `Position` property, then MATLAB automatically sets the `Location` property to `'none'`.

Example: `'northoutside'`

`Orientation` — Orientation

`'vertical'` (default) | `'horizontal'`

Orientation, specified as one of these values:

- `'vertical'` — Stacks the legend entries vertically. This is the default.
- `'horizontal'` — Lists the legend entries side-by-side.

Example: `'horizontal'`

`Position` — Custom location and size

four-element vector

Custom location and size, specified as a four-element vector of the form `[left, bottom, width, height]`. The first two values, `left` and `bottom`, specify the distance from the lower-left corner of the figure to the lower-left corner of the legend. The last two values, `width` and `height`, specify the dimensions of the legend. The `Units` property determines the position units.

If you specify the `Position` property, then MATLAB automatically changes the legend's `Location` property to `'none'`.

Example: `[0.1,0.1,0.3,0.7]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

`Units`	Description
`'normalized'` (default)	Normalized with respect to the container, which is usually the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.

Units	Description
'points'	Points. One point equals 1/72 inch.
'pixels'	Pixels. Pixel size depends on the screen resolution.

All units are measured from the lower-left corner of the container window.

This property affects the Position property. If you change the units, then it is good practice to return it to its default value after completing your computation to prevent affecting other functions that assume Units is the default value.

If you specify the Position and Units properties as `Name,Value` pairs when creating the legend, then the order of specification matters. If you want to define the position with particular units, then you must set the Units property before the Position property.

Text

`String` — Legend entry descriptions

cell array of strings

Legend entry descriptions, specified as a cell array of strings. If you specify strings as input to `legend`, then MATLAB sets the `String` property to the specified strings and uses these strings in the legend display. If you do not specify strings, then MATLAB uses the `DisplayName` properties of the objects, or a string of the form `'data1'`, `'data2'`, and so on. The `String` property reflects the entries in the legend.

Example: `{'Sine','Cosine','Tangent'}`

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	Φ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	ß	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	Θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	v	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

Font Style

`FontAngle` — Character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Note: The `'oblique'` value has been removed. Use `'italic'` instead.

`FontName` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`FontSize` — Font size

9 (default) | scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default value is 9 points. If you change the axes font size, then MATLAB automatically sets the legend font size to 90% of the axes font size. If you manually set the legend font size, then changing the axes font size does not affect the legend.

Example: `12`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`FontWeight` — Thickness of text characters

`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

Visibility

`Visible` — Visibility of legend

`'on'` (default) | `'off'`

Visibility of legend, specified as one of these values:

- `'on'` — Display the legend.
- `'off'` — Hide the legend without deleting it. You still can access the properties of an invisible legend object.

Identifiers

`Type` — Type of graphics object

`'legend'` (default)

Type of graphics object, returned as `'legend'`. Use this property to find all objects of a given type within a plotting hierarchy.

`Tag` — Tag to associate with legend

`'legend'` (default) | string

Tag to associate with the legend, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

Data Types: `char`

`UserData` — Data to associate with legend

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the legend object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent of legend

figure object | uipanel object | uitab object

Parent of the legend, specified as a figure object, uipanel object, or a uitab object.

The legend must have the same parent as the associated axes. If you change the parent of the associated axes, then the legend automatically updates to use the same parent.

`Children` — Children of legend

empty `GraphicsPlaceholder` array

The legend has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of legend object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The legend object handle is always visible.
- `'off'` — The legend object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The legend object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the legend at the command-line, but allows callback functions to access it.

If the legend object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Callback that executes when you click the legend

`@bdowncb` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the legend. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The legend object — You can access properties of the legend object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu handle (default)

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the legend. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the legend when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the legend.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the legend that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The HitTest property determines if the legend responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the legend passes the click to the object below it in the current view of the figure window. The HitTest property of the legend has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the legend. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the legend that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the legend object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'off'` (default) | `'on'`

Callback interruption, specified as `'off'` or `'on'`. The `Interruptible` property determines if a running callback can be interrupted.

If the `ButtonDownFcn` callback of the legend is the running callback, then the `Interruptible` property determines if it can be interrupted by another callback. The `Interruptible` property has two possible values:

- `'off'` — The running callback cannot be interrupted. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.
- `'on'` — The running callback can be interrupted. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

Example: `'off'`

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the legend tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the legend. Setting the `CreateFcn` property on an existing legend has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during legend creation. MATLAB executes the callback after creating the legend and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The legend object — You can access properties of the legend object from within the callback function. You also can access the legend object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the legend. MATLAB executes the callback before destroying the legend so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The legend object — You can access properties of the legend object from within the callback function. You also can access the legend object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of legend

`'off'` (default) | `'on'`

Deletion status of legend, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the legend begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the legend no longer exists.

Check the value of the BeingDeleted property to verify that the legend is not about to be deleted before querying or modifying it.

See Also

`legend`

More About

- "Access Property Values"
- "Graphics Object Properties"

legendre

Associated Legendre functions

Syntax

```
P = legendre(n,X)
S = legendre(n,X,'sch')
N = legendre(n,X,'norm')
```

Description

`P = legendre(n,X)` computes the associated Legendre functions of degree `n` and order `m = 0,1,...,n`, evaluated for each element of `X`. Argument `n` must be a scalar integer, and `X` must contain real values in the domain $-1 \le x \le 1$.

If `X` is a vector, then `P` is an `(n+1)`-by-`q` matrix, where `q = length(X)`. Each element `P(m+1,i)` corresponds to the associated Legendre function of degree `n` and order `m` evaluated at `X(i)`.

In general, the returned array `P` has one more dimension than `X`, and each element `P(m+1,i,j,k,...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(i,j,k,...)`. Note that the first row of `P` is the Legendre polynomial evaluated at `X`, i.e., the case where `m` = 0.

`S = legendre(n,X,'sch')` computes the "Schmidt Seminormalized Associated Legendre Functions" on page 1-4453.

`N = legendre(n,X,'norm')` computes the "Fully Normalized Associated Legendre Functions" on page 1-4453.

Examples

Example 1

The statement `legendre(2,0:0.1:0.2)` returns the matrix

	x = 0	x = 0.1	x = 0.2
m = 0	-0.5000	-0.4850	-0.4400
m = 1	0	-0.2985	-0.5879
m = 2	3.0000	2.9700	2.8800

Example 2

Given,

```
X = rand(2,4,5);
n = 2;
P = legendre(n,X)
```

then

```
size(P)
ans =
     3     2     4     5
```

and

```
P(:,1,2,3)
ans =
   -0.2475
   -1.1225
    2.4950
```

is the same as

```
legendre(n,X(1,2,3))
ans =
   -0.2475
   -1.1225
    2.49501
```

More About

Associated Legendre Functions

The Legendre functions are defined by

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$$

where

$$P_n(x)$$

is the Legendre polynomial of degree n:

$$P_n(x) = \frac{1}{2^n n!}\left[\frac{d^n}{dx^n}(x^2-1)^n\right]$$

Schmidt Seminormalized Associated Legendre Functions

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions $P_n^m(x)$ by

$$P_n(x) \text{ for } m = 0,$$
$$S_n^m(x) = (-1)^m \sqrt{\frac{2(n-m)!}{(n+m)!}} P_n^m(x) \text{ for } m > 0.$$

Fully Normalized Associated Legendre Functions

The fully normalized associated Legendre functions are normalized such that

$$\int_{-1}^{1} \left(N_n^m(x)\right)^2 dx = 1$$

and are related to the unnormalized associated Legendre functions $P_n^m(x)$ by

$$N_n^m = (-1)^m \sqrt{\frac{(n+\frac{1}{2})(n-m)!}{(n+m)!}} P_n^m(x)$$

Algorithms

`legendre` uses a three-term backward recursion relationship in `m`. This recursion is on a version of the Schmidt seminormalized associated Legendre functions $Q_n^m(x)$, which are complex spherical harmonics. These functions are related to the standard Abramowitz and Stegun [1] functions $P_n^m(x)$ by

$$P_n^m(x) = \sqrt{\frac{(n+m)!}{(n-m)!}}\, Q_n^m(x)$$

They are related to the Schmidt form given previously by

$$m = 0: \quad S_n^m(x) = Q_n^0(x)$$
$$m > 0: \quad S_n^m(x) = (-1)^m \sqrt{2}\, Q_n^m(x)$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Ch.8.

[2] Jacobs, J. A., *Geomagnetism*, Academic Press, 1987, Ch.4.

Introduced before R2006a

length

Length of largest array dimension

Syntax

```
L = length(X)
```

Description

`L = length(X)` returns the length of the largest array dimension in `X`. For vectors, the length is simply the number of elements. For arrays with more dimensions, the length is `max(size(X))`. The length of an empty array is zero.

Examples

Number of Vector Elements

Find the length of a uniformly spaced vector in the interval `[5,10]`.

```
v = 5:10
```

```
v =

     5     6     7     8     9    10
```

```
L = length(v)
```

```
L =

     6
```

Length of Rectangular Matrix

Find the length of a 3-by-7 matrix of zeros.

```
X = zeros(3,7);
L = length(X)
```

```
L =

     7
```

Length of Structure Fields

Create a structure with fields for `Day` and `Month`. Use the `structfun` function to apply `length` to each field.

```
S = struct('Day',[1 13 14 26],'Month',{{'Jan','Feb', 'Mar'}})
```

```
S =

      Day: [1 13 14 26]
    Month: {'Jan'  'Feb'  'Mar'}
```

```
L = structfun(@(field) length(field),S)
```

```
L =

     4
     3
```

Input Arguments

X — Input array

numeric array | character array | logical array | structure | cell array | categorical array | datetime array | duration array | calendarDuration array

Input array, specified as a numeric array, character array, logical array, structure, cell array, categorical array, datetime array, duration array, or calendarDuration array.

`length` does not operate on tables. To examine the dimensions of a table, use the `height`, `width`, or `size` functions.

Complex Number Support: Yes

See Also

`ndims` | `numel` | `size`

Introduced before R2006a

length

Class: containers.Map
Package: containers

Length of `containers.Map` object

Syntax

```
mapLength = length(mapObj)
```

Description

`mapLength = length(mapObj)` returns the number of key-value pairs in `mapObj`.

Input Arguments

`mapObj`

Object of class `containers.Map`.

Output Arguments

`mapLength`

Scalar numeric value that indicates the number of key-value pairs in `mapObj`. This number is equivalent to `size(mapObj,1)` and to `mapObj.Count`.

Examples

Determine the Length of a Map

Create a map, and determine the number of key-value pairs:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
mapObj = containers.Map(myKeys,myValues);

mapLength = length(mapObj)
```

This code returns

```
mapLength =
     3
```

See Also

isKey | keys | values | containers.Map | size

length (serial)

Length of serial port object array

Syntax

```
length(obj)
```

Description

`length(obj)` returns the length of the serial port object, `obj`. It is equivalent to the command `max(size(obj))`.

See Also

`size`

Introduced before R2006a

length (tscollection)

Length of time vector

Syntax

```
length(tsc)
```

Description

`length(tsc)` returns an integer that represents the length of the time vector for the `tscollection` object `tsc`.

See Also

`tscollection` | `isempty (tscollection)` | `size (tscollection)`

Introduced before R2006a

libfunctions

Return information on functions in shared library

Syntax

```
libfunctions libname

m = libfunctions(libname)
___ = libfunctions( ___ ,'-full')
```

Description

`libfunctions libname` displays names of functions defined in library, libname. If you called `loadlibrary` using the `alias` option, then you must use the alias name for the libname argument.

`m = libfunctions(libname)` returns names of functions in cell array m.

`___ = libfunctions( ___ ,'-full')` returns function signatures.

Examples

Display Function Signatures

Load the example library, `shrlibsample`, and list the functions.

```
if not(libisloaded('shrlibsample'))
  addpath(fullfile(matlabroot,'extern','examples','shrlib'))
  loadlibrary('shrlibsample')
end
m = libfunctions('shrlibsample','-full')
```

```
m =

    '[double, doublePtr] addDoubleRef(double, doublePtr, double)'
    'double addMixedTypes(int16, int32, double)'
    '[double, c_structPtr] addStructByRef(c_structPtr)'
    'double addStructFields(c_struct)'
```

```
'c_structPtrPtr allocateStruct(c_structPtrPtr)'
'voidPtr deallocateStruct(voidPtr)'
'lib.pointer exportedDoubleValue'
'lib.pointer getListOfStrings'
'doublePtr multDoubleArray(doublePtr, int32)'
'[lib.pointer, doublePtr] multDoubleRef(doublePtr)'
'int16Ptr multiplyShort(int16Ptr, int32)'
'doublePtr print2darray(doublePtr, int32)'
'printExportedDoubleValue'
'cstring readEnum(Enum1)'
'[cstring, cstring] stringToUpper(cstring)'
```

Input Arguments

`libname` — Name of shared library

string

Name of shared library, specified as a string. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

Output Arguments

`m` — Function names

cell array

Functions names, returned as a cell array.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

See Also

`calllib` | `libfunctionsview` | `loadlibrary`

Introduced before R2006a

libfunctionsview

Display shared library function signatures in window

Syntax

```
libfunctionsview libname
```

Description

`libfunctionsview libname` displays information about functions in library, libname, in a new window.

Input Arguments

`libname` — Name of shared library

string

Name of shared library, specified as a string. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Examples

Display Function Signatures for Library libmx

```
if not(libisloaded('libmx'))
```

```
  hfile = fullfile(matlabroot,'extern','include','matrix.h');
  loadlibrary('libmx',hfile)
end
libfunctionsview libmx
```

MATLAB creates a new window displaying function signatures.

See Also

`libfunctions` | `calllib`

Introduced before R2006a

libisloaded

Determine if shared library is loaded

Syntax

```
tf = libisloaded(libname)
```

Description

`tf = libisloaded(libname)` returns logical `1` (`true`) if the shared library, libname, is loaded. Otherwise, it returns logical `0` (`false`).

Examples

Load Functions in Library

Load example library, `shrlibsample`, if it is not already loaded.

```
if ~libisloaded('shrlibsample')
  loadlibrary('shrlibsample')
end
```

Input Arguments

`libname` — Name of shared library

string

Name of shared library, specified as a string. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

See Also

`loadlibrary` | `unloadlibrary`

Introduced before R2006a

libpointer

Pointer object for use with shared library

Syntax

```
p = libpointer
p = libpointer(DataType)
p = libpointer(DataType,Value)
```

Description

`p = libpointer` creates `NULL` pointer `p` of type `voidPtr`.

`p = libpointer(DataType)` creates `NULL` pointer of specified `DataType`.

`p = libpointer(DataType,Value)` creates pointer initialized to a copy of `Value`.

Examples

Create NULL Pointer for string Argument

```
p = libpointer('string');
```

- "Multilevel Pointers"

Input Arguments

`DataType` — Type of pointer

string

Type of pointer, specified as a string, of any MATLAB numeric type, structure defined in the library, or enumeration defined in the library. For a list of valid MATLAB numeric types, refer to these tables in "C and MATLAB Equivalent Types".

- MATLAB Primitive Types

- MATLAB Extended Types

Example: `'int16Ptr'`

Data Types: `char`

`Value` — Value for pointer object

any valid value

Value, specified as any valid value for given type.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

More About

Tips

- This is an advanced feature for experienced C programmers. MATLAB automatically converts data passed to and from external library functions to the data type expected by the external function. Use a `lib.pointer` object instead of automatic conversion in the following situations.
  - You want to modify the data in the input arguments.
  - You are passing large amounts of data, and you want to control when MATLAB makes copies of the data.
  - The library stores and uses the pointer for a period of time so you want the MATLAB function to control the lifetime of the `lib.pointer` object.
- "Working with Pointer Arguments"

See Also

`lib.pointer` | `calllib` | `libstruct`

Introduced before R2006a

lib.pointer class

Package: lib

Pointer object compatible with C pointer

Description

MATLAB automatically converts arguments that are passed by reference to a function in an external library. Use a pointer object instead of automatic conversion in the following situations.

- The function modifies data in an input argument.
- You are passing large amounts of data, and you want to control when MATLAB makes copies of the data.
- The library stores and uses the pointer for a period of time so you want the MATLAB function to control the lifetime of the `lib.pointer` object.

Construction

To create a `lib.pointer` object, use the MATLAB `libpointer` function.

A library function can return a `lib.pointer` object. Use the `setdatatype` method to manually convert the argument to use in MATLAB.

Properties

`DataType`

Type of pointer, specified as a string, of any MATLAB numeric type, structure defined in the library, or enumeration defined in the library. For a list of valid MATLAB numeric types, refer to these tables in “C and MATLAB Equivalent Types”.

- MATLAB Primitive Types
- MATLAB Extended Types

`Value`

Value, specified as any valid value for given type.

Methods

disp	Display lib.pointer type
isNull	Points to NULL pointer
plus	+ (plus) operator for pointer arithmetic
reshape	Reshape lib.pointer array
setdatatype	Initialize type and size of lib.pointer object

Definitions

A passed-by-reference argument in the function signature has type names ending with `Ptr` or `PtrPtr`.

Examples

Create Pointer

Create a pointer, `pv`, of type `int16`, initialized to `485`.

```
pv = libpointer('int16Ptr',485);
```

Display the properties of `pv`.

```
get(pv)
```

```
        Value: 485
```

```
    DataType: 'int16Ptr'
```

See Also

`libpointer`

More About

- “Working with Pointer Arguments”

Introduced before R2006a

disp

Class: lib.pointer
Package: lib

Display lib.pointer type

Syntax

```
disp(h)
```

Description

`disp(h)` displays type for `lib.pointer` object, h.

Input Arguments

`h` — `lib.pointer` object.

handle

`lib.pointer` object, specified as a handle.

Examples

Display lib.pointer Type

Create a `double` pointer set to `15`.

```
xp = libpointer('doublePtr',15);
```

Display pointer data type.

```
disp(xp)
```

```
libpointer
```

isNull

Class: lib.pointer
Package: lib

Points to NULL pointer

Syntax

```
tf = isNull(h)
```

Description

`tf = isNull(h)` returns true if h is a `lib.pointer` object.

Input Arguments

`h` — `lib.pointer` object.

handle

`lib.pointer` object, specified as a handle.

Examples

Create Null lib.pointer Object

```
nullp = libpointer('doublePtr',[]);
isNull(nullp)
```

plus

Class: lib.pointer
Package: lib

+ (plus) operator for pointer arithmetic

Syntax

```
hout = plus(h,offset)
hout = h + offset
```

Description

`hout = plus(h,offset)` returns pointer `hout`. Pointer `hout` is valid only as long as the original pointer, h, exists.

`hout = h + offset` is an alternative syntax.

Input Arguments

`h` — `lib.pointer` object.

handle

`lib.pointer` object, specified as a handle.

`offset` — Scalar increment

`uint64`

Scalar increment, specified as `uint64`, from h.

Related Examples

- "Iterate Through an Array"

reshape

Class: lib.pointer
Package: lib

Reshape lib.pointer array

Syntax

```
reshape(h,xdim,ydim)
```

Description

`reshape(h,xdim,ydim)` creates an xdim-by-ydim matrix from `lib.pointer` object h.

Input Arguments

`h` — `lib.pointer` object.

handle

`lib.pointer` object, specified as a handle.

`xdim` — Size of x dimension

double

Size of x dimension, specified as `double`.

`ydim` — Size of y dimension

double

Size of y dimension, specified as `double`.

setdatatype

Class: lib.pointer
Package: lib

Initialize type and size of lib.pointer object

Syntax

```
setdatatype(h,type,size)
```

Description

`setdatatype(h,type,size)` sets data type, type, of size, to `lib.pointer`, `h`.

Input Arguments

`h` — `lib.pointer` object.

handle

`lib.pointer` object, specified as a handle.

`type` — Data type

string

Data type, specified as a string.

`size` — Size

double

Size, specified as `double`.

Examples

Set Size and Type of lib.pointer Output Variable

Load the `shrlibsample` library.

```
if ~libisloaded('shrlibsample')
  addpath(fullfile(matlabroot,'extern','examples','shrlib'))
  loadlibrary('shrlibsample')
end
```

The `multDoubleRef` function takes a scalar value specified as `doubleptr`. Create variable `xp` as a `lib.pointer` object, and call the function.

```
xp = libpointer('doublePtr',99);
[xobj,xval] = calllib('shrlibsample','multDoubleRef',xp);
```

To use the variable `xobj`, set its size and data type.

```
setdatatype(xobj,'doublePtr',1,1)
xobj.Value
```

```
ans =

   495
```

libstruct

Convert MATLAB structure to C-style structure for use with shared library

Syntax

```
S = libstruct(structtype)
S = libstruct(structtype,mlstruct)
```

Description

`S = libstruct(structtype)` creates `NULL` pointer to MATLAB `libstruct` object S.

`S = libstruct(structtype,mlstruct)` creates pointer initialized to mlstruct.

Examples

Call Function with `c_struct` Input Argument

Call the `addStructFields` function by creating a variable of type `c_struct`.

Load the `shrlibsample` library, which contains the `c_struct` type.

```
if ~libisloaded('shrlibsample')
   addpath(fullfile(matlabroot,'extern','examples','shrlib'))
   loadlibrary('shrlibsample')
end
```

Display function signatures for `shrlibsample` and search the list for the `addStructFields` entry.

```
libfunctionsview shrlibsample
```

```
double addStructFields(c_struct)
```

The input argument is a pointer to a `c_struct` data type.

Create a MATLAB structure, `sm`.

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
```

Construct a `libstruct` object `sc` from the `c_struct` type.

```
sc = libstruct('c_struct',sm)
```

The fields of `sc` contain the values of the MATLAB structure, `sm`.

Call the `addStructFields` function.

```
calllib('shrlibsample','addStructFields',sc)
```

```
ans =
    1177
```

To clean up, first clear the `libstruct` object, and then unload the library.

```
clear sc
unloadlibrary shrlibsample
```

Input Arguments

`structtype` — C structure

structure

C structure defined in shared library.

`mlstruct` — MATLAB structure

structure

MATLAB structure used to initialize the fields in `S`.

Data Types: `struct`

Output Arguments

`S` — Pointer

MATLAB `libstruct` object

Pointer, returned as MATLAB `libstruct` object.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.
- You can only use the `libstruct` function on scalar structures.
- When converting a MATLAB structure to a `libstruct` object, the structure must adhere to the requirements listed in “Structure Argument Requirements”.

More About

Tips

- If a function in the shared library has a structure argument, use `libstruct` to create the argument. The `libstruct` function creates a C-style structure that you pass to functions in the library. You handle this structure in MATLAB as you would a true MATLAB structure.
- “Working with Structure Arguments”

See Also

`libfunctionsview` | `loadlibrary`

Introduced before R2006a

license

Get license number or perform licensing task

Syntax

```
license
license('inuse')

S = license('inuse')
S = license('inuse',feature)

status = license('test',feature)
license('test',feature,toggle)

[status,errmsg] = license('checkout',feature)
```

Description

`license` returns the license number for this MATLAB product. The return value also can be `'demo'` for a demonstration version of MATLAB, `'student'`, for a student version of MATLAB, or `'unknown'`, if the license number cannot be determined.

`license('inuse')` displays a list of licenses checked out in the current MATLAB session. The product list is alphabetical by license feature name. These names are the same as the valid values for the feature input.

`S = license('inuse')` returns an array of structures indicating checked-out licenses and the user name of each person who has a license checked out.

`S = license('inuse',feature)` checks if the product specified by `feature` is checked out in the current MATLAB session. If the product is checked out, then `license` returns the product name and the user name of the person who has it checked out. Otherwise, the fields of `S` are empty.

`status = license('test',feature)` tests if a license exists for the product specified by `feature`.

`license('test',feature,toggle)` enables or disables testing of the product specified by `feature`, depending on the value of `toggle`.

`[status,errmsg] = license('checkout',feature)` checks out a license for the specified product. If you specify the optional second output argument, `errmsg`, then license returns the text of any error message encountered if the checkout is unsuccessful.

Examples

Display Licenses in Use

Display a list of licenses currently being used.

```
license('inuse')
```

```
image_toolbox
map_toolbox
matlab
```

`license` displays a list of products in alphabetical order by the license feature name.

Get Licenses in Use and User Names

Get a list of licenses in use with information about each user.

```
S = license('inuse');
```

`license` returns a structure array.

View the first element of `S`.

```
S(1)
```

```
ans =

    feature: 'image_toolbox'
       user: 'juser'
```

Determine If License Is in Use

Determine if the license for MATLAB is in use.

```
S = license('inuse','MATLAB')

S =

    feature: 'matlab'
       user: 'jsmith'
```

If the license is in use, then `S` is a structure array with nonempty fields.

Determine If License Exists

Determine if a license exists for Mapping Toolbox™.

```
status = license('test','map_toolbox')

status =

    1
```

`license` returns `1` if a license exists.

Check Out License

Check out a license for Control System Toolbox.

```
[status,errmsg] = license('checkout','control_toolbox')

status =

     1


errmsg =

     ''
```

The `status` output is `1` and the `errmsg` output is empty if the checkout is successful.

Input Arguments

`feature` — License feature name

string

License feature name, specified as a string. The `INCREMENT` lines in a license file indicate the valid strings. To locate your license file, see Where are the license files for MATLAB located?

As an example, this table lists the `feature` value for a few commonly used products.

`feature` Value	MathWorks Product
`'MATLAB'`	MATLAB
`'SIMULINK'`	Simulink
`'Control_Toolbox'`	Control System Toolbox™
`'Curve_Fitting_Toolbox'`	Curve Fitting Toolbox™
`'Signal_Blocks'`	DSP System Toolbox™
`'Image_Toolbox'`	Image Processing Toolbox
`'Optimization_Toolbox'`	Optimization Toolbox™
`'Distrib_Computing_Toolbox'`	Parallel Computing Toolbox
`'Signal_Toolbox'`	Signal Processing Toolbox
`'Stateflow'`	Stateflow®
`'Statistics_Toolbox'`	Statistics and Machine Learning Toolbox
`'Symbolic_Toolbox'`	Symbolic Math Toolbox™

Values of `feature` are not case-sensitive.

`toggle` — Ability to test product license

`'enable' | 'disable'`

Ability to test the existence of a product license, specified as either `'enable'` or `'disable'`.

- If `toggle` is `'enable'`, then the syntax, `license('test',feature)` returns `1` when the product license exists and `0` when the product license does not exist.
- If `toggle` is `'disable'`, then the syntax, `license('test',feature)` always returns `0` (product license does not exist) for the specified product.

Note Disabling a test for a particular product can affect other tests for the existence of the license, not just tests performed by the `license` command.

Output Arguments

`S` — Checked out products

array of structures

Checked out products, returned as an array of structures, where each structure represents a checked-out license. The structures contain two fields:

- `feature` — license feature name
- `user` — user name of the person who has the license checked out

If the fields are empty, then the product is not currently checked out.

`status` — Test or checkout status

`1` | `0`

Test or checkout status, returned as `1` or `0`.

- When testing for the existence of a license, `1` indicates that the license exists, and `0` indicates that the license does not exist.

  The existence of a license does not necessarily mean that the license can be checked out or that the product is installed. `status` is `1` even if the license has expired or if a system administrator has excluded you from using the product.
- When checking out a license, `1` indicates that the checkout is successful, and `0` indicates that the `license` function could not check out a license.

`errmsg` — Error message

string

Error message for unsuccessful license checkout, returned as a string. If the checkout is successful, then `errmsg` is empty.

More About

- "License Management and Software Updates"

See Also

`isstudent`

Introduced before R2006a

light

Create light object

Syntax

```
light('PropertyName',propertyvalue,...)
handle = light(...)
```

Properties

For a list of properties, see Light Properties.

Description

`light` creates a light object in the current axes. Lights affect only patch and surface objects.

`light('PropertyName',propertyvalue,...)` creates a light object using the specified values for the named properties. For a description of the properties, see Light Properties. The MATLAB software parents the light to the current axes unless you specify another axes with the `Parent` property.

`handle = light(...)` returns the handle of the light object created.

Examples

Light the `peaks` surface plot with a local light source oriented along the direction defined by the vector `[-1 0 0]`, that is, looking from `0` along the positive x-axis.

```
surf(peaks,'FaceLighting','gouraud','FaceColor','interp',...
      'AmbientStrength',0.5)
light('Position',[-1 0 0],'Style','local')
```

Tutorials

For more information about lighting, see “Lighting Overview”.

More About

Tips

You cannot see a light object *per se*, but you can see the effects of the light source on patch and surface objects. You can also specify an axes-wide ambient light color that illuminates these objects. However, ambient light is visible only when at least one light object is present and visible in the axes.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see `set` and `get` for examples of how to specify these data types).

See also the Patch Properties and Primitive Surface Properties `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, `SpecularColorReflectance`, and `VertexNormals` properties. Also see the `lighting` and `material` commands.

See Also

`lighting` | `patch` | `material` | `surface`

Introduced before R2006a

Light Properties

Control light appearance and behavior

Light properties control the appearance and behavior of light objects. By changing property values, you can modify certain aspects of the light.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = light;
c = h.Color;
h.Style = 'local';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Color, Position, and Style

`Color` — Color of light

`[1 1 1]` (default) | RGB triplet | color string

Color of light emanating from the light object, specified as an RGB triplet or a color string. The default RGB triplet of `[1 1 1]` corresponds to white.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`

Long Name	Short Name	RGB Triplet
'black	'k'	[0 0 0]

Example: 'green'

Position — Location of light source

[1 0 1] (default) | three-element vector of the form [x y z]

Location of light source, specified as a three-element vector of the form [x y z]. Define the vector elements in data units from the axes origin to the (*x*, *y*, *z*) coordinate. The actual location of the light depends on the value of the Style property.

Example: [-40 -4 140]

Style — Type of light source

'infinite' (default) | 'local'

Type of light source, specified as one of these values:

- 'infinite' — Place the light at infinity. Use the Position property to specify the direction from which the light shines in parallel rays.
- 'local' — Place the light at the location specified by the Position property. The light is a point source that radiates from the location in all directions.

Visibility

Visible — Visibility of light from light source

'on' (default) | 'off'

Visibility of light from light source, specified as 'on' or 'off'.

Identifiers

Type — Type of graphics object

'light'

Type of graphics object, returned as the string 'light'. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using findobj.

`Tag` — Tag to associate with light

`''` (default) | string

Tag to associate with the light, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with light

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the light object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent of light

axes object | group object | transform object

Parent of light, specified as an axes, group, or transform object.

`Children` — Children of light

empty `GraphicsPlaceholder` array

The light has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of light object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The light object handle is always visible.

- `'off'` — The light object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The light object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the light at the command-line, but allows callback functions to access it.

If the light object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the light. Setting the `CreateFcn` property on an existing light has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during light creation. MATLAB executes the callback after creating the light and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The light object — You can access properties of the light object from within the callback function. You also can access the light object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.

- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the light. MATLAB executes the callback before destroying the light so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The light object — You can access properties of the light object from within the callback function. You also can access the light object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of light

`'off'` (default) | `'on'`

Deletion status of light, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the light begins execution (see the

DeleteFcn property). The BeingDeleted property remains set to `'on'` until the light no longer exists.

Check the value of the BeingDeleted property to verify that the light is not about to be deleted before querying or modifying it.

Unused Properties

`ButtonDownFcn` — (unused) Mouse-click callback
`''` (default) | function handle | cell array | string

Light objects do not use this property.

`UIContextMenu` — (unused) Context menu
uicontextmenu object

Light objects do not use this property.

`Selected` — (unused) Selection state
`'off'` (default) | `'on'`

Light objects do not use this property.

`SelectionHighlight` — (unused) Display of selection handles when selected
`'on'` (default) | `'off'`

Light objects do not use this property.

`PickableParts` — (unused) Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Light objects do not use this property.

`HitTest` — (unused) Response to captured mouse clicks
`'on'` (default) | `'off'`

Light objects do not use this property.

`Interruptible` — (unused) Callback interruption
`'on'` (default) | `'off'`

Light objects do not use this property.

`BusyAction` — (unused) Callback queuing
`'queue'` (default) | `'cancel'`

Light objects do not use this property.

See Also

`light`

More About

- "Access Property Values"
- "Graphics Object Properties"

lightangle

Create or position light object in spherical coordinates

Syntax

```
lightangle(az,el)
light_handle = lightangle(az,el)
lightangle(light_handle,az,el)
[az,el] = lightangle(light_handle)
```

Description

`lightangle(az,el)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `el` is the vertical elevation (both in degrees). The interpretation of azimuth and elevation is the same as that of the `view` command.

`light_handle = lightangle(az,el)` creates a light and returns the handle of the light in `light_handle`.

`lightangle(light_handle,az,el)` sets the position of the light specified by `light_handle`.

`[az,el] = lightangle(light_handle)` returns the azimuth and elevation of the light specified by `light_handle`.

Examples

```
surf(peaks)
axis vis3d
h = light;
for az = -50:10:50
   lightangle(h,az,30)
   pause(.2)
end
```

More About

Tips

By default, when a light is created, its style is `infinite`. If the light handle passed in to `lightangle` refers to a local light, the distance between the light and the camera target is preserved as the position is changed.

- "Lighting Overview"

See Also

`light` | `camlight` | `view`

Introduced before R2006a

lighting

Specify lighting algorithm

Syntax

```
lighting flat
lighting gouraud
lighting none
```

Description

`lighting` selects the algorithm used to calculate the effects of `light` objects on all `surface` and `patch` objects in the current axes. In order for the `lighting` command to have any effects, however, you must create a lighting object by using the `light` function.

`lighting flat` produces uniform lighting across each of the faces of the object. Select this method to view faceted objects.

`lighting gouraud` calculates the vertex normals and interpolates linearly across the faces. Select this method to view curved surfaces.

`lighting none` turns off lighting.

More About

Tips

The `surf`, `mesh`, `pcolor`, `fill`, `fill3`, `surface`, and `patch` functions create graphics objects that are affected by light sources. The `lighting` command sets the `FaceLighting` and `EdgeLighting` properties of surfaces and patches appropriately for the graphics object.

- "Lighting Overview"

See Also

`fill` | `fill3` | `light` | `material` | `mesh` | `patch` | `pcolor` | `shading` | `surface`

Introduced before R2006a

lin2mu

Convert linear audio signal to mu-law

Syntax

```
mu = lin2mu(y)
```

Description

`mu = lin2mu(y)` converts linear audio signal amplitudes in the range `-1 ≤ Y ≤ 1` to mu-law encoded "flints" in the range `0 ≤ u ≤ 255`.

See Also

`auwrite` | `mu2lin`

Introduced before R2006a

line

Create line object

Syntax

```
line
line(X,Y)
line(X,Y,Z)
line(X,Y,Z,'PropertyName',propertyvalue,...)
line('XData',x,'YData',y,'ZData',z,...)
h = line(...)
```

Properties

For a list of properties, see Primitive Line Properties.

Description

`line` creates a line object in the current axes with default values `x = [0 1]` and `y = [0 1]`. You can specify the color, width, line style, and marker type, as well as other characteristics.

The `line` function has two forms:

- Automatic color and line style cycling. When you specify multiple line coordinate data as a column array using the informal syntax (i.e., the first three arguments are interpreted as the coordinates),

  ```
  line(X,Y,Z)
  ```

  MATLAB cycles through the axes `ColorOrder` and `LineStyleOrder` property values the way the `plot` function does. However, unlike `plot`, `line` does not call the `newplot` function.
- Purely low-level behavior. When you call `line` with only property name/property value pairs,

```
line('XData',x,'YData',y,'ZData',z)
```

MATLAB draws a line object in the current axes using the default line color (see the `colordef` function for information on color defaults). Note that you cannot specify matrix coordinate data with the low-level form of the `line` function.

`line(X,Y)` adds the line defined in vectors `X` and `Y` to the current axes. If `X` and `Y` are matrices of the same size, `line` draws one line per column.

`line(X,Y,Z)` creates lines in three-dimensional coordinates.

`line(X,Y,Z,'`*`PropertyName`*`',propertyvalue,...)` creates a line using the values for the property name/property value pairs specified and default values for all other properties. For a description of the properties, see Primitive Line Properties.

`line('XData',x,'YData',y,'ZData',z,...)` creates a line in the current axes using the property values defined as arguments. This is the low-level form of the `line` function, which does not accept matrix coordinate data as the other informal forms described above.

`h = line(...)` returns a column vector of primitive line handles corresponding to each line object the function creates.

Examples

This example uses the `line` function to add a shadow to plotted data. First, plot some data and save the line's handle:

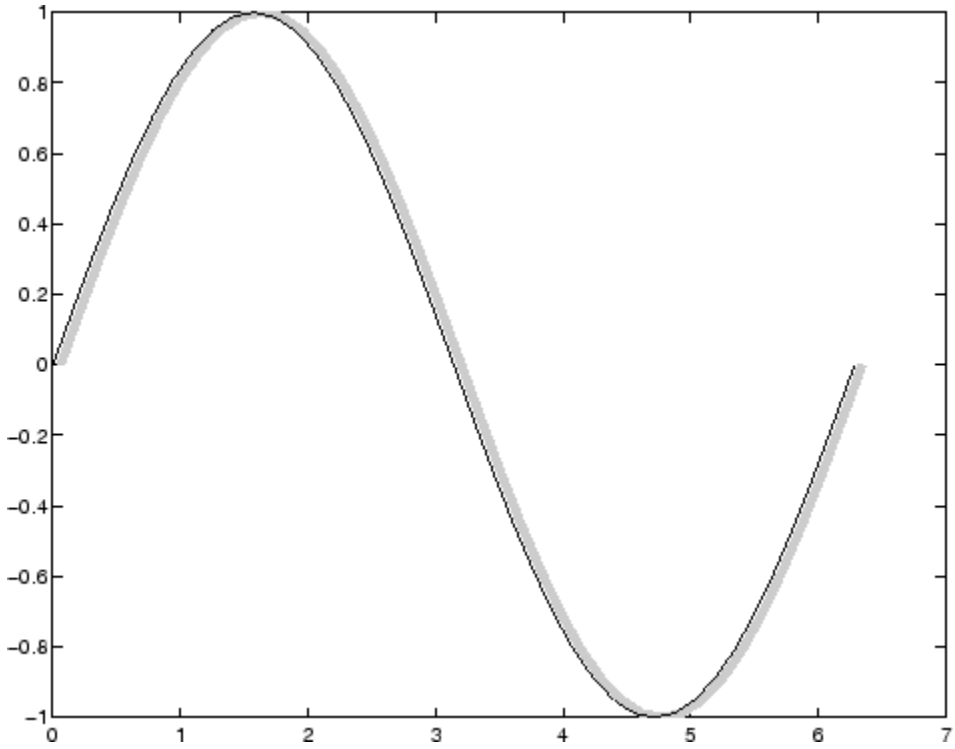
```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t),'k');
ax = gca;
```

Next, add a shadow by offsetting the *x*-coordinates. Make the shadow line light gray and wider than the default `LineWidth`:

```
hline2 = line(t+.06,sin(t),...
   'LineWidth',4,...
   'Color',[.8 .8 .8],...
   'Parent',ax);
```

Finally, pull the first line to the front:

```
set(gca,'Children',[hline1 hline2])
```



Drawing Lines Interactively

You can use the `ginput` function to select points from a figure. For example:

```
[x,y] = ginput(5);
line(x,y)
```

Drawing with mouse motion

You can use the axes `CurrentPoint` property and the figure `WindowButtonDownFcn` and `WindowButtonMotionFcn` properties to select a point with a mouse click and draw a line to another point by dragging the mouse, like a simple drawing program. The following example illustrates a few useful techniques for doing this type of interactive drawing.

Click to view in editor — This example enables you to click and drag the cursor to draw lines.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

Input Argument Dimensions — Informal Form

This statement reuses the one-column matrix specified for `ZData` to produce two lines, each having four points.

```
line(rand(4,2),rand(4,2),rand(4,1))
```

If all the data has the same number of columns and one row each, MATLAB transposes the matrices to produce data for plotting. For example,

```
line(rand(1,4),rand(1,4),rand(1,4))
```

is changed to

```
line(rand(4,1),rand(4,1),rand(4,1))
```

This also applies to the case when just one or two matrices have one row. For example, the statement

```
line(rand(2,4),rand(2,4),rand(1,4))
```

is equivalent to

```
line(rand(4,2),rand(4,2),rand(4,1))
```

More About

Tips

In its informal form, the `line` function interprets the first three arguments (two for 2-D) as the `X`, `Y`, and `Z` coordinate data, allowing you to omit the property names. You must specify all other properties as name/value pairs. For example,

```
line(X,Y,Z,'Color','r','LineWidth',4)
```

The low-level form of the `line` function can have arguments that are only property name/property value pairs. For example,

```
line('XData',x,'YData',y,'ZData',z,...
     'Color','r','LineWidth',4)
```

Line properties control various aspects of the line object. You can also set and query property values after creating the line using dot notation or the `set` and `get` functions

Unlike high-level functions such as `plot`, `line` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds line objects to the current axes. However, axes properties that are under automatic control, such as the axis limits, can change to accommodate the line within the current axes.

Connecting the dots

The coordinate data is interpreted as vectors of corresponding x, y, and z values:

```
X = [x(1) x(2) x(3)...x(n)]
Y = [y(1) y(2) y(3)...y(n)]
Z = [z(1) z(2) z(3)...z(n)]
```

where a point is determined by the corresponding vector elements:

```
p1(x(i),y(i),z(i))
```

For example, to draw a line from the point located at `x = .3` and `y = .4` and `z = 1` to the point located at `x = .7` and `y = .9` and `z = 1`, use the following data:

```
axis([0 1 0 1])
line([.3 .7],[.4 .9],[1 1],'Marker','.','LineStyle','-')
```

See Also

Functions

`annotation` | `axes` | `loglog` | `plot` | `plot3`

Properties

Primitive Line Properties

Introduced before R2006a

LineSpec (Line Specification)

Line specification string syntax

Description

Plotting functions accept *string specifiers* as arguments and modify the graph generated accordingly. Three components can be specified in the *string specifiers* along with the plotting command. They are:

- Line style
- Marker symbol
- Color

For example:

```
plot(x,y,'-.or')
```

plots `y` versus `x` using a dash-dot line (`-.`), places circular markers (`o`) at the data points, and colors both line and marker red (`r`). Specify the components (in any order) as a quoted string after the data arguments. Note that linespecs are single strings, not property-value pairs.

Plotting Data Points with No Line

If you specify a marker, but not a line style, only the markers are plotted. For example:

```
plot(x,y,'d')
```

Line Style Specifiers

You indicate the line styles, marker types, and colors you want to display using *string specifiers*, detailed in the following tables:

Specifier	LineStyle
`'-'`	Solid line (default)
`'--'`	Dashed line

Specifier	LineStyle
':'	Dotted line
'-.'	Dash-dot line

Marker Specifiers

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)

Note: The point (.) marker type does not change size when the specified value is less than 5.

Color Specifiers

Specifier	Color
r	Red
g	Green
b	Blue

Specifier	Color
`c`	Cyan
`m`	Magenta
`y`	Yellow
`k`	Black
`w`	White

Related Properties

This page also describes how to specify the properties of lines used for plotting. MATLAB graphics give you control over these visual characteristics:

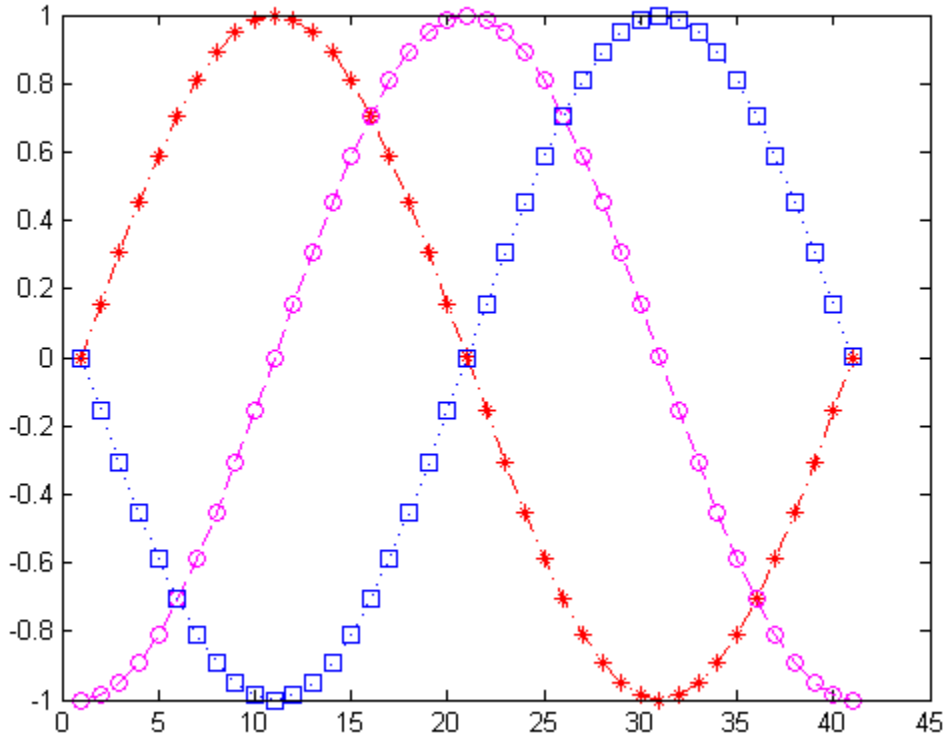
- `LineWidth` — Specifies the width (in points) of the line.
- `MarkerEdgeColor` — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` — Specifies the color of the face of filled markers.
- `MarkerSize` — Specifies the size of the marker in points (must be greater than 0).

In addition, you can specify the `LineStyle`, `Color`, and `Marker` properties instead of using the symbol string. This is useful if you want to specify a color that is not in the list by using RGB triplet values. See Chart Line Properties for details on these properties and `ColorSpec` for more information on color.

Examples

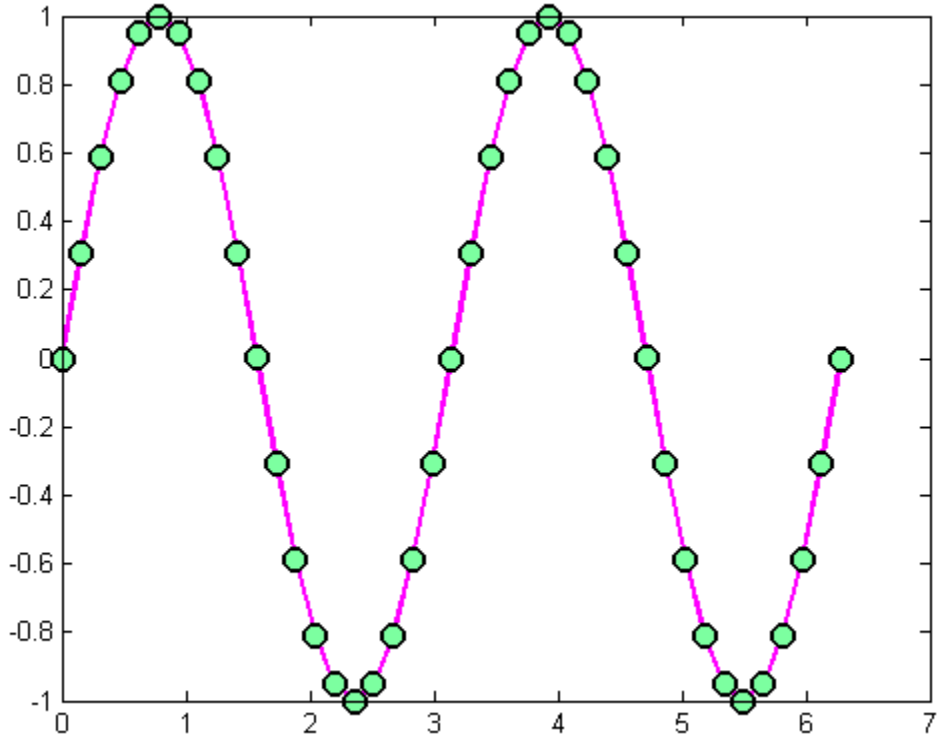
Plot the sine function over three different ranges using different line styles, colors, and markers.

```
figure
t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*')
hold on
plot(t,sin(t-pi/2),'--mo')
plot(t,sin(t-pi),':bs')
hold off
```



Create a plot illustrating how to set line properties:

```
figure
plot(t,sin(2*t),'-mo',...
                'LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor',[.49 1 .63],...
                'MarkerSize',10)
```



See Also

`axes` | `line` | `plot` | `patch` | `set` | `surface` | `ColorSpec`

linkaxes

Synchronize limits of specified 2-D axes

Syntax

```
linkaxes(axes_handles)
linkaxes(axes_handles,'option')
```

Description

Use `linkaxes` to synchronize the individual axis limits across several figures or subplots within a figure. Calling `linkaxes` makes all input axes have identical limits. Linking axes is best when you want to zoom or pan in one subplot and display the same range of data in another subplot.

Note: Use `linkaxes` for axes that remain in 2-D views.

`linkaxes(axes_handles)` links the *x*- and *y*-axis limits of the axes specified in the vector `axes_handles`. You can link any number of existing plots or subplots. The `axes_handles` input should be a vector of the handles for each plot or subplot. Entering an array of values results in an error message.

`linkaxes(axes_handles,'option')` links the axes' `axes_handles` according to the specified option. The `option` argument can be one of the following strings:

`x`	Link *x*-axis only.
`y`	Link *y*-axis only.
`xy`	Link *x*-axis and *y*-axis.
`off`	Remove linking.

See the `linkprop` function for more advanced capabilities that allow you to link object properties on any graphics object.

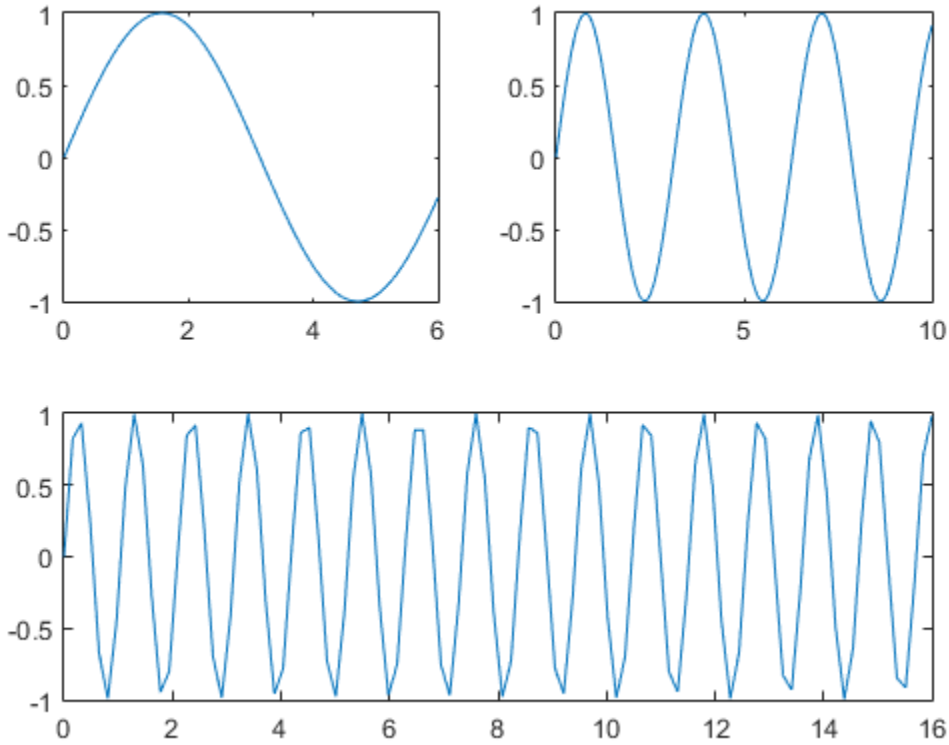
Examples

Create Subplots and Link Their Axes

Create three subplots and link their *x*-axes and *y*-axes so that they are always in sync.

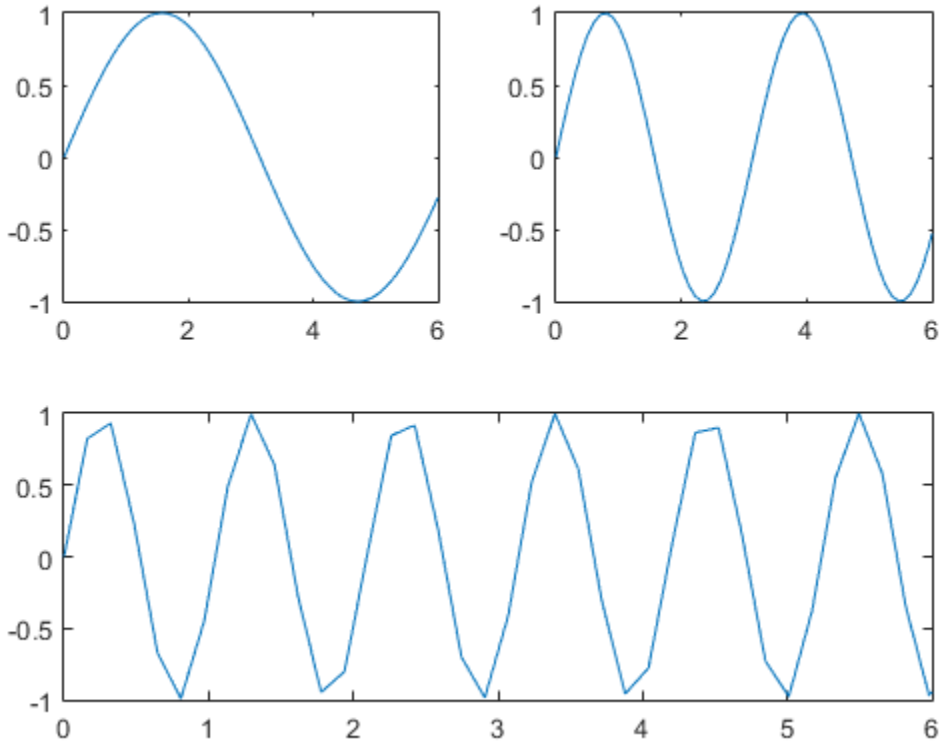
First, create a figure with three subplots. Plot data in each subplot.

```
figure
ax1 = subplot(2,2,1);
x1 = linspace(0,6);
y1 = sin(x1);
plot(x1,y1)

ax2 = subplot(2,2,2);
x2 = linspace(0,10);
y2 = sin(2*x2);
plot(x2,y2)

ax3 = subplot(2,2,[3,4]);
x3 = linspace(0,16);
y3 = sin(6*x3);
plot(x3,y3)
```



Now link the axes for the three subplots using the `linkaxes` function. Pass the string `'xy'` as an input argument to the function to link both the *x*-axes and *y*-axes. The `linkaxes` function uses the limits from the first axes passed to it.

```
linkaxes([ax1,ax2,ax3],'xy')
```



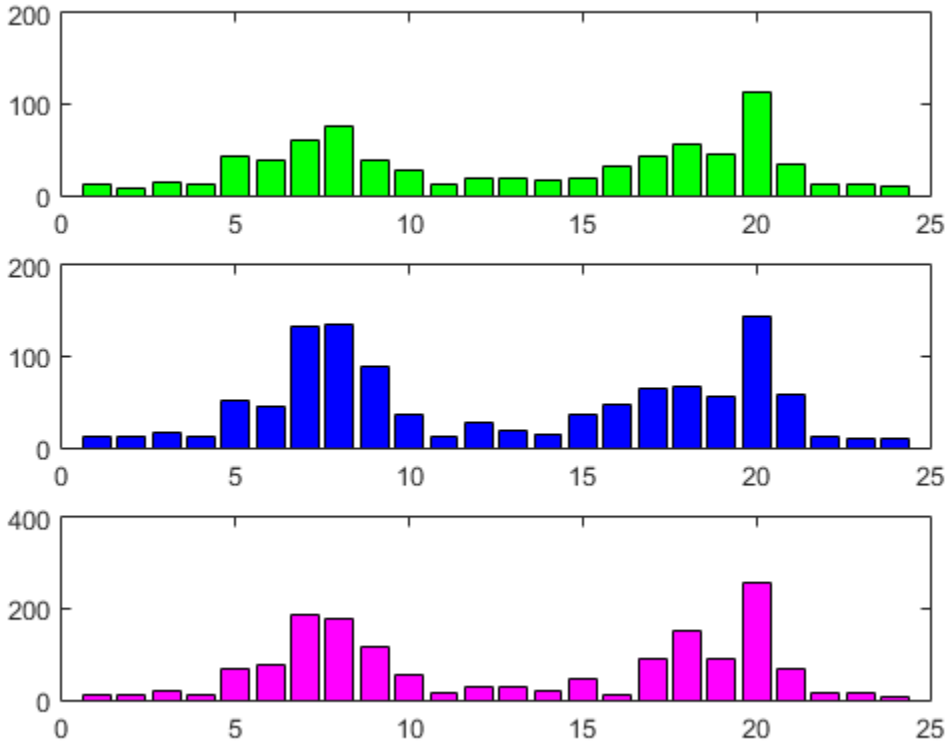
Panning or zooming into one of the subplots displays the same range of data in the other two subplots. To remove the linking, use `linkaxes([ax1,ax2,ax3]),'off'`.

Use Specific Subplot to Determine Limits for All Subplots

Load the `count.dat` data set which returns a three-column array named `count`. Create a new figure with three subplots and return the axes handles. In each subplot, create a bar graph.

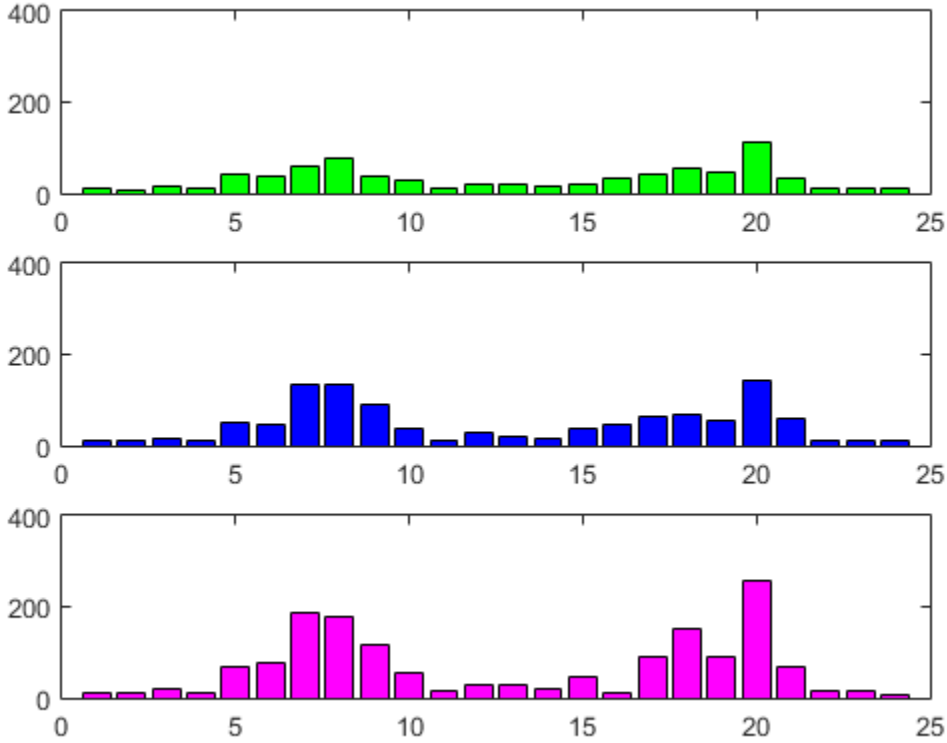
```
load count.dat

figure
ax1 = subplot(3,1,1);
bar(count(:,1),'g');
```

```
ax2 = subplot(3,1,2);
bar(count(:,2),'b');

ax3 = subplot(3,1,3);
bar(count(:,3),'m');
```



Link the *x*-axes and *y*-axes limits using `linkaxes` with the argument `'xy'`. Base the limits on the third subplot by passing its axes handle, `ax3`, as the first input to `linkaxes`.

```
linkaxes([ax3,ax2,ax1],'xy');
```



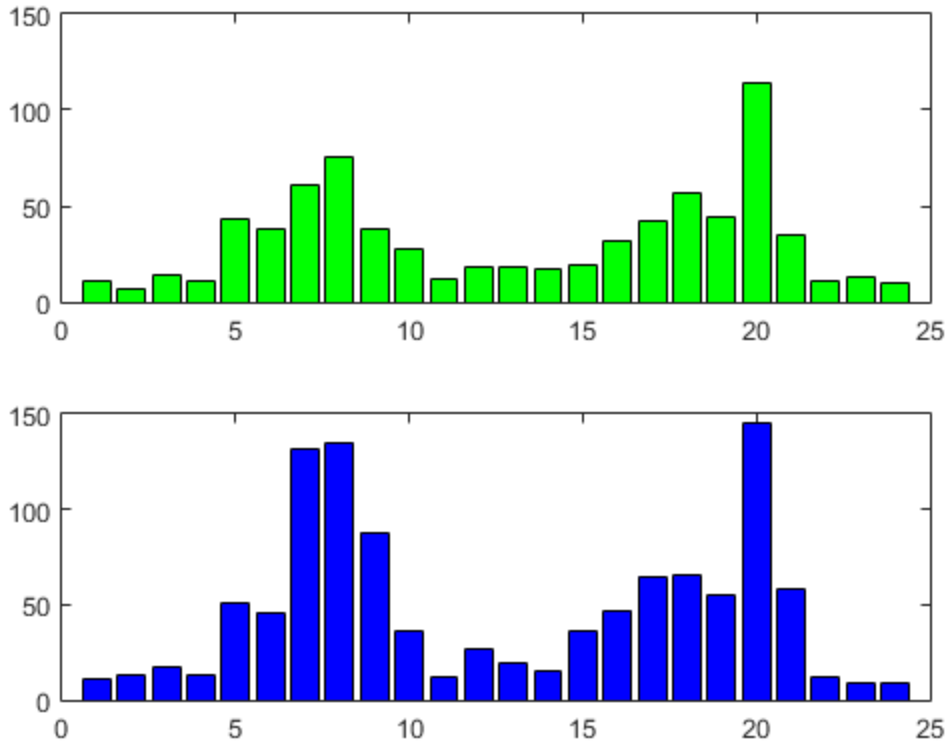
If you zoom in on one subplot, then the other two subplots behave in the same manner. To enable interactive zooming, use `zoom on`. To disable zooming, use `zoom off`.

Link x-Axes And Change Axis Limits

Load the `count.dat` data set which returns a three-column array named `count`. Create a new figure with two subplots and return the axes handles. In each subplot, create a bar graph.

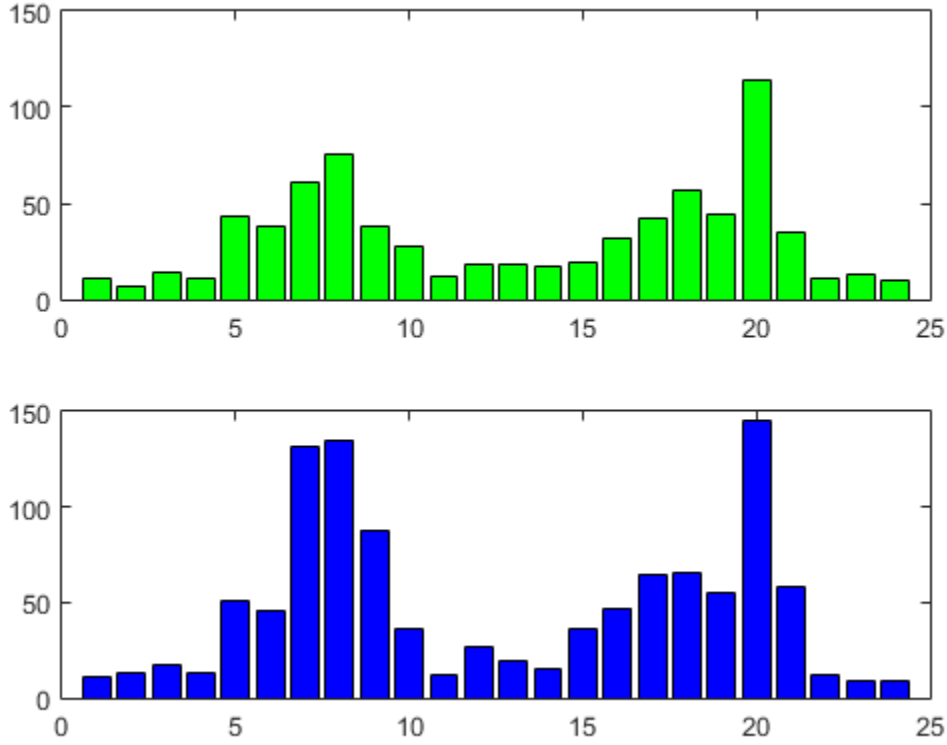
```
load count.dat
figure
ax1 = subplot(2,1,1);
bar(count(:,1),'g');
```

```
ax2 = subplot(2,1,2);
bar(count(:,2),'b');
```



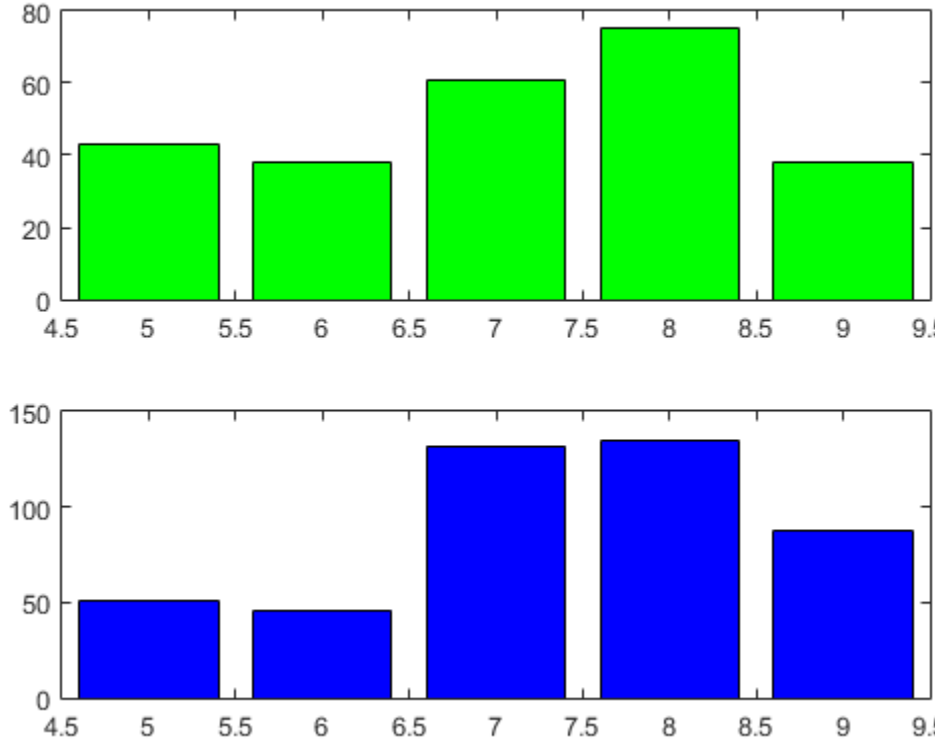
Link the *x*-axes for the two subplots.

```
linkaxes([ax1,ax2],'x');
```



Set the *x*-axis limits for the second subplot. Setting the limits overrides the limits initially set by `linkaxes`. Changing the *x*-axis limits effects both subplots.

```
ax2.XLim = [4.5,9.5];
```



If you pan either subplot, then both subplots pan uniformly in the *x*-direction, but only one subplot moves in the *y*-direction. To enable interactive panning, use `pan on`. To disable panning, use `pan off`.

More About

Tips

The first axes you supply to `linkaxes` determines the *x*- and *y*-limits for all linked axes. This can cause plots to partly or entirely disappear if their limits or scaling are very different. To override this behavior, after calling `linkaxes`, specify the limits of the axes that you want to control with the `set` command, as the third example illustrates.

Note: `linkaxes` is not designed to be transitive across multiple invocations. If you have three axes, `ax1`, `ax2`, and `ax3` and want to link them together, call `linkaxes` with `[ax1, ax2, ax3]` as the first argument. Linking `ax1` to `ax2`, then `ax2` to `ax3`, "unbinds" the `ax1`-`ax2` linkage.

See Also

`linkdata` | `linkprop` | `pan` | `zoom`

Introduced before R2006a

linkdata

Automatically update graphs when variables change

Syntax

```
linkdata on
linkdata off
linkdata
linkdata(figure_handle,...)
linkobj = linkdata(figure_handle)
```

Description

`linkdata on` turns on data linking for the current figure.

`linkdata off` turns data linking off.

`linkdata` by itself toggles the state of data linking.

`linkdata(figure_handle,...)` applies the function to the specified figure handle.

`linkobj = linkdata(figure_handle)` returns a *linkdata object* for the specified figure. The object has one read-only property, `Enable`, the string `'on'` or `'off'`, depending on the linked state of the figure.

Data linking connects graphs in figure windows to variables in the base or a function's workspace via their `XDataSource`, `YDataSource`, and `ZDataSource` properties. When you turn on data linking for a figure, MATLAB compares variables in the current (base or function caller) workspace with the `XData`, `YData`, and `ZData` properties of graphs in the affected figure to try to match them. When a match is found, the appropriate `XDataSource`, `YDataSource` and/or `ZDataSource` for the graph are set to strings that name the matching variables.

Any subsequent changes to linked variables are reflected in graphs that use them as data sources and in the Variables editor, if the linked variables are displayed there.

Conversely, any changes to plotted data values made at the command line, in the Variables editor, or with the Brush tool (such as deleting or replacing data points), are immediately reflected in the workspace variables linked to the data points.

When a figure containing graphs is linked and any variable identified as `XDataSource`, `YDataSource`, and/or `ZDataSource` changes its values in the workspace, all graphs displaying it in that and other linked figures automatically update. This operation is equivalent to automatically calling the `refreshdata` function on the corresponding figure when a variable changes.

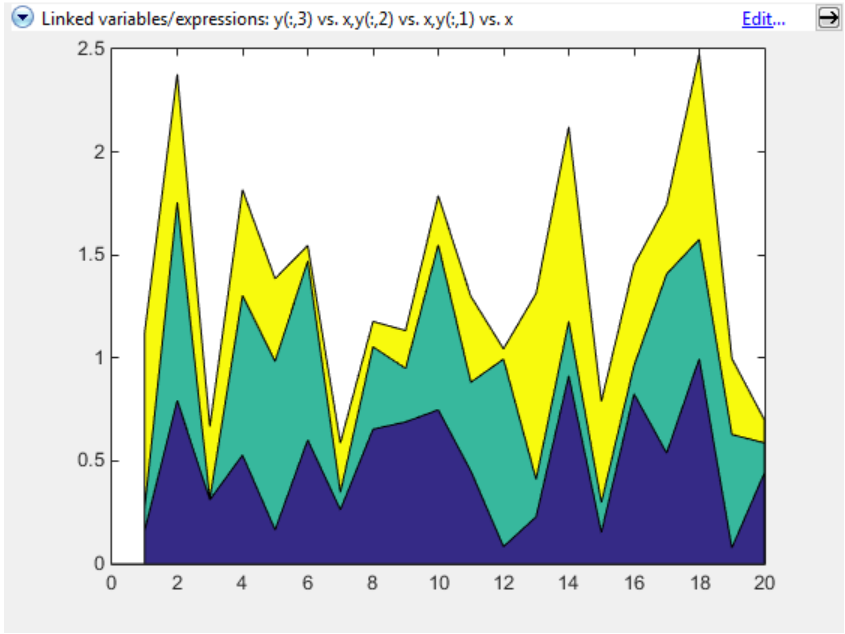
Linked figure windows identify themselves by the appearance of the Linked Plot information bar at the top of the window. When `linkdata` is `off` for a figure, the Linked Plot information bar is removed. If `linkdata` cannot unambiguously identify data sources for a graph in a linked figure, it reports this via the Linked Plot information bar, which gives the user an opportunity to identify data sources. The information bar displays a warning icon and a message, **No graphics have data sources** and also prompts **fix it**. Clicking **fix it** opens the Specify Data Source Properties dialog box for identifying variable names and ranges of data sources used in the graph.

Examples

Example 1

Create two variables, plot them as area charts, and link the plot to them:

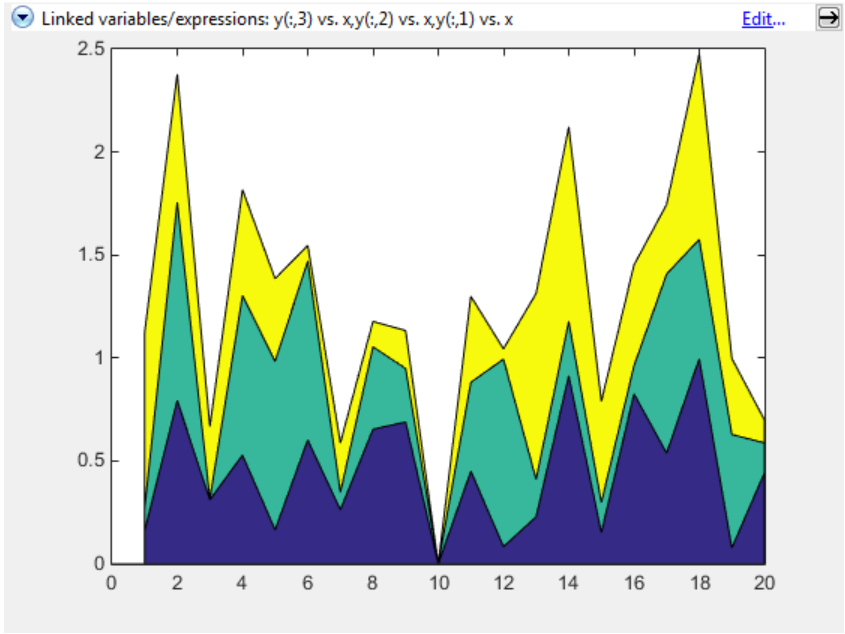
```
x = 1:20;
y = rand(20,3);
area(x,y)
linkdata on
```



Change values for linked variable `y` in the workspace:

```
y(10,:) = 0;
```

The area graph immediately updates.



Example 2

Delete a figure if it is not linked, based on a returned `linkdata` object:

```
fig = figure;
ld = linkdata(fig)
```

```
ld =
     graphics.linkdata
```

```
if strcmp(ld.Enable,'off')
    delete(fig)
end
```

Example 3

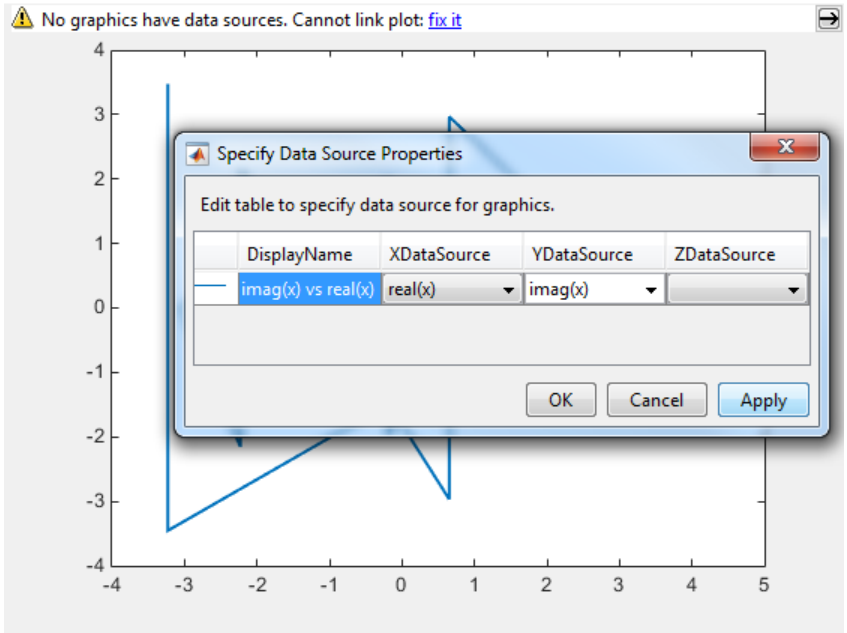
If a plotting function can display a complex variable, then you can link such plots. To do so, you need to describe the data sources as expressions to separate the real and imaginary parts of the variable. For example,

```
x = eig(randn(20,20));
whos
  Name        Size       Bytes  Class     Attributes
  x          20x1         320  double    complex
```

yields a complex vector. You can use `plot` to display the real portion as *x* and the imaginary portion as *y*, then link the graph to the variable:

```
plot(x)
linkdata
```

However, `linkdata` cannot unambiguously identify the graph's data sources, and you must tell it by typing `real(x)` and `imag(x)` into the Specify Data Source Properties dialog box that displays when you click **fix it** in the Linked Plot information bar.



To avoid having to type the data source names in the dialog box, you can specify them when you plot:

```
plot(x,'XDataSource','real(x)','YDataSource','imag(x)')
```

If you subsequently change values of `x` programmatically or manually, the plot updates accordingly.

Note: Although you can use data brushing on linked plots of complex data, your brush marks only appear in the plot you are brushing, not in other plots or in the Variables editor. This is because function calls, such as `real(x)` and `imag(x)`, that you specify as data sources are not interpreted when brushing graphed data.

More About

Tips

Types of Variables You Can Link

You can use `linkdata` to connect a graph with scalar, vector and matrix numeric variables of any class (including `complex`, if the graphing function can plot it) — essentially any data for which `isnumeric` equals `true`. See "Example 3" on page 1-4525 for instructions on linking complex variables. You can also link plots to numeric fields within structures. You can specify MATLAB expressions as data sources, for example, `sqrt(y)+1`.

Restoring Links that Break

Refreshing data on a linked plot fails if the strings in the `XDataSource`, `YDataSource`, or `ZDataSource` properties, when evaluated, are incompatible with what is in the current workspace, such that the corresponding `XData`, `YData`, or `ZData` are unable to respond. The visual appearance of the object in the graph is not affected by such failures, so graphic objects show no indication of broken links. Instead, a warning icon and the message **Failing links** appear on the Linked Plot information bar along with an **Edit** button that opens the Specify Data Sources dialog box.

Linking Rapidly Changing Data

`linkdata` buffers updates to data and dispatches them to plots at roughly half-second intervals. This makes data linking not suitable for smoothly animating changes in data

values unless they are updated in loops that are forced to execute two times per second or less.

One consequence of buffering link updates is that `linkdata` might not detect changes in data streams it monitors. If you are running a function that uses `assignin` or `evalin` to update workspace variables, `linkdata` can sometimes fail to process updates that change values but not the size and class of workspace variables. Such failures only happen when the function itself updates the plot.

Linking Brushed Graphs

If you link data sources to graphs that have been brushed, their brushing marks can change or vanish. This is because the workspace variables in those graphs now dictate which, if any, observations are brushed, superseding any brushing annotations that were applied to their graphical data (`YData`, etc.). For more details, see "How Data Linking Affects Data Brushing" in the `brush` reference page.

- "Making Graphs Responsive with Data Linking"

See Also

`brush` | `linkaxes` | `linkprop` | `refreshdata`

linkprop

Keep same value for corresponding properties of graphics objects

Syntax

```
hlink = linkprop(obj_handles,'PropertyName')
```

```
hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})
```

Description

Use `linkprop` to maintain the same values for the corresponding properties of different graphics objects.

Note: Use `linkprop` only with Handle Graphics objects.

`hlink = linkprop(obj_handles,'PropertyName')` maintains the same value for the property *`PropertyName`* on all objects whose handles appear in `obj_handles`. `linkprop` returns the link object in `hlink`. See “About Link Objects” on page 1-4530 for more information.

`hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})` maintains the same respective values for all properties passed as a cell array on all objects whose handles appear in `obj_handles`.

MATLAB updates the linked properties of all linked objects immediately when `linkprop` is called. The first object in the list `obj_handles` determines the property values for the other objects.

A set of graphics objects can have only one link object connecting their properties at any given time. Calling `linkprop` creates a new link object. This new link object replaces any existing link object that is associated with the objects specified in `obj_handles`. However, you can manage which properties and which objects are linked by calling methods on that object:

- To add an object to the list of linked objects, use the `addtarget` method.

- To link new properties of currently-linked objects, use the `addprop` method.
- To stop linking an object, use the `removetarget` method.
- To stop properties from linking, use the `removeprop` method.

About Link Objects

The link object that `linkprop` returns stores the mechanism that links the properties of different graphics objects. Therefore, the link object must exist within the context where you want property linking to occur (such as in the base workspace if users are to interact with the objects from the command line or figure tools).

The following list describes ways to maintain a reference to the link object.

- Return the link object as an output argument from a function and keep it in the base workspace while interacting with the linked objects.
- Make the `hlink` variable global.
- Store the `hlink` variable in an object's `UserData` property or in application data.

Updating a Link Object

If you want to change either the graphics objects or the properties that are linked, you need to use the link object methods designed for that purpose. These methods are functions that operate only on link objects. To use them, you must first create a link object using `linkprop`.

Method	Purpose
`addtarget`	Add specified graphics object to the link object's targets.
`removetarget`	Remove specified graphics object from the link object's targets.
`addprop`	Add specified property to the linked properties.
`removeprop`	Remove specified property from the linked properties.

Method Syntax

```
ddtarget(hlink,obj_handles)
```

```
removetarget(hlink,obj_handles)
addprop(hlink,'PropertyName')
removeprop(hlink,'PropertyName')
```

Method Arguments

- `hlink` — Link object returned by `linkprop`
- `obj_handles` — One or more graphic object handles
- *`PropertyName`* — Name of a property common to all target objects

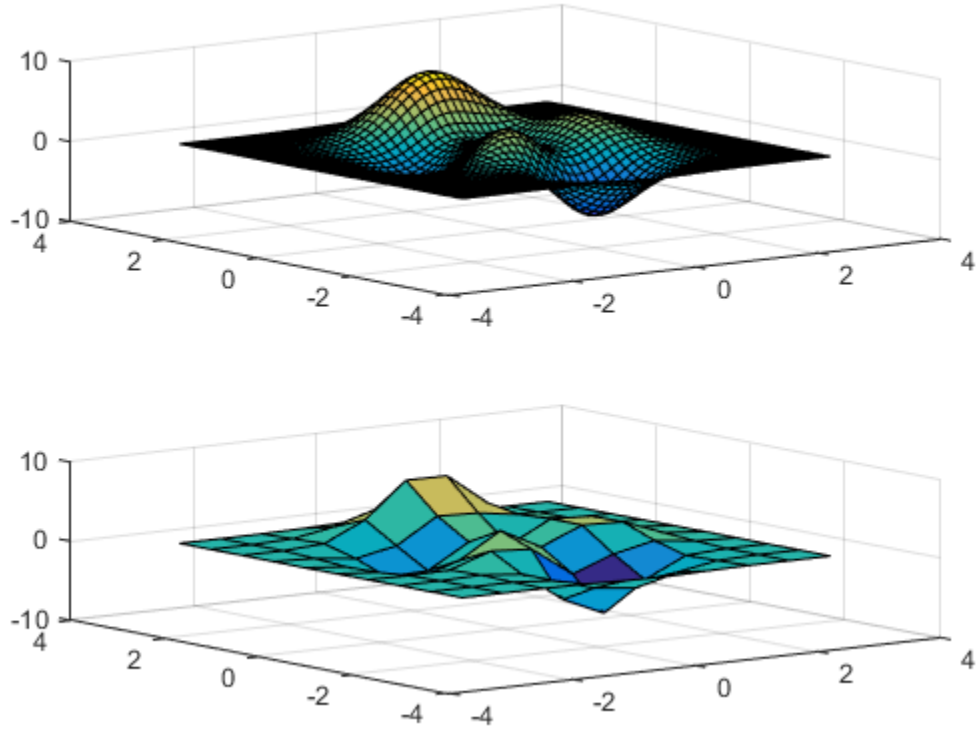
Examples

Link Axes So They Rotate Simultaneously

Link properties of two axes so that rotating one axes automatically rotates the other.

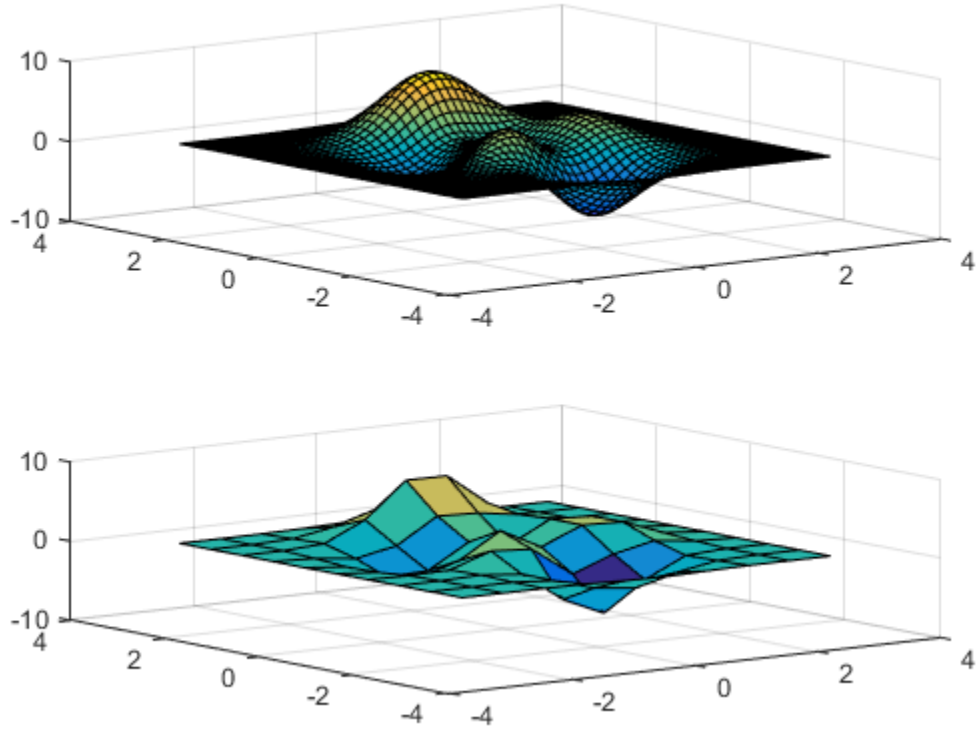
Create a figure with two axes and store the axes handles. Add plots to both axes.

```
figure
ax1 = subplot(2,1,1);
[X1,Y1,Z1] = peaks;
surf(X1,Y1,Z1)

ax2 = subplot(2,1,2);
[X2,Y2,Z2] = peaks(10);
surf(X2,Y2,Z2)
```



Link the `CameraPosition` and `CameraUpVector` properties of the axes and return the link object handle. Then, enable interactive rotation and use the mouse to rotate either axes. Rotating one axes automatically rotates the other in the same manner.

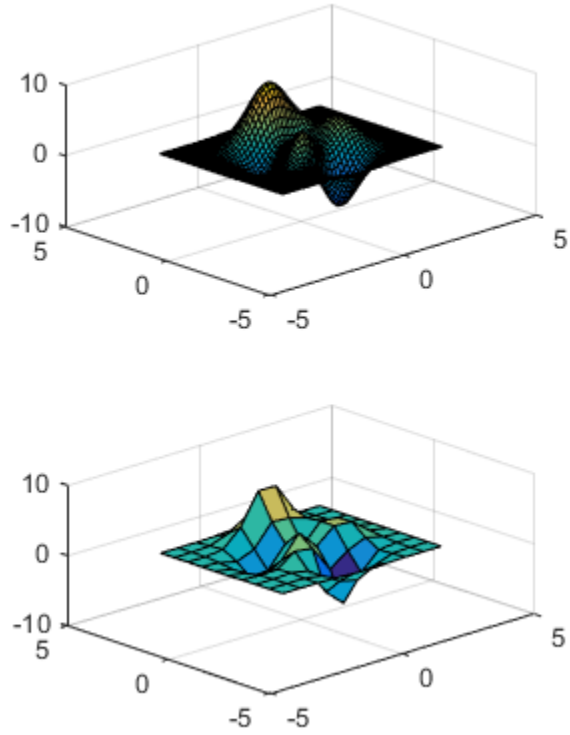
```
hlink = linkprop([ax1,ax2],{'CameraPosition','CameraUpVector'});
rotate3d on
```



To disable interactive rotation, use `rotate3d off`.

Link an additional property by passing the link object handle and the property name to `addprop`.

```
addprop(hlink,'PlotBoxAspectRatio')
```



See Also

`getappdata` | `linkdata` | `setappdata` | `ishghandle` | `linkaxes`

Introduced before R2006a

linsolve

Solve linear system of equations

Syntax

```
X = linsolve(A,B)
X = linsolve(A,B,opts)
```

Description

`X = linsolve(A,B)` solves the linear system `A*X = B` using LU factorization with partial pivoting when A is square and QR factorization with column pivoting otherwise. The number of rows of `A` must equal the number of rows of `B`. If `A` is m-by-n and `B` is m-by-k, then `X` is n-by-k. `linsolve` returns a warning if `A` is square and ill conditioned or if it is not square and rank deficient.

`[X, R] = linsolve(A,B)` suppresses these warnings and returns `R`, which is the reciprocal of the condition number of `A` if `A` is square, or the rank of `A` if `A` is not square.

`X = linsolve(A,B,opts)` solves the linear system `A*X = B` or `A'*X = B`, using the solver that is most appropriate given the properties of the matrix `A`, which you specify in `opts`. For example, if `A` is upper triangular, you can set `opts.UT = true` to make `linsolve` use a solver designed for upper triangular matrices. If `A` has the properties in `opts`, `linsolve` is faster than `mldivide`, because `linsolve` does not perform any tests to verify that `A` has the specified properties.

Notes If `A` does not have the properties that you specify in `opts`, `linsolve` returns incorrect results and does not return an error message. If you are not sure whether `A` has the specified properties, use `mldivide` instead.

For small problems, there is no speed benefit in using `linsolve` on triangular matrices as opposed to using the `mldivide` function.

The `TRANSA` field of the `opts` structure specifies the form of the linear system you want to solve:

- If you set `opts.TRANSA = false`, `linsolve(A,B,opts)` solves `A*X = B`.
- If you set `opts.TRANSA = true`, `linsolve(A,B,opts)` solves `A'*X = B`.

The following table lists all the field of `opts` and their corresponding matrix properties. The values of the fields of `opts` must be `logical` and the default value for all fields is `false`.

Field Name	Matrix Property
`LT`	Lower triangular
`UT`	Upper triangular
`UHESS`	Upper Hessenberg
`SYM`	Real symmetric or complex Hermitian
`POSDEF`	Positive definite
`RECT`	General rectangular
`TRANSA`	Conjugate transpose — specifies whether the function solves `A*X = B` or `A'*X = B`

The following table lists all combinations of field values in `opts` that are valid for `linsolve`. A true/false entry indicates that `linsolve` accepts either true or false.

LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
true	false	false	false	false	true/false	true/false
false	true	false	false	false	true/false	true/false
false	false	true	false	false	false	true/false
false	false	false	true	true/fals	false	true/false
false	false	false	false	false	true/false	true/false

Examples

The following code solves the system `A'x = b` for an upper triangular matrix `A` using both `mldivide` and `linsolve`.

```
A = triu(rand(5,3)); x = [1 1 1 0 0]'; b = A'*x;
y1 = (A')\b
```

```
opts.UT = true; opts.TRANSA = true;
y2 = linsolve(A,b,opts)

y1 =

    1.0000
    1.0000
    1.0000
         0
         0


y2 =

    1.0000
    1.0000
    1.0000
         0
         0
```

Note If you are working with matrices having different properties, it is useful to create an options structure for each type of matrix, such as `opts_sym`. This way you do not need to change the fields whenever you solve a system with a different type of matrix `A`.

See Also

`mldivide`

Introduced before R2006a

linspace

Generate linearly spaced vector

Syntax

```
y = linspace(x1,x2)
y = linspace(x1,x2,n)
```

Description

`y = linspace(x1,x2)` returns a row vector of 100 evenly spaced points between `x1` and `x2`.

`y = linspace(x1,x2,n)` generates `n` points. The spacing between the points is `(x2-x1)/(n-1)`.

`linspace` is similar to the colon operator, "`:`", but gives direct control over the number of points and always includes the endpoints. "`lin`" in the name "`linspace`" refers to generating linearly spaced values as opposed to the sibling function `logspace`, which generates logarithmically spaced values.

Examples

Vector of Evenly Spaced Numbers

Create a vector of 100 evenly spaced points in the interval `[-5,5]`.

```
y = linspace(-5,5);
```

Vector with Specified Number of Values

Create a vector of 7 evenly spaced points in the interval `[-5,5]`.

```
y1 = linspace(-5,5,7)
```

```
y1 =
```

```
   -5.0000   -3.3333   -1.6667         0    1.6667    3.3333    5.0000
```

Vector of Evenly Spaced Complex Numbers

Create a vector of complex numbers with 8 evenly spaced points between `1+2i` and `10+10i`.

```
y = linspace(1+2i,10+10i,8)
```

```
y =

  Columns 1 through 5

   1.0000 + 2.0000i   2.2857 + 3.1429i   3.5714 + 4.2857i   4.8571 + 5.4286i   6.1429 +

  Columns 6 through 8

   7.4286 + 7.7143i   8.7143 + 8.8571i  10.0000 +10.0000i
```

Input Arguments

`x1,x2` — Point interval

pair of numeric scalars

Point interval, specified as a pair of numeric scalars. `x1` and `x2` define the interval over which `linspace` generates points. `x1` and `x2` can be real or complex, and `x2` can be either larger or smaller than `x1`. If `x2` is smaller than `x1`, then the vector contains descending values.

Data Types: `single` | `double` | `datetime` | `duration`
Complex Number Support: Yes

`n` — Number of points

100 (default) | real numeric scalar

Number of points, specified as a real numeric scalar.

- If `n` is `1`, `linspace` returns `x2`.
- If `n` is zero or negative, `linspace` returns an empty 1-by-0 matrix.
- If `n` is not an integer, `linspace` rounds down and returns `floor(n)` points.

See Also

`colon` | `logspace`

Introduced before R2006a

RandStream.list

Random number generator algorithms

Class

`RandStream`

Syntax

```
RandStream.list
```

Description

`RandStream.list` lists all the generator algorithms that can be used when creating a random number stream with `RandStream` or `RandStream.create`. The available generator algorithms and their properties are given in the following table.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
mt19937ar	Mersenne twister (used by default stream at MATLAB startup)	No	$2^{19937}-1$
dsfmt19937	SIMD-oriented fast Mersenne twister	No	$2^{19937}-1$
mcg16807	Multiplicative congruential generator	No	$2^{31}-2$
mlfg6331_64	Multiplicative lagged Fibonacci generator	Yes	2^{124}
mrg32k3a	Combined multiple recursive generator	Yes	2^{127}

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
shr3cong	Shift-register generator summed with linear congruential generator	No	2^{64}
swb2712	Modified subtract with borrow generator	No	2^{1492}

See " Choosing a Random Number Generator" for details about these generator algorithms. See http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html for a full description of the Mersenne twister algorithm.

More About

- "Creating and Controlling a Random Number Stream"

listdlg

Create list-selection dialog box

Syntax

```
[Selection,ok] = listdlg(Name,Value,...)
```

Description

`[Selection,ok] = listdlg(Name,Value,...)` creates a modal dialog that allows the user to select one or more items from a list. The function returns two output arguments containing information about the items the user selected. The first output, `Selection`, contains the indices of the selected rows in the list. `Selection` is an empty array when no selection is made. The second output, `ok`, is 0 when no selection is made, or 1 when a selection is made.

The input arguments, `Name,Value`, are pairs of arguments that specify certain aspects of the dialog box. The pair, `'ListString',S` is required, but you can optionally specify any number of other pairs. This table lists all possible `Name,Value` input arguments:

Name	Value
`'ListString'` (required)	Cell array of strings that specify the list items.
`'SelectionMode'`	String indicating whether one or many items can be selected: `'single'` or `'multiple'` (the default).
`'ListSize'`	List box size in pixels, specified as a two-element vector `[width height]`. Default is `[160 300]`.
`'InitialValue'`	Vector of indices of the list box items that are initially selected. Default is 1, the first item.
`'Name'`	String for the dialog box's title. Default is an empty string.
`'PromptString'`	String matrix or cell array of strings that appears as text above the list box. Default is an empty cell array.

Name	Value
`'OKString'`	String for the OK button. Default is `'OK'`.
`'CancelString'`	String for the Cancel button. Default is `'Cancel'`.
`'uh'`	Uicontrol button height, in pixels. Default is 18.
`'fus'`	Frame/uicontrol spacing, in pixels. Default is 8.
`'ffs'`	Frame/figure spacing, in pixels. Default is 8.

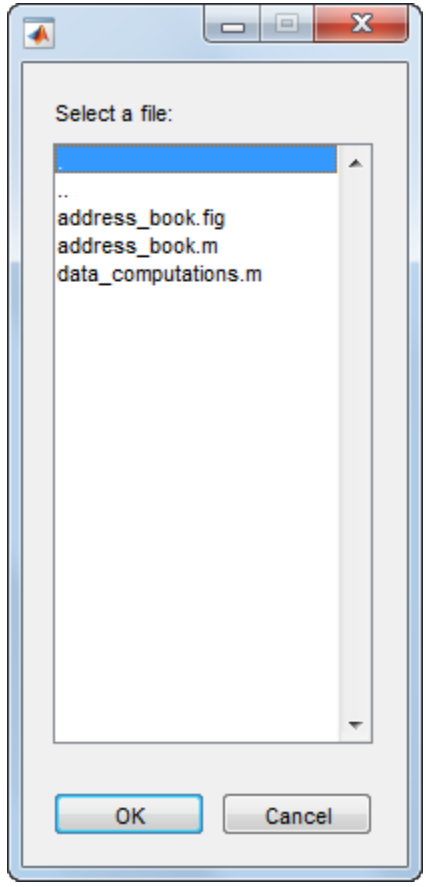
Double-clicking on an item or pressing **Return** when multiple items are selected has the same effect as clicking the **OK** button. The dialog box has a **Select all** button (when in multiple selection mode) that enables you to select all list items.

Examples

This example displays a dialog box that enables the user to select a file from the current directory.

The `listdlg` function returns two output arguments. The first output, `s`, is the index to the selected row in the list. The second output, `v`, is 0 when no selection is made, or 1 when a selection is made.

```
d = dir;
str = {d.name};
[s,v] = listdlg('PromptString','Select a file:',...
                'SelectionMode','single',...
                'ListString',str)
```



More About

Modal Dialog

A window that blocks interaction with other windows until the user closes the blocking window. For more information, see the figure `WindowStyle` property description.

See Also

`uigetfile` | `uigetdir` | `questdlg` | `uicontrol`

Introduced before R2006a

listfonts

List available system fonts

Syntax

```
c = listfonts
c = listfonts(h)
```

Description

`c = listfonts` returns sorted list of available system fonts.

`c = listfonts(h)` returns sorted list of available system fonts and includes the `FontName` property of the object with handle `h`.

Examples

Example 1

This example returns a list of available system fonts similar in format to the one shown.

```
list = listfonts
```

```
list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
    'ZapfChancery'
    'ZapfDingbats'
    'ZWAdobeF'
```

Example 2

This example returns a list of available system fonts with the value of the `FontName` property, for the object with handle `h`, included and sorted in the list.

```
 h = uicontrol('Style','text','String',...
     'My Font','FontName','MyFont');
 list = listfonts(h)
```

```
list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
    'MyFont'
    ...
    'ZapfChancery'
    'ZapfDingbats'
    'ZWAdobeF'
```

More About

Tips

Calling `listfonts` returns a list of all fonts on your system, possibly including fonts that you cannot use with MATLAB. Consider using the `uisetfont` function to preview fonts to that MATLAB can render in figure windows. Like `uisetfont`, the **Custom Fonts** pane of MATLAB Preferences also previews available fonts and only shows those that MATLAB can render.

See Also

`uisetfont`

load

Load variables from file into workspace

Syntax

```
load(filename)
load(filename,variables)
load(filename,'-ascii')
load(filename,'-mat')
load(filename,'-mat',variables)

S = load( ___ )

load filename
```

Description

`load(filename)` loads data from `filename`.

- If `filename` is a MAT-file, then `load(filename)` loads variables in the MAT-File into the MATLAB workspace.
- If `filename` is an ASCII file, then `load(filename)` creates a double-precision array containing data from the file.

`load(filename,variables)` loads the specified variables from the MAT-file, `filename`.

`load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`load(filename,'-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`load(filename,'-mat',variables)` loads the specified variables from `filename`.

`S = load( ___ )` loads data into `S`, using any of the input arguments in the previous syntax group.

- If `filename` is a MAT-file, then `S` is a structure array.
- If `filename` is an ASCII file, then `S` is a double-precision array containing data from the file.

`load filename` is the command form of the syntax. Command form requires fewer special characters. You do not need to type parentheses or enclose input strings in single quotes. Separate inputs with spaces instead of commas.

For example, to load a file named `durer.mat`, these statements are equivalent:

```
load durer.mat      % command form
load('durer.mat')   % function form
```

You can include any of the inputs described in previous syntaxes. For example, to load the variable named `X`:

```
load durer.mat X       % command form
load('durer.mat','X')  % function form
```

Do not use command form when any of the inputs, such as `filename`, are variables.

Examples

Load All Variables from MAT-File

Load all variables from the example MAT-file, `gong.mat`. Check the contents of the workspace before and after the load operation.

```
disp('Contents of workspace before loading file:')
whos

disp('Contents of gong.mat:')
whos('-file','gong.mat')

load('gong.mat')
disp('Contents of workspace after loading file:')
whos
```

You also can use command syntax to load the variables. Clear the previously loaded variables and repeat the `load` operation.

```
clear y Fs
```

```
load gong.mat
```

Load Specific Variable From MAT-File

Load only variable `y` from example file `handel.mat`. If the workspace already contains variable `y`, the `load` operation overwrites it with data from the file.

```
load('handel.mat','y')
```

You also can use command syntax to load the variable, `y`.

```
load handel.mat y
```

Use Regular Expressions to Load Specific Variables

View the contents of the example file, `accidents.mat`.

```
whos -file accidents.mat
```

```
 Name              Size             Bytes  Class     Attributes

  datasources       3x1              2724  cell
  hwycols           1x1                 8  double
  hwydata          51x17             6936  double
  hwyheaders        1x17              2758  cell
  hwyidx           51x1                408  double
  hwyrows           1x1                 8  double
  statelabel       51x1               6596  cell
  ushwydata         1x17               136  double
  uslabel           1x1                138  cell
```

Use function syntax to load all variables with names not beginning with `'hwy'`, from the file.

```
load('accidents.mat', '-regexp', '^(?!hwy)...')
```

Alternatively, use command syntax to load the same variables.

```
load accidents.mat -regexp '^(?!hwy)...'
```

Load List of Variables into Structure Array

The file, `durer.mat`, contains variables `X`, `caption`, and `map`. Create a cell array of variable names to load.

```
filename = 'durer.mat';
```

```
myVars = {'X','caption'};
S = load(filename,myVars{:})

S =

          X: [648x509 double]
    caption: [2x28 char]
```

Only the variables `X` and `caption` are loaded into the structure array, `S`.

Load ASCII File

Create an ASCII file from several 4-column matrices, and load the data back into a double-precision array.

```
a = magic(4);
b = ones(2, 4) * -5.7;
c = [8 6 4 2];
save -ascii mydata.dat a b c
clear a b c

load mydata.dat -ascii
```

`load` creates an array of type `double` named `mydata`.

View information about `mydata`.

```
whos mydata

Name        Size            Bytes  Class     Attributes

mydata      7x4               224  double
```

Input Arguments

`filename` — Name of file

`matlab.mat` (default) | string

Name of file, specified as a string. If you do not specify `filename`, the `load` function searches for a file named `matlab.mat`.

`filename` can include a file extension and a full or partial path. If `filename` has no extension (that is, no text after a period), `load` looks for a file named `filename.mat`. If

`filename` has an extension other than `.mat`, the `load` function treats the file as ASCII data.

When using the command form of `load`, it is unnecessary to enclose input strings in single quotes. However, if `filename` contains a space, you must enclose the argument in single quotes. For example, `load 'filename withspace.mat'`.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, `%`).

Example: `'myFile.mat'`

Data Types: `char`

`variables` — Names of variables to load

string

Names of variables to load, specified as one or more strings. When using the command form of `load`, you do not need to enclose input strings in single quotes. `variables` can be in one of the following forms.

Form of `variables` Input	Variables to Load
`var1,...,varN`	Load the listed variables, specified as individual strings. Use the `'*'` wildcard to match patterns. For example, `load('filename.mat','A*')` or `load filename.mat A*` loads all variables in the file whose names start with `A`.
`'-regexp',`*`expr1,...,exprN`*	Load only the variables or fields whose names match the regular expressions, specified as strings. For example, `load('filename.mat','-regexp','^Mon','^Tues')` or `load filename.mat -regexp ^Mon ^Tues` loads only the variables in the file whose names begin with `Mon` or `Tues`.

Data Types: `char`

Output Arguments

S — Loaded variables or data

structure array | m-by-n array

Loaded variables, returned as a structure array, if `filename` is a MAT-File.

Loaded data, returned as an `m`-by-`n` array of type `double`, if `filename` is an ASCII file. `m` is equal to the number of lines in the file, and `n` is equal to the number of values on a line.

More About

Algorithms

If you do not specify an output for the `load` function, MATLAB creates a variable named after the loaded file (minus any file extension). For example, the command

```
load mydata.dat
```

reads data into a variable called `mydata`.

To create the variable name, `load` precedes any leading underscores or digits in `filename` with an `X` and replaces any other nonalphabetic characters with underscores. For example, the command

```
load 10-May-data.dat
```

creates a variable called `X10_May_data`.

- "Supported File Formats for Import and Export"
- "Save, Load, and Delete Workspace Variables"
- "Ways to Import Text Files"
- "Loading Variables within a Function"
- "Import or Export a Sequence of Files"
- "Command vs. Function Syntax"

See Also

`clear` | `importdata` | `matfile` | `regexp` | `save` | `uiimport` | `whos`

Introduced before R2006a

load (COM)

Initialize control object from file

Syntax

```
load(h,'filename')
```

Description

`load(h,'filename')` initializes the COM object associated with the interface represented by the MATLAB COM object `h` from file specified in the string `filename`. The file must have been created previously by serializing an instance of the same control.

COM functions are available on Microsoft Windows systems only.

Note The COM `load` function is only supported for controls at this time.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f);
save(h,'mwsample')
```

Now, alter the figure by changing its label and the radius of the circle.

```
h.Label = 'Circle';
h.Radius = 50;
Redraw(h);
```

Using the `load` function, you can restore the control to its original state.

```
load(h,'mwsample');
get(h)

ans =
     Label: 'Label'
```

```
Radius: 20
```

See Also

save (COM) | delete (COM) | actxcontrol | actxserver | release

Introduced before R2006a

load (serial)

Load serial port objects and variables into MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2 ...
out = load('filename','obj1','obj2',...)
```

Description

`load filename` returns all variables from the file specified by `filename` into the MATLAB workspace.

`load filename obj1 obj2 ...` returns the serial port objects specified by `obj1 obj2 ...` from the file `filename` into the MATLAB workspace.

`out = load('filename','obj1','obj2',...)` returns the specified serial port objects from the file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded serial port objects.

Examples

Note: This example is based on a Windows platform.

Suppose that you create the serial port objects `s1` and `s2`, configure a few properties for `s1`, and connect both objects to their instruments:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s1,'Parity','mark','DataBits',7);
fopen(s1);
fopen(s2);
```

Save `s1` and `s2` to the file `MyObject.mat`, and then load the objects back into the workspace:

```
save MyObject s1 s2;
load MyObject s1;
load MyObject s2;

get(s1, {'Parity', 'DataBits'})

ans =
    'mark'    [7]

get(s2, {'Parity', 'DataBits'})

ans =
    'none'    [8]
```

More About

Tips

Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

See Also

`save` | `Status`

Introduced before R2006a

loadlibrary

Load C/C++ shared library into MATLAB

Syntax

```
loadlibrary(libname,hfile)
loadlibrary(libname)
loadlibrary(libname,hfile,Name,Value)

loadlibrary(libname,@protofile)

[notfound,warnings] = loadlibrary( ___ )
```

Description

`loadlibrary(libname,hfile)` loads functions from shared library, libname, defined in header file, hfile, into MATLAB.

`loadlibrary(libname)` loads the library if the name of the header file is the same as the name of the library file.

`loadlibrary(libname,hfile,Name,Value)` loads the library with one or more Name,Value arguments.

`loadlibrary(libname,@protofile)` uses a prototype file, `protofile`, in place of a header file.

`[notfound,warnings] = loadlibrary( ___ )` returns warning information, and can include any of the input arguments in previous syntaxes.

Examples

Load Functions in shrlibsample Library

Add path to `examples` folder.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Load `shrlibsample`.

```
loadlibrary('shrlibsample')
```

Cleanup.

```
unloadlibrary shrlibsample
```

Load Library Using Header File

The header file for the `libmx` library is `matrix.h`.

```
hfile = fullfile(matlabroot,'extern','include','matrix.h');
loadlibrary('libmx',hfile)
```

Cleanup.

```
unloadlibrary libmx
```

Load Library Using Multiple Header Files

Suppose that you have a library, `mylib`, with the header file, `mylib.h`. The header file contains the statement, `#include header2.h`. To use functions defined in `header2.h`, call `loadlibrary` with the `addheader` option.

```
loadlibrary('mylib','mylib.h','addheader','header2')
```

Load Library Using an Alias Name

Create an alias, `lib`, for library, `shrlibsample`.

```
loadlibrary('shrlibsample','shrlibsample.h','alias','lib')
```

Use the alias name, `lib`, to call a function in the library.

```
str = 'This was a Mixed Case string';
calllib('lib','stringToUpper',str)
```

```
ans =
   THIS WAS A MIXED CASE STRING
```

Cleanup.

```
unloadlibrary lib
```

Search Alternative Paths for Header Files

Add path to folder containing `shrlibsample` and its header file, `shrlibsample.h`.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

The `shrlibsample.h` header file includes the header file, `shrhelp.h`. If `shrhelp.h` is in a different folder, for example, `c:\work`, use the `'includepath'` option to tell MATLAB where to find the file.

```
loadlibrary('shrlibsample','shrlibsample.h','includepath','c:\work')
```

Cleanup.

```
unloadlibrary shrlibsample
```

Create Alias Name for MATLAB mxGetNumberOfDimensions Function

This example shows how to replace the `mxGetNumberOfDimensions` function name in the MATLAB Matrix Library API with a shorter alias name, `mxGetDims`. Use a prototype file to define the alias name, then load the library using the prototype file as the header file.

Use a folder for which you have write-access.

```
cd('c:\work')
```

Create a prototype file, `mxproto.m`.

```
hfile = fullfile(matlabroot,'extern','include','matrix.h');
loadlibrary('libmx',hfile,'mfilename','mxproto')
```

MATLAB creates the prototype file in the current folder.

Add the alias name to the prototype file. Open the file in MATLAB Editor.

```
edit mxproto.m
```

Search for the function `mxGetNumberOfDimensions`.

The following command assigns the alias `mxGetDims`.

```
fcns.alias{fcnNum}='mxGetDims';
```

Add the command to the line before the command to increment `fcnNum`. The new function prototype, with the new command shown in bold, looks like this:

```
fcns.name{fcnNum}='mxGetNumberOfDimensions';
fcns.calltype{fcnNum}='cdecl';
fcns.LHS{fcnNum}='int32';
fcns.RHS{fcnNum}={'MATLAB array'};
fcns.alias{fcnNum}='mxGetDims'; % Alias defined
fcnNum=fcnNum+1; % Increment fcnNum
```

Reload `libmx` using the prototype file.

```
unloadlibrary libmx
loadlibrary('libmx',@mxproto)
```

Call the function by its alias name.

```
y = rand(4,7,2);
calllib('libmx','mxGetDims',y)

ans =
     3
```

Input Arguments

`libname` — Name of shared library

string

Name of shared library, specified as a string. The name is case-sensitive and must match the file on your system.

On Microsoft Windows systems, `libname` refers to the name of a shared library (`.dll`) file. On Linux systems, it refers to the name of a shared object (`.so`) file. On Apple Macintosh systems, it refers to a dynamic shared library (`.dylib`). If you do not include a file extension with the `libname` argument, `loadlibrary` attempts to find the library with either the appropriate platform MEX-file extension or the appropriate platform library extension. For a list of MEX-file extensions, use `mexext`.

MATLAB extracts the name portion of the `libname` string to identify the library in other shared library functions. For example, when you call the `calllib` function, do not include the path or file extension in the library argument name.

Data Types: char

hfile — Name of C header file
string

Name of C header file, specified as a string. The name is case-sensitive and must match the file on your system. If you do not include a file extension in the file name, loadlibrary uses .h for the extension.

Data Types: char

protofile — Name of prototype file
string

Name of prototype file, specified as a string. The name is case-sensitive and must match the file on your system. The string @protofile specifies a function handle to the prototype file. When using a prototype file, the only valid Name,Value pair argument is **alias**.

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example:

'addheader' — Header file
string

Header file, specified as the comma-separated pair consisting of 'addheader' and a string. Specify the file name without a file extension.

Each file specified by addheader must have a corresponding #include statement in the base header file. To load only the functions defined in the header file that you want to use in MATLAB, use addheader.

MATLAB does not verify the existence of header files and ignores any that are not needed.

`'alias'` — Alternative name for library

string

Alternative name for library, specified as the comma-separated pair consisting of `'alias'` and a string. Associates the specified name with the library. All subsequent calls to MATLAB functions that reference this library must use this alias until the library is unloaded.

`'includepath'` — More search path for subordinate header files

string

more search path for subordinate header files—header files within header files, specified as the comma-separated pair consisting of `'includepath'` and a string.

`'mfilename'` — Prototype file

string

Prototype file, specified as the comma-separated pair consisting of `'mfilename'` and a string. Generates a prototype file in the current folder. The prototype file name must be different from the library name. Use this file in place of a header file when loading the library.

`'thunkfilename'` — Thunk file

string

Thunk file, specified as the comma-separated pair consisting of `'thunkfilename'` and a string. Overrides the default thunk file name.

Output Arguments

`notfound` — Names of functions

cell array

Names of functions found in header files but missing from the library, returned as cell array.

Data Types: `cell`

`warnings` — Warnings

character array

Warnings produced while processing the header file, returned as character array.

Limitations

- You must have a supported C compiler and Perl must be available.
- Do not call `loadlibrary` if the library is already in memory. To test this condition, call `libisloaded`.
- `loadlibrary` does not support libraries generated by the MATLAB Compiler product.
- The MATLAB Shared Library interface does not support library functions with function pointer inputs.
- For more information, see "Limitations to Shared Library Support".

More About

Prototype File

A prototype file is a file of MATLAB commands which you can modify and use in place of a header file.

Thunk File

A thunk file is a compatibility layer to a 64-bit library generated by MATLAB. The name of the thunk file is `BASENAME_thunk_COMPUTER.c` where *`BASENAME`* is either the name of the shared library or, if specified, the mfilename prototype name. *`COMPUTER`* is the string returned by the `computer` function.

MATLAB compiles this file and creates the file `BASENAME_thunk_COMPUTER.LIBEXT`, where *`LIBEXT`* is the platform-dependent default shared library extension, for example, `dll` on Windows.

Tips

- If you have more than one library file of the same name, load the first using the library file name. Then load the additional libraries using the **`alias`** option.
- Use the **`alias`** option as an alternate name for a library. Use the @protofile argument to load an alternate header file.

- “When to Use a Prototype File”
- Supported and Compatible Compilers

See Also

`calllib` | `computer` | `libfunctions` | `libisloaded` | `mex` | `mexext` | `unloadlibrary`

Introduced before R2006a

loadobj

Modify load process for object

Syntax

```
b = loadobj(a)
```

Description

`b = loadobj(a)` is called by the `load` function if the class of `a` defines a `loadobj` method. `load` returns `b` as the value loaded from a MAT-file.

Define a `loadobj` method when objects of the class require special processing when loaded from MAT-files. If you define a `saveobj` method, then define a `loadobj` method to restore the object to the desired state. Define `loadobj` as a static method so it can accept as an argument whatever object or structure that you saved in the MAT-file.

When loading a subclass object, `load` calls only the subclass `loadobj` method. If a superclass defines a `loadobj` method, the subclass inherits this method. However, it is possible that the inherited method does not perform the necessary operations to load the subclass object. Consider overriding superclass `loadobj` methods.

If any superclass in a class hierarchy defines a `loadobj` method, then the subclass `loadobj` method must ensure that the subclass and superclass objects load properly. Ensure proper loading by calling the superclass `loadobj` (or other methods) from the subclass `loadobj` method.

Input Arguments

a

The input argument, `a`, can be:

- The object as loaded from the MAT-file.
- A structure created by `load` (if `load` cannot resolve the object).

- A structure returned by the `saveobj` method that was saved instead of the object.

Implement your `loadobj` method to work with scalar objects or structures. When you have saved an object array, `load` calls `loadobj` on each element of the saved array.

See Also

`load` | `save` | `saveobj`

Tutorials

- "Save and Load"

Introduced before R2006a

localfunctions

Function handles to all local functions in MATLAB file

Syntax

```
fcns = localfunctions
```

Description

`fcns = localfunctions` returns a cell array of function handles, `fcns`, to all local functions in the current file.

You cannot define local functions in the context of the command line, scripts, or anonymous functions, so when you call `localfunctions` from these contexts, you get an empty cell array. Within the cell array, `localfunctions` returns the function handles in an undefined order.

Examples

Display Handles to Local Functions in File

Create a new file, `fileWithLocalFunctions.m`, in your MATLAB path. In the main function, call and display the results of `localfunctions`. In the same file, create two local functions.

```
function fileWithLocalFunctions

fcns = localfunctions;
display(fcns)

function alocalfunction

function anotherlocalfunction
```

From the command line, call your function.

```
fileWithLocalFunctions
```

```
fcns =

    @alocalfunction
    @anotherlocalfunction
```

More About

- "Local Functions"

See Also

`functiontests`

log

Natural logarithm

Syntax

```
Y = log(X)
```

Description

`Y = log(X)` returns the natural logarithm ln(x) of each element in array `X`.

The `log` function's domain includes negative and complex numbers, which can lead to unexpected results if used unintentionally. For negative and complex numbers `z = u + i*w`, the complex logarithm `log(z)` returns

```
log(abs(z)) + 1i*atan2(w,u)
```

If you want negative and complex numbers to return error messages rather than return complex results, use `reallog` instead.

Examples

Natural Logarithm of Negative Number

Show that the natural logarithm of -1 is $i\pi$.

```
log(-1)
```

```
ans =

   0.0000 + 3.1416i
```

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`Y` — Logarithm values

scalar | vector | matrix | multidimensional array

Logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

For positive real values of `X` in the interval (`0`, `Inf`), `Y` is in the interval (`-Inf`,`Inf`). For complex and negative real values of `X`, `Y` is complex. The data type of `Y` is the same as that of `X`.

See Also

`exp` | `log10` | `log1p` | `log2` | `loglog` | `logm` | `reallog` | `semilogx` | `semilogy`

Introduced before R2006a

log10

Common logarithm (base 10)

Syntax

```
Y = log10(X)
```

Description

`Y = log10(X)` returns the common logarithm of each element in array `X`. The function accepts both real and complex inputs. For real values of `X` in the interval (0, `Inf`), `log10` returns real values in the interval (`-Inf` ,`Inf`). For complex and negative real values of `X`, the `log10` function returns complex values.

Examples

Calculate Scalar Common Logarithm Values

Examine several values of the base 10 logarithm function.

Calculate the common logarithm of 1.

```
log10(1)
```

```
ans =

     0
```

The result is `0`, which is the x-intercept of the `log10` function.

Calculate the common logarithm of 10.

```
log10(10)
```

```
ans =

     1
```

The result is `1` since `10^1 = 10`.

Calculate the common logarithm of 100.

```
log10(100)
```

```
ans =

     2
```

The result is `2` since `10^2 = 100`.

Calculate the common logarithm of 0.

```
log10(0)
```

```
ans =

  -Inf
```

The result is `-Inf` since `10^(-Inf) = 0`.

Real-Valued Common Logarithm

Create a vector of real numbers in the interval `[0.5,10]`.

```
X = (0.5:0.5:10)';
```

Calculate the common logarithm of `X`.

```
Y = log10(X)
```

```
Y =

   -0.3010
         0
    0.1761
    0.3010
    0.3979
    0.4771
    0.5441
    0.6021
    0.6532
    0.6990
```

```
    0.7404
    0.7782
    0.8129
    0.8451
    0.8751
    0.9031
    0.9294
    0.9542
    0.9777
    1.0000
```

Complex-Valued Common Logarithm

Create two Cartesian grids for `X` and `Y`.

```
[X,Y] = meshgrid(0:0.5:1.5,-2:0.5:2);
```

Calculate the complex base 10 logarithm $log_{10}(X + iY)$ on the grid. Use `1i` for improved speed and robustness with complex arithmetic.

```
Z = log10(X + 1i*Y)
```

```
Z =

   0.3010 - 0.6822i   0.3142 - 0.5758i   0.3495 - 0.4808i   0.3979 - 0.4027i
   0.1761 - 0.6822i   0.1990 - 0.5425i   0.2559 - 0.4268i   0.3266 - 0.3411i
   0.0000 - 0.6822i   0.0485 - 0.4808i   0.1505 - 0.3411i   0.2559 - 0.2554i
  -0.3010 - 0.6822i  -0.1505 - 0.3411i   0.0485 - 0.2014i   0.1990 - 0.1397i
     -Inf + 0.0000i  -0.3010 + 0.0000i   0.0000 + 0.0000i   0.1761 + 0.0000i
  -0.3010 + 0.6822i  -0.1505 + 0.3411i   0.0485 + 0.2014i   0.1990 + 0.1397i
   0.0000 + 0.6822i   0.0485 + 0.4808i   0.1505 + 0.3411i   0.2559 + 0.2554i
   0.1761 + 0.6822i   0.1990 + 0.5425i   0.2559 + 0.4268i   0.3266 + 0.3411i
   0.3010 + 0.6822i   0.3142 + 0.5758i   0.3495 + 0.4808i   0.3979 + 0.4027i
```

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

See Also

`exp` | `log` | `log1p` | `log2` | `loglog` | `logm` | `reallog` | `semilogx` | `semilogy`

Introduced before R2006a

log1p

Compute log(1+x) accurately for small values of x

Syntax

```
y = log1p(x)
```

Description

`y = log1p(x)` computes `log(1+x)`, compensating for the roundoff in `1+x`. `log1p(x)` is more accurate than `log(1+x)` for small values of x. For small `x`, `log1p(x)` is approximately `x`, whereas `log(1+x)` can be zero.

See Also

`log` | `expm1`

Introduced before R2006a

log2

Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

Syntax

```
Y = log2(X)
[F,E] = log2(X)
```

Description

`Y = log2(X)` computes the base 2 logarithm of the elements of `X`.

`[F,E] = log2(X)` returns arrays `F` and `E`. Argument `F` is an array of real values, usually in the range `0.5 <= abs(F) < 1`. For real `X`, `F` satisfies the equation: `X = F.*2.^E`. Argument `E` is an array of integers that, for real `X`, satisfy the equation: `X = F.*2.^E`.

Examples

For IEEE arithmetic, the statement `[F,E] = log2(X)` yields the values:

X	F	E
1	1/2	1
pi	pi/4	2
-3	-3/4	2
eps	1/2	-51
realmax	1-eps/2	1024
realmin	1/2	-1021

More About

Tips

This function corresponds to the ANSI C function `frexp()` and the IEEE floating-point standard function `logb()`. Any zeros in `X` produce `F = 0` and `E = 0`.

See Also

`log` | `pow2`

Introduced before R2006a

logical

Convert numeric values to logicals

Syntax

```
L = logical(A)
```

Description

`L = logical(A)` converts numeric input `A` into an array of logical values. Any nonzero element of input `A` is converted to logical `1` (`true`) and zeros are converted to logical `0` (`false`). Complex values and NaNs cannot be converted to logical values and result in a conversion error.

Examples

Pick Odd Elements from Numeric Matrix

Pick out the odd-numbered elements of a numeric matrix.

Create a numeric matrix.

```
A = [1 -3 2;5 4 7;-8 1 3];
```

Find the modulus, `mod(A,2)`, and convert it to a logical array for indexing.

```
L = logical(mod(A,2))
```

```
L =

     1     1     0
     1     0     1
     0     1     1
```

The array has logical `1` (`true`) values where `A` is odd.

Use `L` as a logical index to pick out the odd elements of `A`.

```
A(L)

ans =

     1
     5
    -3
     1
     7
     3
```

The result is a vector containing all odd elements of `A`.

Use the logical NOT operator, `~`, on `L` to find the even elements of `A`.

```
A(~L)

ans =

    -8
     4
     2
```

Input Arguments

A — Numeric input

scalar | vector | matrix | multidimensional array

Numeric input, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

More About

Tips

- Most arithmetic operations involving logical arrays return double values. For example, adding zero to a logical array returns a double array.
- Logical arrays also are created by the relational operators (==,<,>,~, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

- " Using Logicals in Array Indexing"
- "Determine if Arrays Are Logical"

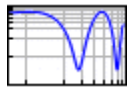
See Also

`false` | `islogical` | `true`

Introduced before R2006a

loglog

Log-log scale plot



Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
```

Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all `Yn` versus `Xn` pairs. If only one of `Xn` or `Yn` is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

`loglog(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

`loglog(...,'PropertyName',PropertyValue,...)` sets line property values for all the charting lines created. For a list of properties, see Chart Line Properties.

`h = loglog(...)` returns a column vector of chart line handles, one handle per line.

If you do not specify a color when plotting more than one line, `loglog` automatically cycles through the colors and line styles in the order specified by the current axes.

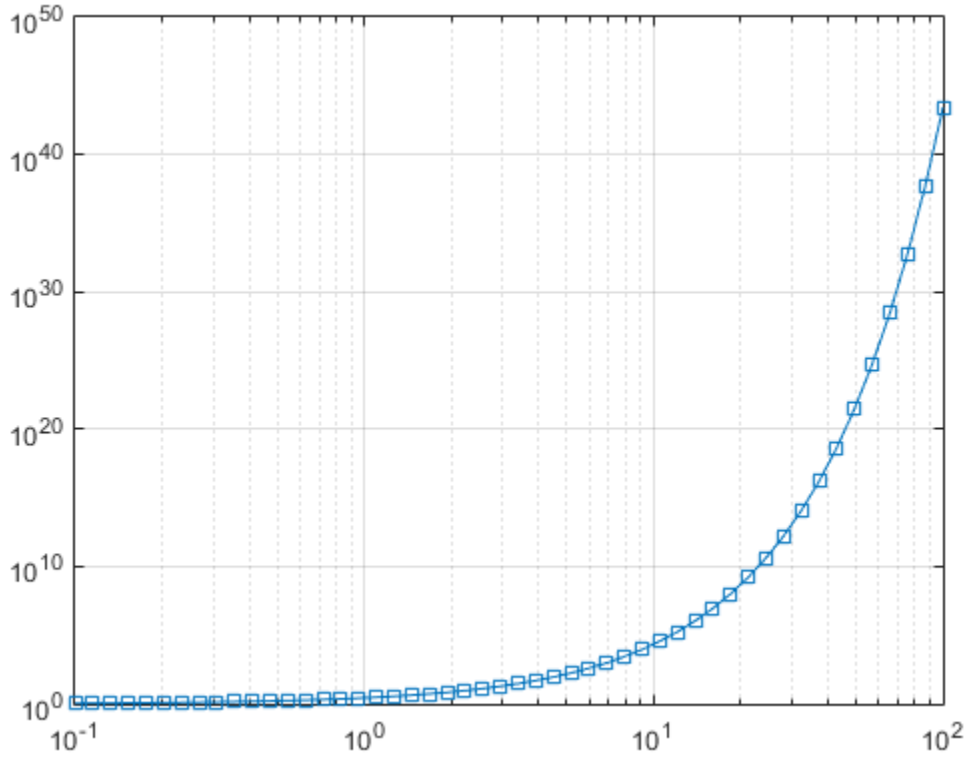
If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

Examples

Logarithmic Scale for Both Axes

Create a plot using a logarithmic scale for both the x-axis and the y-axis. Set the `LineSpec` string so that `loglog` plots using a line with square markers. Display the grid.

```
x = logspace(-1,2);
y = exp(x);

figure
loglog(x,y,'-s')
grid on
```



See Also

Functions
`LineSpec` | `plot` | `semilogx` | `semilogy`

Properties
Chart Line Properties

Introduced before R2006a

logm

Matrix logarithm

Syntax

```
L = logm(A)
[L, exitflag] = logm(A)
```

Description

`L = logm(A)` is the principal matrix logarithm of A, the inverse of `expm(A)`. `L` is the unique logarithm for which every eigenvalue has imaginary part lying strictly between $-\pi$ and π. If `A` is singular or has any eigenvalues on the negative real axis, the principal logarithm is undefined. In this case, `logm` computes a nonprincipal logarithm and returns a warning message.

`[L, exitflag] = logm(A)` returns a scalar `exitflag` that describes the exit condition of `logm`:

- If `exitflag = 0`, the algorithm was successfully completed.
- If `exitflag = 1`, too many matrix square roots had to be computed. However, the computed value of `L` might still be accurate.

The input `A` can have class `double` or `single`.

Limitations

For most matrices:

```
logm(expm(A)) = A = expm(logm(A))
```

These identities may fail for some `A`. For example, if the computed eigenvalues of `A` include an exact zero, then `logm(A)` generates infinity. Or, if the elements of `A` are too large, `expm(A)` may overflow.

Examples

Suppose `A` is the 3-by-3 matrix

```
     1     1     0
     0     0     2
     0     0    -1
```

and `Y = expm(A)` is

```
Y =
    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679
```

Then `A = logm(Y)` produces the original matrix `A`.

```
Y =
    1.0000    1.0000    0.0000
         0         0    2.0000
         0         0   -1.0000
```

But `log(A)` involves taking the logarithm of zero, and so produces

```
ans =
    0.0000                  0          -35.5119
     -Inf               -Inf            0.6931
     -Inf               -Inf            0.0000 + 3.1416i
```

More About

Tips

If `A` is real symmetric or complex Hermitian, then so is `logm(A)`.

Some matrices, like `A = [0 1; 0 0]`, do not have any logarithms, real or complex, so `logm` cannot be expected to produce one.

Algorithms

The algorithm `logm` uses is described in [1].

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Cheng, S. H., N. J. Higham, C. S. Kenney, and A. J. Laub, "Approximating the logarithm of a matrix to specified accuracy," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1112-1125, 2001.

[3] Higham, N. J., "Evaluating Pade approximants of the matrix logarithm," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1126-1135, 2001.

[4] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[5] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review* 20, 1978, pp. 801-836.

See Also

`expm` | `funm` | `sqrtm`

Introduced before R2006a

logspace

Generate logarithmically spaced vectors

Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
y = logspace(a,pi)
```

Description

The `logspace` function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of `linspace` and the ":" or colon operator.

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades `10^a` and `10^b`.

`y = logspace(a,b,n)` generates `n` points between decades `10^a` and `10^b`.

`y = logspace(a,pi)` generates the points between `10^a` and `pi`, which is useful for digital signal processing where frequencies over this interval go around the unit circle.

More About

Tips

All the arguments to `logspace` must be scalars.

See Also

`linspace` | `colon operator`

Introduced before R2006a

lookfor

Search for keyword in all help entries

Alternatives

As an alternative to the `lookfor` function, use the Function Browser.

Syntax

```
lookfor topic
lookfor topic -all
```

Description

`lookfor topic` searches for the string `topic` in the first comment line (the H1 line) of the help text in all MATLAB program files found on the search path. For all files in which a match occurs, `lookfor` displays the H1 line.

`lookfor topic -all` searches the entire first comment block of a MATLAB program file looking for `topic`.

Examples

For example:

```
lookfor inverse
```

finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with

```
which inverse
```

or

```
what inverse
```

These functions run more quickly, but probably fail to find anything because MATLAB does not have a function `inverse`.

In summary, `what` lists the functions in a given folder, `which` finds the folder containing a given function or file, and `lookfor` finds all functions in all folders that might have something to do with a given keyword.

Even more extensive than the `lookfor` function are the find features in the Current Folder browser. For example, you can look for all occurrences of a specified word in all the MATLAB program files in the current folder and its subfolders. For more information, see "Find Files and Folders".

More About

- "Find Functions to Use"
- "Search Syntax and Tips"

See Also

`dir` | `doc` | `filebrowser` | `strfind` | `help` | `regexp` | `what` | `which` | `who`

Introduced before R2006a

lower

Convert string to lowercase

Syntax

```
t = lower('str')
B = lower(A)
```

Description

`t = lower('str')` returns the string formed by converting any uppercase characters in `str` to the corresponding lowercase characters and leaving all other characters unchanged.

`B = lower(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `lower` to each string within `A`.

Examples

`lower('MathWorks')` is `mathworks`.

More About

Tips

Character sets supported:

- PC: Latin-1 for the Microsoft Windows operating system
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`upper`

Introduced before R2006a

ls

List folder contents

Syntax

```
ls
ls name
list = ls(name)
```

Description

`ls` lists the contents of the current folder.

`ls name` lists the files and folders in the current folder that match the specified name. You can use wildcards.

`list = ls(name)` returns the files and folders in the current folder that match the specified name to `list`.

Input Arguments

`name`

A string value specifying a file or folder name.

Output Arguments

`list`

- On UNIX platforms, list is a character row vector of names separated by tab and space characters.
- On Microsoft Windows platforms, list is an `m`-by-`n` character array of names—`m` is the number of names and `n` is the number of characters in the longest name. MATLAB pads names shorter than `n` characters with space characters.

Examples

List all the files and folders in the current folder:

```
ls
```

List all the files and folders in the current folder that begin with the letter `h`:

```
ls h*
```

Return the list of all the files and folders in the current folder to `mylist`:

```
mylist = ls;
```

Alternatives

- View files and folders in the Current Folder browser.

  Open the Current Folder browser by issuing the `filebrowser` command.

More About

Tips

- On UNIX platforms, you can add any flags to `ls` that the operating system supports.

See Also

`dir` | `pwd`

Introduced before R2006a

lscov

Least-squares solution in presence of known covariance

Syntax

```
x = lscov(A,B)
x = lscov(A,B,w)
x = lscov(A,B,V)
x = lscov(A,B,V,alg)
[x,stdx] = lscov(...)
[x,stdx,mse] = lscov(...)
[x,stdx,mse,S] = lscov(...)
```

Description

`x = lscov(A,B)` returns the ordinary least squares solution to the linear system of equations `A*x = B`, i.e., `x` is the n-by-1 vector that minimizes the sum of squared errors `(B - A*x)'*(B - A*x)`, where `A` is m-by-n, and `B` is m-by-1. `B` can also be an m-by-k matrix, and `lscov` returns one solution for each column of `B`. When `rank(A) < n`, `lscov` sets the maximum possible number of elements of `x` to zero to obtain a "basic solution".

`x = lscov(A,B,w)`, where `w` is a vector length m of real positive weights, returns the weighted least squares solution to the linear system `A*x = B`, that is, `x` minimizes `(B - A*x)'*diag(w)*(B - A*x)`. `w` typically contains either counts or inverse variances.

`x = lscov(A,B,V)`, where `V` is an m-by-m real symmetric positive definite matrix, returns the generalized least squares solution to the linear system `A*x = B` with covariance matrix proportional to `V`, that is, `x` minimizes `(B - A*x)'*inv(V)*(B - A*x)`.

More generally, `V` can be positive semidefinite, and `lscov` returns `x` that minimizes `e'*e`, subject to `A*x + T*e = B`, where the minimization is over `x` and `e`, and `T*T' = V`. When `V` is semidefinite, this problem has a solution only if `B` is consistent with `A` and `V` (that is, `B` is in the column space of `[A T]`), otherwise `lscov` returns an error.

By default, `lscov` computes the Cholesky decomposition of `V` and, in effect, inverts that factor to transform the problem into ordinary least squares. However, if `lscov` determines that `V` is semidefinite, it uses an orthogonal decomposition algorithm that avoids inverting `V`.

`x = lscov(A,B,V,alg)` specifies the algorithm used to compute `x` when `V` is a matrix. `alg` can have the following values:

- `'chol'` uses the Cholesky decomposition of `V`.
- `'orth'` uses orthogonal decompositions, and is more appropriate when `V` is ill-conditioned or singular, but is computationally more expensive.

`[x,stdx] = lscov(...)` returns the estimated standard errors of `x`. When `A` is rank deficient, `stdx` contains zeros in the elements corresponding to the necessarily zero elements of `x`.

`[x,stdx,mse] = lscov(...)` returns the mean squared error. If `B` is assumed to have covariance matrix σ^2`V` (or (σ^2)×`diag`(1./`W`)), then `mse` is an estimate of σ^2.

`[x,stdx,mse,S] = lscov(...)` returns the estimated covariance matrix of `x`. When `A` is rank deficient, `S` contains zeros in the rows and columns corresponding to the necessarily zero elements of `x`. `lscov` cannot return `S` if it is called with multiple right-hand sides, that is, if `size(B,2) > 1`.

The standard formulas for these quantities, when `A` and `V` are full rank, are

- `x = inv(A'*inv(V)*A)*A'*inv(V)*B`
- `mse = B'*(inv(V) - inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)`
- `S = inv(A'*inv(V)*A)*mse`
- `stdx = sqrt(diag(S))`

However, `lscov` uses methods that are faster and more stable, and are applicable to rank deficient cases.

`lscov` assumes that the covariance matrix of `B` is known only up to a scale factor. `mse` is an estimate of that unknown scale factor, and `lscov` scales the outputs `S` and `stdx` appropriately. However, if `V` is known to be exactly the covariance matrix of `B`, then that scaling is unnecessary. To get the appropriate estimates in this case, you should rescale `S` and `stdx` by `1/mse` and `sqrt(1/mse)`, respectively.

Examples

Example 1 — Computing Ordinary Least Squares

The MATLAB backslash operator (\) enables you to perform linear regression by computing ordinary least-squares (OLS) estimates of the regression coefficients. You can also use `lscov` to compute the same OLS estimates. By using `lscov`, you can also compute estimates of the standard errors for those coefficients, and an estimate of the standard deviation of the regression error term:

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
X = [ones(size(x1)) x1 x2];
y = [.17 .26 .28 .23 .27 .34]';

a = X\y
a =
    0.1203
    0.3284
   -0.1312

[b,se_b,mse] = lscov(X,y)
b =
    0.1203
    0.3284
   -0.1312
se_b =
    0.0643
    0.2267
    0.1488
mse =
    0.0015
```

Example 2 — Computing Weighted Least Squares

Use `lscov` to compute a weighted least-squares (WLS) fit by providing a vector of relative observation weights. For example, you might want to downweight the influence of an unreliable observation on the fit:

```
w = [1 1 1 1 1 .1]';

[bw,sew_b,msew] = lscov(X,y,w)
```

```
bw =
    0.1046
    0.4614
   -0.2621
sew_b =
    0.0309
    0.1152
    0.0814
msew =
  3.4741e-004
```

Example 3 — Computing General Least Squares

Use `lscov` to compute a general least-squares (GLS) fit by providing an observation covariance matrix. For example, your data may not be independent:

```
V = .2*ones(length(x1)) + .8*diag(ones(size(x1)));

[bg,sew_b,mseg] = lscov(X,y,V)
bg =
    0.1203
    0.3284
   -0.1312
sew_b =
    0.0672
    0.2267
    0.1488
mseg =
    0.0019
```

Example 4 — Estimating the Coefficient Covariance Matrix

Compute an estimate of the coefficient covariance matrix for either OLS, WLS, or GLS fits. The coefficient standard errors are equal to the square roots of the values on the diagonal of this covariance matrix:

```
[b,se_b,mse,S] = lscov(X,y);

S
S =
    0.0041   -0.0130    0.0075
   -0.0130    0.0514   -0.0328
    0.0075   -0.0328    0.0221
```

```
[se_b sqrt(diag(S))]
ans =
    0.0643    0.0643
    0.2267    0.2267
    0.1488    0.1488
```

More About

Algorithms

The vector `x` minimizes the quantity `(A*x-B)'*inv(V)*(A*x-B)`. The classical linear algebra solution to this problem is

```
x = inv(A'*inv(V)*A)*A'*inv(V)*B
```

but the `lscov` function instead computes the QR decomposition of `A` and then modifies `Q` by `V`.

References

[1] Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

See Also

`lsqnonneg` | `mldivide` | `mrdivide` | `qr`

Introduced before R2006a

lsqnonneg

Solve nonnegative least-squares constraints problem

Equation

Solves nonnegative least-squares curve fitting problems of the form

$$\min_x \|C \cdot x - d\|_2^2, \text{ where } x \geq 0.$$

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,options)
x = lsqnonneg(problem)
[x,resnorm] = lsqnonneg(...)
[x,resnorm,residual] = lsqnonneg(...)
[x,resnorm,residual,exitflag] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)
```

Description

`x = lsqnonneg(C,d)` returns the vector `x` that minimizes `norm(C*x-d)` subject to `x >= 0`. `C` and `d` must be real.

`x = lsqnonneg(C,d,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `lsqnonneg` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`TolX`	Termination tolerance on `x`.

`x = lsqnonneg(problem)` finds the minimum for `problem`, where `problem` is a structure with the following fields:

`C`	Matrix
`d`	Vector
`solver`	`'lsqnonneg'`
`options`	Options structure created using `optimset`

`[x,resnorm] = lsqnonneg(...)` returns the value of the squared 2-norm of the residual: `norm(C*x-d)^2`.

`[x,resnorm,residual] = lsqnonneg(...)` returns the residual, `d-C*x`.

`[x,resnorm,residual,exitflag] = lsqnonneg(...)` returns a value `exitflag` that describes the exit condition of `lsqnonneg`:

>0	Indicates that the function converged to a solution `x`.
0	Indicates that the iteration count was exceeded. Increasing the tolerance (`TolX` parameter in `options`) may lead to a solution.

`[x,resnorm,residual,exitflag,output] = lsqnonneg(...)` returns a structure `output` that contains information about the operation in the following fields:

`algorithm`	`'active-set'`
`iterations`	The number of iterations taken
`message`	Exit message

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)` returns the dual vector (Lagrange multipliers) `lambda`, where `lambda(i)<=0` when `x(i)` is (approximately) `0`, and `lambda(i)` is (approximately) `0` when `x(i)>0`.

Examples

Compare the unconstrained least squares solution to the `lsqnonneg` solution for a 4-by-2 problem:

```
C = [
```

```
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170];
d = [
    0.8587
    0.1781
    0.0747
    0.8405];
[C\d lsqnonneg(C,d)] =
    -2.5627         0
     3.1108    0.6929
[norm(C*(C\d)-d) norm(C*lsqnonneg(C,d)-d)] =
     0.6674 0.9118
```

The solution from `lsqnonneg` does not fit as well (has a larger residual), as the least squares solution. However, the nonnegative least squares solution has no negative components.

More About

Algorithms

`lsqnonneg` uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector `lambda`. It then selects the basis vector corresponding to the maximum value in `lambda` in order to swap out of the basis in exchange for another possible candidate. This continues until `lambda <= 0`.

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974, Chapter 23, p. 161.

See Also

`optimset`

Introduced before R2006a

lsqr

LSQR method

Syntax

```
x = lsqr(A,b)
lsqr(A,b,tol)
lsqr(A,b,tol,maxit)
lsqr(A,b,tol,maxit,M)
lsqr(A,b,tol,maxit,M1,M2)
lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)
```

Description

`x = lsqr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x` if `A` is consistent, otherwise it attempts to solve the least squares solution `x` that minimizes `norm(b-A*x)`. The `m`-by-`n` coefficient matrix `A` need not be square but it should be large and sparse. The column vector `b` must have length `m`. You can specify `A` as a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `lsqr` converges, a message to that effect is displayed. If `lsqr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`lsqr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `lsqr` uses the default, `1e-6`.

`lsqr(A,b,tol,maxit)` specifies the maximum number of iterations.

`lsqr(A,b,tol,maxit,M)` and `lsqr(A,b,tol,maxit,M1,M2)` use `n`-by-`n` preconditioner `M` or `M = M1*M2` and effectively solve the system `A*inv(M)*y = b` for `y`, where `y = M*x`. If `M` is `[]` then `lsqr` applies no preconditioner. `M` can be a function `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`lsqr(A,b,tol,maxit,M1,M2,x0)` specifies the `n`-by-`1` initial guess. If `x0` is `[]`, then `lsqr` uses the default, an all zero vector.

`[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a convergence flag.

Flag	Convergence
0	`lsqr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`lsqr` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`lsqr` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `lsqr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if you specify the `flag` output.

`[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns an estimate of the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norm estimates at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of estimates of the scaled normal equations residual at each iteration: `norm((A*inv(M))'*(B-A*X))/norm(A*inv(M),'fro')`. Note that the estimate of `norm(A*inv(M),'fro')` changes, and hopefully improves, at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = lsqr(A,b,tol,maxit,M1,M2);
```

displays the following message:

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in a function `run_lsqr` that

- Calls `lsqr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_lsqr` are available to `afun`.

The following shows the code for `run_lsqr`:

```
function x1 = run_lsqr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = lsqr(@afun,b,tol,maxit,M1,M2);
```

```
    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
       elseif strcmp(transp_flag,'notransp') % y = A*x
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - x(2:n);
       end
    end
end
```

When you enter

```
x1=run_lsqr;
```

MATLAB software displays the message

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations And Sparse Least Squares," *ACM Trans. Math. Soft.*, Vol.8, 1982, pp.43-71.

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `minres` | `norm` | `pcg` | `qmr` | `symmlq` | `function_handle`

Introduced before R2006a

lt, <

Determine less than

Syntax

```
A < B
lt(A,B)
```

Description

`A < B` returns an array with elements set to logical 1 (`true`) where `A` is less than `B`; otherwise, it returns logical `0` (`false`).

The test compares only the real part of numeric arrays. `lt` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`lt(A,B)` is an alternate way to execute `A < B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Determine if vector elements are less than a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are less than `12`.

```
A < 12
```

```
ans =

     1     0     0     1     1     1     1     0
```

The result is a vector with values of logical 1 (`true`) where the elements of A satisfy the expression.

Use the vector of logical values as an index to view the values in A that are less than 12.

```
A(A < 12)
```

```
ans =

     1     7     9    11     2
```

The result is a subset of the elements in A.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values less than 9 with the value 10.

```
A(A < 9) = 10
```

```
A =

    16    10    10    13
    10    11    10    10
     9    10    10    12
    10    14    15    10
```

The result is a new matrix whose smallest element is 9.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)
```

```
A =

     large       medium      small
     medium      small       large
```

The array has three categories: `'small'`, `'medium'`, and `'large'`.

Find all values less than the category `'medium'`.

```
A < 'medium'
```

```
ans =

     0     0     1
     0     1     0
```

A value of logical `1` (`true`) indicates a value less than the category `'medium'`.

Compare the rows of `A`.

```
A(1,:) < A(2,:)
```

```
ans =

     0     0     1
```

The function returns logical `1` (`true`) where the first row has a category value less than the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are less than `3`.

```
A(A < 3)
```

```
ans =

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i
```

`lt` compares only the real part of the elements in `A`.

Use `abs` to find which elements are within a radius of `3` from the origin.

```
A(abs(A) < 3)

ans =

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 - 2.0000i
```

The result has one less element. The element `1.0000 + 3.0000i` is not within a radius of `3` from the origin.

Compare Dates

Create a vector of dates.

```
A = datetime([2014,05,01;2014,05,31])

A =

   01-May-2014
   31-May-2014
```

Find the dates that occur before May 10, 2014.

```
A(A < '2014-05-10')

ans =

   01-May-2014
```

Input Arguments

A — Left array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Left array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See “Compare Categorical Array Elements” for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

B — Right array

numeric array | logical array | character array | ordinal categorical array | datetime array | duration array

Right array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

More About

- "Ordinal Categorical Arrays"

See Also

`eq` | `ge` | `gt` | `le` | `ne`

Introduced before R2006a

lu

LU matrix factorization

Syntax

```
Y = lu(A)
[L,U] = lu(A)
[L,U,P] = lu(A)
[L,U,P,Q] = lu(A)
[L,U,P,Q,R] = lu(A)
[...] = lu(A,'vector')
[...] = lu(A,thresh)
[...] = lu(A,thresh,'vector')
```

Description

The `lu` function expresses a matrix `A` as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the *LU*, or sometimes the *LR*, factorization. `A` can be rectangular.

`Y = lu(A)` returns matrix `Y` that, for sparse `A`, contains the strictly lower triangular `L`, i.e., without its unit diagonal, and the upper triangular `U` as submatrices. That is, if `[L,U,P] = lu(A)`, then `Y = U+L-eye(size(A))`. For nonsparse `A`, `Y` is the output from the LAPACK `dgetrf` or `zgetrf` routine. The permutation matrix `P` is not returned.

`[L,U] = lu(A)` returns an upper triangular matrix in `U` and a permuted lower triangular matrix in `L` such that `A = L*U`. Return value `L` is a product of lower triangular and permutation matrices.

`[L,U,P] = lu(A)` returns an upper triangular matrix in `U`, a lower triangular matrix `L` with a unit diagonal, and a permutation matrix `P`, such that `L*U = P*A`. The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q] = lu(A)` for sparse nonempty `A`, returns a unit lower triangular matrix `L`, an upper triangular matrix `U`, a row permutation matrix `P`, and a column reordering

matrix Q, so that P*A*Q = L*U. If A is empty or not sparse, lu displays an error message. The statement lu(A,'matrix') returns identical output values.

[L,U,P,Q,R] = lu(A) returns unit lower triangular matrix L, upper triangular matrix U, permutation matrices P and Q, and a diagonal scaling matrix R so that P*(R\A)*Q = L*U for sparse non-empty A. Typically, but not always, the row-scaling leads to a sparser and more stable factorization. The statement lu(A,'matrix') returns identical output values.

[...] = lu(A,'vector') returns the permutation information in two row vectors p and q. You can specify from 1 to 5 outputs. Output p is defined as A(p,:)=L*U, output q is defined as A(p,q)=L*U, and output R is defined as R(:,p)\A(:,q)=L*U.

[...] = lu(A,thresh) controls pivoting. This syntax applies to sparse matrices only. The thresh input is a one- or two-element vector of type single or double that defaults to [0.1, 0.001]. If A is a square matrix with a mostly symmetric structure and mostly nonzero diagonal, MATLAB uses a symmetric pivoting strategy. For this strategy, the diagonal where

```
A(i,j) >= thresh(2) * max(abs(A(j:m,j)))
```

is selected. If the diagonal entry fails this test, a pivot entry below the diagonal is selected, using thresh(1). In this case, L has entries with absolute value 1/min(thresh) or less.

If A is not as described above, MATLAB uses an asymmetric strategy. In this case, the sparsest row i where

```
A(i,j) >= thresh(1) * max(abs(A(j:m,j)))
```

is selected. A value of 1.0 results in conventional partial pivoting. Entries in L have an absolute value of 1/thresh(1) or less. The second element of the thresh input vector is not used when MATLAB uses an asymmetric strategy.

Smaller values of thresh(1) and thresh(2) tend to lead to sparser LU factors, but the solution can become inaccurate. Larger values can lead to a more accurate solution (but not always), and usually an increase in the total work and memory usage. The statement lu(A,thresh,'matrix') returns identical output values.

[...] = lu(A,thresh,'vector') controls the pivoting strategy and also returns the permutation information in row vectors, as described above. The thresh input must precede 'vector' in the input argument list.

Note In rare instances, incorrect factorization results in `P*A*Q` ≠ `L*U`. Increase `thresh`, to a maximum of `1.0` (regular partial pivoting), and try again.

Arguments

`A`	Rectangular matrix to be factored.
`thresh`	Pivot threshold for sparse matrices. Valid values are in the interval `[0,1]`. If you specify the fourth output `Q`, the default is `0.1`. Otherwise, the default is `1.0`.
`L`	Factor of `A`. Depending on the form of the function, `L` is either a unit lower triangular matrix, or else the product of a unit lower triangular matrix with `P'`.
`U`	Upper triangular matrix that is a factor of `A`.
`P`	Row permutation matrix satisfying the equation `L*U = P*A`, or `L*U = P*A*Q`. Used for numerical stability.
`Q`	Column permutation matrix satisfying the equation `P*A*Q = L*U`. Used to reduce fill-in in the sparse case.
`R`	Row-scaling matrix

Examples

Example 1

Start with

```
A = [ 1    2    3
      4    5    6
      7    8    0 ];
```

To see the LU factorization, call `lu` with two output arguments.

```
[L1,U] = lu(A)

L1 =
    0.1429    1.0000         0
```

```
    0.5714    0.5000    1.0000
    1.0000         0         0

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000
```

Notice that `L1` is a permutation of a lower triangular matrix: if you switch rows 2 and 3, and then switch rows 1 and 2, the resulting matrix is lower triangular and has `1`s on the diagonal. Notice also that `U` is upper triangular. To check that the factorization does its job, compute the product

```
L1*U
```

which returns the original `A`. The inverse of the example matrix, `X = inv(A)`, is actually computed from the inverses of the triangular factors

```
X = inv(U)*inv(L1)
```

Using three arguments on the left side to get the permutation matrix as well,

```
[L2,U,P] = lu(A)
```

returns a truly lower triangular `L2`, the same value of `U`, and the permutation matrix `P`.

```
L2 =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000

P =
    0    0    1
    1    0    0
    0    1    0
```

Note that `L2 = P*L1`.

```
P*L1
```

```
ans =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
```

To verify that `L2*U` is a permuted version of `A`, compute `L2*U` and subtract it from `P*A`:

```
P*A - L2*U

ans =
     0     0     0
     0     0     0
     0     0     0
```

In this case, `inv(U)*inv(L)` results in the permutation of `inv(A)` given by `inv(P)*inv(A)`.

The determinant of the example matrix is

```
d = det(A)

d =    27
```

It is computed from the determinants of the triangular factors

```
d = det(L)*det(U)
```

The solution to $Ax = b$ is obtained with matrix division

```
x = A\b
```

The solution is actually computed by solving two triangular systems

```
y = L\b
x = U\y
```

Example 2

The 1-norm of their difference is within roundoff error, indicating that `L*U = P*B*Q`.

Generate a 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster-Fuller geodesic dome.

```
B = bucky;
```

Use the sparse matrix syntax with four outputs to get the row and column permutation matrices.

```
[L,U,P,Q] = lu(B);
```

Apply the permutation matrices to `B`, and subtract the product of the lower and upper triangular matrices.

```
Z = P*B*Q - L*U;
norm(Z,1)

ans =
  7.9936e-015
```

Example 3

This example illustrates the benefits of using the `'vector'` option. Note how much memory is saved by using the `lu(F,'vector')` syntax.

```
F = gallery('uniformdata',[1000 1000],0);
g = sum(F,2);
[L,U,P] = lu(F);
[L,U,p] = lu(F,'vector');
whos P p

Name          Size            Bytes  Class      Attributes
P          1000x1000       8000000  double
p             1x1000          8000  double
```

The following two statements are equivalent. The first typically requires less time:

```
x = U\(L\(g(p,:)));
y = U\(L\(P*g));
```

More About

Tips

Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with `inv` and the

determinant with `det`. It is also the basis for the linear equation solution or matrix division obtained with `\` and `/`.

See Also

`cond` | `det` | `inv` | `ilu` | `qr` | `rref`

Introduced before R2006a

Primitive Line Properties

Control primitive line appearance and behavior

Primitive line properties control the appearance and behavior of a primitive line object. By changing property values, you can modify certain aspects of the line.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = line;
s = h.LineStyle;
h.LineStyle = ':';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Line

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the color as `'none'`, then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.

- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the primitive line object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

`6` (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Data

`XData` — x values

`[0 1]` (default) | vector

x values, specified as a vector. `XData` and `YData` must have equal lengths.

Example: `1:10`

`YData` — y values

`[0 1]` (default) | vector

y values, specified as a vector. `XData` and `YData` must have equal lengths.

Example: `1:10`

`ZData` — z values

empty matrix (default) | vector

z values, specified as a vector. `ZData` must have the same length as `XData` and `YData`.

Example: `1:10`

Visibility

`Visible` — Visibility of primitive line

`'on'` (default) | `'off'`

Visibility of primitive line, specified as one of these values:

- `'on'` — Display the primitive line.
- `'off'` — Hide the primitive line without deleting it. You still can access the properties of an invisible primitive line object.

`Clipping` — Clipping of primitive line to axes limits

`'on'` (default) | `'off'`

Clipping of primitive line to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the primitive line that are outside the axes limits.
- `'off'` — Display the entire primitive line, even if parts of it appear outside the axes limits. Parts of the primitive line might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the primitive line that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'line'`

Type of graphics object, returned as `'line'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Tag to associate with primitive line

`''` (default) | string

Tag to associate with the primitive line, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with primitive line

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the primitive line object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the primitive line.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the primitive line object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the primitive line from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the primitive line object in the legend as one entry (default).
   - `'off'` — Do not include the primitive line object in the legend.
   - `'children'` — Include only children of the primitive line object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of primitive line

axes object | group object | transform object

Parent of primitive line, specified as an axes, group, or transform object.

`Children` — Children of primitive line

empty `GraphicsPlaceholder` array

The primitive line has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of primitive line object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The primitive line object handle is always visible.
- `'off'` — The primitive line object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

- `'callback'` — The primitive line object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the primitive line at the command-line, but allows callback functions to access it.

If the primitive line object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the primitive line. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The primitive line object — You can access properties of the primitive line object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the primitive line. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the primitive line when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the primitive line.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the primitive line that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the primitive line responds to the click or if an ancestor does.
- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'` and you can click a part of the primitive line that has no color. The HitTest property determines if the primitive line responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the primitive line passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the primitive line. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the primitive line that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the primitive line object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the primitive line is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the

`BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the primitive line tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the primitive line. Setting the `CreateFcn` property on an existing primitive line has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during primitive line creation. MATLAB executes the callback after creating the primitive line and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The primitive line object — You can access properties of the primitive line object from within the callback function. You also can access the primitive line object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the primitive line. MATLAB executes the callback before destroying the primitive line so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The primitive line object — You can access properties of the primitive line object from within the callback function. You also can access the primitive line object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of primitive line

`'off'` (default) | `'on'`

Deletion status of primitive line, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the primitive line begins

execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the primitive line no longer exists.

Check the value of the BeingDeleted property to verify that the primitive line is not about to be deleted before querying or modifying it.

See Also

`line`

More About

- "Access Property Values"
- "Graphics Object Properties"

magic

Magic square

Syntax

```
M = magic(n)
```

Description

`M = magic(n)` returns an `n`-by-`n` matrix constructed from the integers `1` through `n^2` with equal row and column sums. The order n must be a scalar greater than or equal to `3`.

Examples

The magic square of order 3 is

```
M = magic(3)
```

```
M =

    8    1    6
    3    5    7
    4    9    2
```

This is called a magic square because the sum of the elements in each column is the same.

```
 sum(M) =

     15    15    15
```

And the sum of the elements in each row, obtained by transposing twice, is the same.

```
 sum(M')' =

     15
```

```
     15
     15
```

This is also a special magic square because the diagonal elements have the same sum.

```
sum(diag(M)) =

     15
```

The value of the characteristic sum for a magic square of order `n` is

```
sum(1:n^2)/n
```

which, when `n = 3`, is `15`.

Limitations

If you supply `n` less than `3`, `magic` returns either a nonmagic square, or else the degenerate magic squares `1` and `[]`.

More About

Tips

A magic square, scaled by its magic sum, is doubly stochastic.

Algorithms

There are three different algorithms:

- `n` odd
- `n` even but not divisible by four
- `n` divisible by four

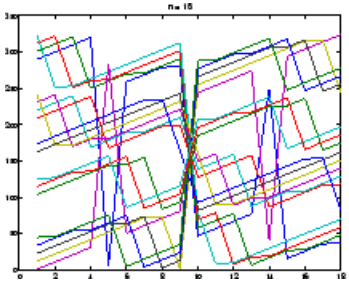
To make this apparent, type

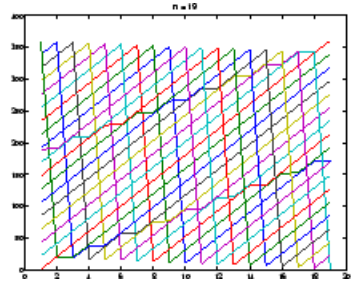
```
for n = 3:20
    A = magic(n);
    r(n) = rank(A);
end
```

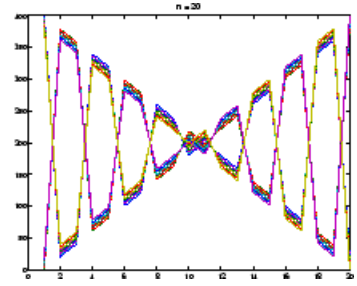
For n odd, the rank of the magic square is n. For n divisible by 4, the rank is 3. For n even but not divisible by 4, the rank is n/2 + 2.

```
[(3:20)',r(3:20)']
ans =
      3      3
      4      3
      5      5
      6      5
      7      7
      8      3
      9      9
     10      7
     11     11
     12      3
     13     13
     14      9
     15     15
     16      3
     17     17
     18     11
     19     19
     20      3
```

Plotting A for n = 18, 19, 20 shows the characteristic plot for each category.







See Also

ones | rand

Introduced before R2006a

makehgtform

Create 4-by-4 transform matrix

Syntax

```
M = makehgtform
M = makehgtform('translate',[tx ty tz])
M = makehgtform('scale',s)
M = makehgtform('scale',[sx,sy,sz])
M = makehgtform('xrotate',t)
M = makehgtform('yrotate',t)
M = makehgtform('zrotate',t)
M = makehgtform('axisrotate',[ax,ay,az],t)
```

Description

Use `makehgtform` to create transform matrices for translation, scaling, and rotation of graphics objects. Apply the transform to graphics objects by assigning the transform to the `Matrix` property of a parent transform object.

`M = makehgtform` returns an identity transform.

`M = makehgtform('translate',[tx ty tz])` or M = makehgtform('translate',tx,ty,tz) returns a transform that translates along the *x*-axis by `tx`, along the *y*-axis by `ty`, and along the *z*-axis by `tz`.

`M = makehgtform('scale',s)` returns a transform that scales uniformly along the *x*-, *y*-, and *z*-axes.

`M = makehgtform('scale',[sx,sy,sz])` returns a transform that scales along the *x*-axis by `sx`, along the *y*-axis by `sy`, and along the *z*-axis by `sz`.

`M = makehgtform('xrotate',t)` returns a transform that rotates around the *x*-axis by `t` radians.

`M = makehgtform('yrotate',t)` returns a transform that rotates around the *y*-axis by `t` radians.

`M = makehgtform('zrotate',t)` returns a transform that rotates around the *z*-axis by `t` radians.

`M = makehgtform('axisrotate',[ax,ay,az],t)` Rotate around axis `[ax ay az]` by `t` radians.

Note that you can specify multiple operations in one call to `makehgtform` and the MATLAB software returns a transform matrix that is the result of concatenating all specified operations. For example,

```
m = makehgtform('xrotate',pi/2,'yrotate',pi/2);
```

is the same as

```
mx = makehgtform('xrotate',pi/2);
my = makehgtform('yrotate',pi/2);
m = mx*my;
```

More About

- "Create Object Groups"

See Also

`hggroup` | `hgtransform`

Introduced before R2006a

mapreduce

Programming technique for analyzing data sets that do not fit in memory

Syntax

```
outds = mapreduce(ds,mapfun,reducefun)
outds = mapreduce(ds,mapfun,reducefun,mr)
outds = mapreduce( ___ ,Name,Value)
```

Description

`outds = mapreduce(ds,mapfun,reducefun)` applies map function `mapfun` to input datastore `ds`, and then passes the values associated with each unique key to reduce function `reducefun`. The output datastore is a `KeyValueDatastore` object that points to `.mat` files in the current folder.

`outds = mapreduce(ds,mapfun,reducefun,mr)` optionally specifies the run-time configuration settings for `mapreduce`. The `mr` input is the result of a call to the `mapreducer` function. Typically, this argument is used with Parallel Computing Toolbox, MATLAB Distributed Computing Server, or MATLAB Compiler. For more information, see “Speed Up and Deploy MapReduce Using Other Products”.

`outds = mapreduce( ___ ,Name,Value)` specifies additional options with one or more Name,Value pair arguments using any of the previous syntaxes. For example, you can specify `'OutputFolder'` followed by a string specifying a path to the output folder.

Examples

- “Find Maximum Value with MapReduce”
- “Compute Mean Value with MapReduce”
- “Create Histograms Using MapReduce”
- “Compute Mean by Group Using MapReduce”
- “Simple Data Subsetting Using MapReduce”

- "Using MapReduce to Compute Covariance and Related Quantities"
- "Compute Summary Statistics by Group Using MapReduce"
- "Using MapReduce to Fit a Logistic Regression Model"
- "Tall Skinny QR (TSQR) Matrix Factorization Using MapReduce"

Input Arguments

`ds` — Input datastore

datastore object

Input datastore, specified as a datastore object. Use the `datastore` function to create a datastore object from your data set.

`mapfun` — Function handle to map function

function handle

Function handle to map function. `mapfun` receives chunks from input datastore `ds`, and then uses the `add` and `addmulti` functions to add key-value pairs to an intermediate `KeyValueStore` object. The number of calls to the map function by `mapreduce` is equal to the number of chunks in the `datastore` (the number of chunks is determined by the `ReadSize` property of the datastore).

The inputs to the map function are `data`, `info`, and `intermKVStore`, which `mapreduce` automatically creates and passes to the map function:

- The `data` and `info` inputs are the result of a call to the `read` function of `datastore`, which `mapreduce` executes automatically before each call to the map function.
- `intermKVStore` is the name of the intermediate `KeyValueStore` object to which the map function needs to add key-value pairs. If none of the calls to the map function add key-value pairs to `intermKVStore`, then `mapreduce` does not call the reduce function and the output datastore is empty.

An example of a template for the map function is

```
function myMapper(data, info, intermKVStore)
%do a calculation with the data chunk
add(intermKVStore, key, value)
end
```

Example: `@myMapper`

Data Types: `function_handle`

`reducefun` — Function handle to reduce function

function handle

Function handle to reduce function. `mapreduce` calls `reducefun` once for each unique key added to the intermediate `KeyValueStore` by the map function. In each call, `mapreduce` passes the values associated with the active key to `reducefun` as a `ValueIterator` object. The `reducefun` function loops through the values for each key using the `hasnext` and `getnext` functions. Then, after performing some calculation(s), it writes key-value pairs to the final output.

The inputs to the reduce function are `intermKey`, `intermValIter`, and `outKVStore`, which `mapreduce` automatically creates and passes to the reduce function:

- `intermKey` is the active key from the intermediate `KeyValueStore` object. Each call to the reduce function by `mapreduce` specifies a new unique key from the keys in the intermediate `KeyValueStore` object.
- `intermValIter` is the `ValueIterator` associated with the active key, `intermKey`. This `ValueIterator` object contains all of the values associated with the active key. Scroll through the values using the `hasnext` and `getnext` functions.
- `outKVStore` is the name for the final `KeyValueStore` object to which the reduce function needs to add key-value pairs. `mapreduce` takes the output key-value pairs from `outKVStore` and returns them in the output datastore, `outds`, which is a `KeyValueDatastore` object by default. If none of the calls to the reduce function add final key-value pairs to `outKVStore`, then the output datastore is empty.

An example of a template for the reduce function is

```
function myReducer(intermKey, intermValIter, outKVStore)
while hasnext(intermValIter)
    X = getnext(intermValIter);
    %do a calculation with the current value, X
end
add(outKVStore, key, value)
end
```

Example: `@myReducer`

Data Types: `function_handle`

`mr` — Execution environment

MapReducer object

Execution environment, specified as a MapReducer object. `mr` is the result of a call to the `mapreducer` function. The default `mr` argument is a call to `gcmr`, which uses the default global execution environment for `mapreduce` (in MATLAB the default is `mapreducer(0)`, which returns a `SerialMapReducer` object).

Note: This setting specifies the execution environment for `mapreduce` and is not necessary to run `mapreduce` on your local computer. For more information, see "Speed Up and Deploy MapReduce Using Other Products".

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `outds = mapreduce(ds, @mapfun, @reducefun, 'Display', 'off', 'OutputFolder', 'C:\Users\username\Desktop')`

`'OutputType'` — Type of datastore output

`'Binary'` (default) | `'TabularText'`

Type of datastore output, specified as `'Binary'` or `'TabularText'`. The default setting of `'Binary'` returns a `KeyValueDatastore` output datastore that points to binary (`.mat` or `.seq`) files in the output folder. The `'TabularText'` option returns a `TabularTextDatastore` output datastore that points to `.txt` files in the output folder.

The table provides the details for each of the output types.

`'OutputType'`	Type of datastore output	Datastore points to files of type	Values that the Reduce function can add	Keys that the Reduce function can add	Details
`'Binary'` (default)	`KeyValueDat`	`.mat` (or `.seq` when running	Any valid MATLAB object.	Strings, or numeric scalars that	N/A

'OutputType	Type of datastore output	Datastore points to files of type	Values that the Reduce function can add	Keys that the Reduce function can add	Details
		against Hadoop).		are not NaN, complex, logical, or sparse.	
'TabularTex	TabularTextD	.txt	Strings, or numeric scalars that are not NaN, complex, logical, or sparse.	Strings, or numeric scalars that are not NaN, complex, logical, or sparse.	• File is UTF-8 encoded. • Keys and values are tab (\t) separated. • The row delimiter is \r\n on Windows, and \n on Linux and Mac.

'OutputFolder' — Destination folder of mapreduce output
pwd (default) | file path

Destination folder for mapreduce output, specified as a file path. The default output folder is the current folder, pwd. You can specify a different path with a fully qualified path or with a path relative to the current folder.

Example: mapreduce(..., 'OutputFolder', 'MyOutputFolder\Results') specifies a file path relative to the current folder for the output.

'Display' — Toggle for command line progress output
'on' (default) | 'off'

Toggle for command line progress output, specified as 'on' or 'off'. The default is 'on', so that mapreduce displays progress information in the command window during the map and reduce phases of execution.

Output Arguments

`outds` — Output datastore

`KeyValueDatastore` (default) | `TabularTextDatastore`

Output datastore, returned as a `KeyValueDatastore` or `TabularTextDatastore` object. By default, `outds` is a `KeyValueDatastore` object that points to `.mat` files in the current folder. Use the `Name,Value` pair arguments for `'OutputType'` and `'OutputFolder'` to return a `TabularTextDatastore` object or change the location of the output files, respectively.

`mapreduce` does not sort the key-value pairs in `outds`. Their order may differ when using other products with `mapreduce`.

To view the contents of `outds`, use the `preview`, `read`, or `readall` functions of `datastore`.

More About

Tips

- Debugging your `mapreduce` algorithms to examine how key-value pairs move through the different phases is always useful. To examine the movement of data, set breakpoints in your map and reduce functions. The breakpoints stop execution of `mapreduce`, allowing you to examine the current status of relevant variables, like the `KeyValueStore` or `ValueIterator`. For more information, see "Debug MapReduce Algorithms".
- Some recommendations to optimize `mapreduce` performance on any platform are:
  - Minimize the number of calls to the map function. The easiest approach is to increase the value of the `ReadSize` property of the input datastore. The result is that `mapreduce` passes larger chunks of data to the map function, and the datastore depletes with fewer reads.
  - Decrease the amount of intermediate data sent between map and reduce functions. One approach is to use `unique` inside a map function to combine similar keys. See "Compute Mean by Group Using MapReduce" for an example of this technique.

- "Getting Started with MapReduce"
- "Write a Map Function"

- “Write a Reduce Function”
- “Speed Up and Deploy MapReduce Using Other Products”
- “Build Effective Algorithms with MapReduce”

See Also

`datastore` | `gcmr` | Using KeyValueStore Objects | `mapreducer` | Using ValueIterator Objects

Introduced in R2014b

mapreducer

Define execution environment for mapreduce

`mapreducer` is the execution configuration function for `mapreduce`. Use this function to set, change, or store the execution environment to run `mapreduce` using Parallel Computing Toolbox, MATLAB Distributed Computing Server, or MATLAB Compiler.

Syntax

```
mapreducer
mapreducer(0)
mapreducer(mr)
mr = mapreducer( ___ )
mr = mapreducer( ___ ,'ObjectVisibility','Off')
```

Description

`mapreducer` sets the global execution environment for `mapreduce` to be the default (`SerialMapReducer` in MATLAB).

`mapreducer(0)` sets the global execution environment for `mapreduce` to be the local MATLAB session.

`mapreducer(mr)` sets the global execution environment for `mapreduce` using a previously created MapReducer object, `mr`.

`mr = mapreducer( ___ )` also returns a MapReducer object using any of the previous syntaxes. Use `mr` as a fourth input argument to `mapreduce` when you want to explicitly specify the execution environment.

`mr = mapreducer( ___ ,'ObjectVisibility','Off')` toggles the visibility of the MapReducer object, `mr`, using any of the previous syntaxes. Use this syntax to create new MapReducer objects without affecting the global execution environment of `mapreduce`.

Note: In MATLAB, the `mapreduce` function automatically runs using a `SerialMapReducer`, and it is unnecessary to specify configuration settings using

`mapreducer`. However, if you have Parallel Computing Toolbox, MATLAB Distributed Computing Server, or MATLAB Compiler, then additional `mapreducer` configuration options are available for running in parallel or deployed environments. For more information, see the documentation in each product.

Output Arguments

`mr` — Execution environment for `mapreduce`

MapReducer object

Execution environment for `mapreduce`, returned as a MapReducer object.

If the `ObjectVisibility` property of `mr` is set to `'On'` (the default), then `mr` defines the default execution environment for `mapreduce`. You can pass `mr` to the `mapreduce` function to explicitly specify the execution environment, even if its `ObjectVisibility` property is set to `'Off'`.

Pass `mr` to the `mapreduce` function when using the following products:

- Parallel Computing Toolbox enables you to run `mapreduce` computations using a parallel pool.
- MATLAB Distributed Computing Server extends Parallel Computing Toolbox, enabling you to run `mapreduce` computations on a remote cluster. This includes support for Hadoop.
- MATLAB Compiler enables you to create standalone applications or deployable archives, which you can share with colleagues or deploy to production Hadoop systems.

More About

Tips

- If you have Parallel Computing Toolbox, see the `mapreducer` function reference page for additional information.
- If you have MATLAB Compiler, see the `mapreducer` function reference page for additional information.
- “Speed Up and Deploy MapReduce Using Other Products”

See Also

`gcmr` | `mapreduce`

Introduced in R2014b

containers.Map class

Package: containers

Map values to unique keys

Description

A `Map` object is a data structure that allows you to retrieve values using a corresponding key. Keys can be real numbers or text strings and provide more flexibility for data access than array indices, which must be positive integers. Values can be scalar or nonscalar arrays.

Construction

`mapObj = containers.Map` constructs an empty `Map` container `mapObj`.

`mapObj = containers.Map(keySet,valueSet)` constructs a `Map` that contains one or more values and a unique key for each value.

`mapObj = containers.Map(keySet,valueSet,'UniformValues',isUniform)` specifies whether all values must be uniform (either all scalars of the same data type, or all strings). Possible values for `isUniform` are logical `true` (`1`) or `false` (`0`).

`mapObj = containers.Map('KeyType',kType,'ValueType',vType)` constructs an empty `Map` object and sets the `KeyType` and `ValueType` properties. The order of the key type and value type argument pairs is not important, but both pairs are required.

Input Arguments

`keySet`

1-by-`n` array that specifies `n` unique keys for the map.

All keys in a `Map` object are real numeric values or all keys are strings. If `n` > 1 and the keys are strings, `keySet` must be a cell array. The number of keys in `keySet` must equal the number of values in `valueSet`.

`valueSet`

1-by-`n` array of any class that specifies `n` values for the map. The number of values in `valueSet` must equal the number of keys in `keySet`.

`'UniformValues'`

Parameter string to use with the `isUniform` argument.

`isUniform`

Logical value that specifies whether all values are uniform. If `isUniform` is `true` (`1`), all values must be scalars of the same data type, or all values must be strings. If `isUniform` is `false` (`0`), then `containers.Map` sets the `ValueType` to `'any'`.

Default: `true` for empty `Map` objects, otherwise determined by the data types of values in `valueSet`.

`'KeyType'`

Parameter string to use with the `kType` argument.

`kType`

String that specifies the data type for the keys. Possible values are `'char'`, `'double'`, `'single'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Default: `'char'` for empty `Map` objects, otherwise determined by the data types of keys in `keySet`. If you specify keys of different numeric types, `kType` is `'double'`.

`'ValueType'`

Literal string parameter to use with the `vType` argument.

`vType`

String that specifies the data type for the values. Possible values are `'any'`, `'char'`, `'logical'`, `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Default: `'any'` when you create an empty `Map` object or when you specify values of different sizes or types, otherwise determined by the data type of `valueSet`.

Properties

`Count`

Unsigned 64-bit integer that represents the total number of key-value pairs contained in the `Map` object. Read only.

`KeyType`

Character array that indicates the data type of all keys in the `Map` object. The default `KeyType` for empty `Map` objects is `'char'`. Otherwise, `KeyType` is determined from the data type of the `keySet` inputs. Read only.

`ValueType`

Character array that indicates the data type of all values in the `Map` object. If you construct an empty `Map` object or specify values with different data types, then the value of `ValueType` is `'any'`. Otherwise, `ValueType` is determined from the data type of the `valueSet` inputs. Read only.

Methods

isKey	Determine if `containers.Map` object contains key
keys	Identify keys of `containers.Map` object
length	Length of `containers.Map` object
remove	Remove key-value pairs from `containers.Map` object
size	Size of `containers.Map` object
values	Identify values in `containers.Map` object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a Map and View Properties

Construct a `Map` object that contains rainfall data for several months:

```
keySet =   {'Jan', 'Feb', 'Mar', 'Apr'};
valueSet = [327.2, 368.2, 197.6, 178.4];
mapObj = containers.Map(keySet,valueSet)
```

This code returns a description of the map, including the property values:

```
mapObj =

  containers.Map handle
  Package: containers

  Properties:
        Count: 4
      KeyType: 'char'
    ValueType: 'double'

  Methods, Events, Superclasses
```

Get a specific property using dot notation, such as

```
mapObj.Count
```

which returns

```
ans =
                    4
```

Look Up Values in a Map

Use the map created in the previous example to find the rainfall data for February:

```
rainFeb = mapObj('Feb')
```

This code returns

```
rainFeb =
  368.2000
```

Add a Single Value and Key to a Map

Add data for the month of May to the map created in the first example:

```
mapObj('May') = 100.0;
```

Add Multiple Values and Keys by Concatenating Maps

Create a map that contains rainfall data for June, July, and August, and add the data to `mapObj` (from previous examples):

```
keySet   = {'Jun','Jul','Aug'};
valueSet = [ 69.9, 32.3, 37.3];
newMap = containers.Map(keySet,valueSet);

mapObj = [mapObj; newMap];
```

Map objects only support vertical concatenation (that is, adding columns with a semicolon, `;`). When concatenating maps, the data type of all values must be consistent with the `ValueType` of the leftmost map. In this example, both maps have the a `ValueType` of `double`.

Get the Keys or Values in a Map

Determine all the keys of `mapObj` (from previous examples) by calling the `keys` method:

```
allKeys = keys(mapObj)
```

This method returns the keys in alphabetical order:

```
allKeys =
    'Apr'    'Aug'    'Feb'    'Jan'    'Jul'    'Jun'    'Mar'    'May'
```

Get multiple values from the map by calling the `values` method. Like the `keys` method, you can request all values with the syntax `values(mapObj)`. Alternatively, request values for specific keys. For example, view the values for March, April, and May in `mapObj`:

```
springValues = values(mapObj,{'Mar','Apr','May'})
```

This method returns the values in a cell array, in the order corresponding to the specified keys:

```
springValues =
    [197.6000]    [178.4000]    [100]
```

Remove Keys and Values

Remove the data for March and April from `mapObj` (from previous examples) by calling the `remove` method, and view the remaining keys:

```
remove(mapObj,{'Mar','Apr'});
keys(mapObj)
```

This code returns

```
ans =
    'Aug'    'Feb'    'Jan'    'Jul'    'Jun'    'May'
```

Create a Map with Nonscalar Values

Map integer keys to nonscalar arrays, and view the value for one of the keys:

```
keySet = [5,10,15];
valueSet = {magic(5),magic(10),magic(15)};
mapObj = containers.Map(keySet,valueSet);
mapObj(5)
```

This code returns

```
ans =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Construct an Empty Map

Construct a map with no values, but set the `KeyType` and `ValueType` properties:

```
mapObj = containers.Map('KeyType','char','ValueType','int32')
```

This code returns

```
mapObj =

  containers.Map handle
  Package: containers

  Properties:
        Count: 0
      KeyType: 'char'
    ValueType: 'int32'

  Methods, Events, Superclasses
```

Specify Whether Values Are Uniform

Construct a map with numeric values, and specify that the values do not have to be uniform:

```
keySet = {'a','b','c'};
valueSet = {1,2,3};
mapObj = containers.Map(keySet,valueSet,'UniformValues',false);
```

This map allows nonnumeric values, so

```
mapObj('d') = 'OK';
values(mapObj)
```

returns

```
ans =
    [1]    [2]    [3]    'OK'
```

See Also

isKey | values | keys

mat2cell

Convert array to cell array with potentially different sized cells

Syntax

```
C = mat2cell(A,dim1Dist,...,dimNDist)
C = mat2cell(A,rowDist)
```

Description

`C = mat2cell(A,dim1Dist,...,dimNDist)` divides array `A` into smaller arrays within cell array `C`. Vectors `dim1Dist,...dimNDist` specify how to divide the rows, columns, and (when applicable) higher dimensions of `A`.

`C = mat2cell(A,rowDist)` divides array `A` into an `n`-by-1 cell array `C`, where `n == numel(rowDist)`.

Input Arguments

`A`

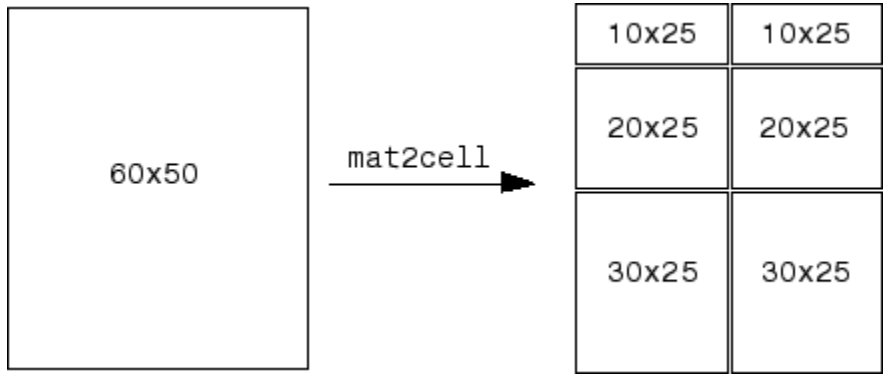
Any type of array.

`dim1Dist,...,dimNDist`

Numeric vectors that describe how to divide each dimension of `A`. For example, this command

```
c = mat2cell(x, [10, 20, 30], [25, 25])
```

divides a 60-by-50 array into six arrays contained in a cell array.



For the *k*th dimension, `sum(dim`*k*`Dist) == size(A, `*k*`)`.

If the *k*th dimension of `A` is zero, set the corresponding `dim`*k*`Dist` to the empty array, `[]`. For example,

```
a = rand(3, 0, 4);
c = mat2cell(a, [1, 2], [], [2, 1, 1]);
```

`rowDist`

Numeric vector that describes how to divide the rows of `A`. When you do not specify distributions for any other dimension, the `mat2cell` function creates an `n`-by-1 cell array `C`, where `n == numel(rowDist)`.

Output Arguments

`C`

Cell array. The *k*th dimension of array `C` is given by `size(C, `*k*`) == numel(dim`*k*`Dist)`. The *k*th dimension of the *i*th cell of `C` is given by `size(C{i}, `*k*`) == dim`*k*`Dist(i)`.

Examples

Divide the 5-by-4 matrix `X` into 2-by-3 and 2-by-2 matrices contained in a cell array.

```
X = reshape(1:20,5,4)'
```

```
C = mat2cell(X, [2 2], [3 2])
celldisp(C)
```

This code returns

```
X =
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20

C =
    [2x3 double]    [2x2 double]
    [2x3 double]    [2x2 double]

C{1,1} =
     1     2     3
     6     7     8

C{2,1} =
    11    12    13
    16    17    18

C{1,2} =
     4     5
     9    10

C{2,2} =
    14    15
    19    20
```

Divide `X` (created in the previous example) into a 2-by-1 cell array.

```
C = mat2cell(X, [1 3])
celldisp(C)
```

This code returns

```
C =
    [1x5 double]
    [3x5 double]

C{1} =
     1     2     3     4     5
```

```
C{2} =
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
```

See Also

`cell2mat` | `num2cell`

Introduced before R2006a

mat2str

Convert matrix to string

Syntax

```
str = mat2str(A)
str = mat2str(A,n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

Description

`str = mat2str(A)` converts matrix `A` into a string. This string is suitable for input to the `eval` function such that `eval(str)` produces the original matrix to within 15 digits of precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

`str = mat2str(A, 'class')` creates a string with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` digits of precision and includes the class information.

Limitations

The `mat2str` function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if `A` is a multidimensional array.

Examples

Example 1

Consider the matrix

```
x = [3.85 2.91; 7.74 8.99]
x =
    3.8500    2.9100
    7.7400    8.9900
```

The statement

```
A = mat2str(x)
```

produces

```
A =
   [3.85 2.91;7.74 8.99]
```

where `A` is a string of 21 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(x))` reproduces `x`.

Example 2

Create a 1-by-6 matrix of signed 16-bit integers, and then use `mat2str` to convert the matrix to a 1-by-33 character array, `A`. Note that output string `A` includes the class name, `int16`:

```
x1 = int16([-300 407 213 418 32 -125]);

A = mat2str(x1, 'class')
A =
   int16([-300 407 213 418 32 -125])

class(A)
ans =
   char
```

Evaluating the string `A` gives you an output `x2` that is the same as the original `int16` matrix:

```
x2 = eval(A);

if isnumeric(x2) && isa(x2, 'int16') && all(x2 == x1)
   disp 'Conversion back to int16 worked'
end
```

```
Conversion back to int16 worked
```

See Also

num2str | int2str | str2num | sprintf | fprintf

Introduced before R2006a

material

Control reflectance properties of surfaces and patches

Syntax

```
material shiny
material dull
material metal
material([ka kd ks])
material([ka kd ks n])
material([ka kd ks n sc])
material default
```

Description

`material` sets the lighting characteristics of `surface` and `patch` objects.

`material shiny` sets the reflectance properties so that the object has a high specular reflectance relative to the diffuse and ambient light, and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light and has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects.

`material default` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects to their defaults.

More About

Tips

The `material` command sets the `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, and `SpecularColorReflectance` properties of all `surface` and `patch` objects in the axes. There must be visible `light` objects in the `axes` for lighting to be enabled. Look at the `materal.m` file to see the actual values set (enter the command `type material`).

- "Lighting Overview"

See Also

`lighting` | `patch` | `light` | `surface`

Introduced before R2006a

matfile

Access and change variables directly in MAT-files, without loading into memory

Syntax

```
m = matfile(filename)
m = matfile(filename,'Writable',isWritable)
```

Description

`m = matfile(filename)` creates a MAT-file object, `m`, connected to the MAT-file named `filename`. The object allows you to access and change variables directly in a MAT-file, without having to load the variables into memory.

The partial loading and saving that the `matfile` function provides requires less memory than the `load` and `save` commands, which always operate on entire variables.

`m = matfile(filename,'Writable',isWritable)` enables or disables write access to the file.

Examples

Load Entire Variable

Load variable `topo` from the example file, `topography.mat`.

Open the example MAT-file, `topography.mat`.

```
filename = 'topography.mat';
m = matfile(filename);
```

Read the variable `topo` from the MAT-file.

```
topo = m.topo;
```

MATLAB loads the entire variable, `topo`, into the workspace.

Save Entire Variable to Existing MAT-file

Generate a 20-by-20 example array, `x`, and save it to a MAT-file called `myFile.mat`.

```
x = magic(20);
save('myFile.mat','x');
```

Create a MAT-file object connected to the existing MAT-file named `myFile.mat`. Enable write access to the MAT-file by setting `Writable` to `true`.

```
m = matfile('myFile.mat','Writable',true);
```

Generate a 15-by-15 example array, `y`.

```
y = magic(15);
```

Save `y` to the MAT-file. Specify the variable in the MAT-file using dot notation similar to accessing fields of structure arrays.

```
m.y = y;
```

MATLAB adds a variable named `y` to the file.

Display all variables stored in the MAT-file, `myFile.mat`.

```
whos('-file','myFile.mat')
```

```
  Name        Size              Bytes  Class     Attributes

  x          20x20              3200  double
  y          15x15              1800  double
```

Load and Save Parts of Variables

Access specific elements of a MAT-file variable.

Open a new MAT-file, `myFile2.mat`.

```
m = matfile('myFile2.mat');
```

Save a 20-by-20 example array to part of a variable, `y`, in `myFile2.mat`. Specify the variable in the MAT-file using dot notation similar to accessing fields of structure arrays.

```
m.y(81:100,81:100) = magic(20);
```

MATLAB inserts the 20-by-20 array into the elements of `y` specified by the indices `(81:100,81:100)`.

Read a subset of array `y` into a new workspace variable, `z`.

```
z = m.y(85:94,85:94);
```

MATLAB reads the 10-by-10 subarray specified by the indices `(85:94,85:94)` from the MAT-file into workspace variable `z`.

Determine Size of Variables

Determine the size of a variable, and then calculate the average of each column.

Open the example MAT-file, `stocks.mat`.

```
filename = 'stocks.mat';
m = matfile(filename);
```

Determine the size of the variable, `stocks`, in `stocks.mat`.

```
[nrows,ncols] = size(m,'stocks');
```

Compute the average of each column of the variable `stocks`.

```
avgs = zeros(1,ncols);
for i = 1:ncols
    avgs(i) = mean(m.stocks(:,i));
end
```

Enable Write Access to MAT-file

Enable write access to the MAT-file, `myFile.mat`, by setting `Writable` to `true` when you open the MAT-file.

```
filename = 'myFile.mat';
m = matfile(filename,'Writable',true);
```

Alternatively, set `Properties.Writable` in a separate step after you open the MAT-file.

```
m.Properties.Writable = true;
```

Input Arguments

`filename` — Name of MAT-file

string

Name of a MAT-file, specified as a string. If the file is not in the current folder, `filename` must include a full or a relative path. If `filename` does not include an extension, then `matfile` appends `.mat`.

If the file does not exist, then `matfile` creates a Version 7.3 MAT-file on the first assignment to a variable.

`matfile` only supports efficient partial loading and saving for MAT-files in Version 7.3 format. If you index into a variable in a Version 7 (the current default) or earlier MAT-file, MATLAB warns and temporarily loads the entire contents of the variable.

Example: `'myFile.mat'`

Data Types: `char`

`isWritable` — Write access to MAT-file

`true` | `false`

Write access to the MAT-file, specified as either `true` or `false`.

- `true` enables saving to the MAT-file. If the file is read only, MATLAB changes the system permissions with the `fileattrib` function.
- `false` disables saving to the MAT-file. MATLAB does not change the system permissions.

The default value is `true` for new files, and `false` for existing files.

Data Types: `logical`

Output Arguments

`m` — MAT-file object

`matlab.io.MatFile` object

MAT-file object connected to a MAT-file.

Access variables in the MAT-file with dot notation similar to accessing fields of structure arrays:

- To load part of variable `varName` from the MAT-file corresponding to `m`, call:

  ```
  loadedData = m.varName(indices);
  ```

- To save part of variable `varName` to the MAT-file corresponding to `m`, call:

  ```
  m.varName(indices) = dataToSave;
  ```

When accessing variables, specify indices for all dimensions. Indices can be a single value, an equally spaced range of increasing values, or a colon `(:)`; for example:

```
m.varName(100:500,200:600)
m.varName(:,501:1000)
m.varName(1:2:1000,80)
```

Limitations

- `matfile` does not support linear indexing. You must specify indices for all dimensions.
- `matfile` does not support indexing into:
  - Variables of tables
  - Cells of cell arrays
  - Fields of structure arrays
  - User-defined classes
  - Sparse arrays
- You cannot assign complex values to an indexed portion of a real array.
- You cannot evaluate function handles using the `m` output. For example, if your MAT-file contains function handle `myfunc`, the syntax `m.myfunc()` attempts to index into the function handle, and does not invoke the function.
- Efficient partial loading and saving requires Version 7.3 MAT-files. To create a Version 7.3 MAT-file, call the `save` function with the `'-v7.3'` option. For example, to convert and existing MAT-file named `durer.mat` to Version 7.3, call:

  ```
  load('durer.mat');
  ```

```
save('mycopy_durer.mat','-v7.3');
```

More About

Tips

- Using the `end` keyword as part of an index causes MATLAB to load the entire variable into memory. For very large variables, this load operation results in `Out of Memory` errors. Rather than using `end`, determine the extent of a variable, `myVar`, with the `size` method, such as:

  ```
  sizeMyVar = size(m,'myVar')
  ```

- "Load Parts of Variables from MAT-Files"
- "Save Parts of Variables to MAT-Files"
- MAT-File Versions

See Also

`load` | `save` | `size` | `whos`

Introduced in R2011b

matlab.codetools.requiredFilesAndProducts

List dependencies of MATLAB program files

Syntax

```
fList = matlab.codetools.requiredFilesAndProducts(files)
[fList, pList] = matlab.codetools.requiredFilesAndProducts(files)
[fList, pList] =
matlab.codetools.requiredFilesAndProducts( ___ ,'toponly')
```

Description

`fList = matlab.codetools.requiredFilesAndProducts(files)` returns a list of the MATLAB program files required to run the program files specified by `files`.

`[fList, pList] = matlab.codetools.requiredFilesAndProducts(files)` also returns a list of the MathWorks products possibly required to run the program files specified by `files`.

`[fList, pList] = matlab.codetools.requiredFilesAndProducts( ___ ,'toponly')` indicates that for a file or product to be included in the output, it must be used directly by at least one file specified in `files`. The `'toponly'` input string is case insensitive.

Examples

Identify Required Files and Products for MATLAB Toolbox Function

Determine the required files and products for the `edge` function in the Image Processing Toolbox.

```
[fList,pList] = matlab.codetools.requiredFilesAndProducts('edge.m')
```

```
fList =

     {}
```

```
pList =

1x2 struct array with fields:

    Name
    Version
    ProductNumber
    Certain
```

There are no required MATLAB files, but there are two required products.

List the required products.

```
{pList.Name}'
```

```
ans =

    'MATLAB'
    'Image Processing Toolbox'
```

Identify Required Files and Products for Your MATLAB Program Files

In your current working folder, create a function in the file `getRandomNumber.m`.

```
function a = getRandomNumber
    rng shuffle
    a = rand;
end
```

Now, at the command line, determine the required files and products for `getRandomNumber.m`.

```
[fList,pList] = matlab.codetools.requiredFilesAndProducts('getRandomNumber.m')
```

```
fList =

    'C:\work\getRandomNumber.m'


pList =

           Name: 'MATLAB'
        Version: '8.5'
```

```
    ProductNumber: 1
          Certain: 1
```

The only file required to run the `getRandomNumber` function is the function file itself. The only required MathWorks product is MATLAB.

In your current working folder, create a function in the file `displayNumber.m`.

```
function displayNumber
    a = getRandomNumber;
    disp(['Your number is ' num2str(a)])
end
```

Now, at the command line, determine the required files and products for `displayNumber.m`.

```
[fList,pList] = matlab.codetools.requiredFilesAndProducts('displayNumber.m')
```

```
fList =

    'C:\work\displayNumber.m'    'C:\work\getRandomNumber.m'


pList =

             Name: 'MATLAB'
          Version: '8.5'
    ProductNumber: 1
          Certain: 1
```

In addition to the function file itself, the `displayNumber` function requires the `getRandomNumber.m` file. The only required MathWorks product is MATLAB.

Identify Top-Level Dependencies Only

In your current working folder, create a handle class in the file `ExampleHandle.m`.

```
classdef ExampleHandle < handle
    % class content
end
```

In your current working folder, create a class in the file `AnotherExampleHandle.m` that inherits from `ExampleHandle`.

```
classdef AnotherExampleHandle < ExampleHandle
```

```
    % class content
end
```

In your current working folder, create a function in the file `getHandles.m` that instantiates `AnotherExampleHandle` objects.

```
function [h1,h2] = getHandles()
    h1 = AnotherExampleHandle;
    h2 = AnotherExampleHandle;
end
```

Now, at the command line, determine the required files for `getHandles.m`.

```
[fList,~] = matlab.codetools.requiredFilesAndProducts('getHandles.m');
fList'

ans =

    'C:\work\AnotherExampleHandle.m'
    'C:\work\ExampleHandle.m'
    'C:\work\getHandles.m'
```

Determine the required files that are directly required for `getHandles.m`.

```
[fList,~] = matlab.codetools.requiredFilesAndProducts('getHandles.m','toponly')

fList =

    'C:\work\AnotherExampleHandle.m'    'C:\work\getHandles.m'
```

Although `AnotherExampleHandle.m` requires `ExampleHandle.m`, that file is not a direct requirement for `getHandles.m`.

Input Arguments

`files` — List of files for analysis

string or cell array of strings

List of files for analysis, specified as a string or cell array of strings. Each string is the name of a single MATLAB program file. For example, `files` is a list of MATLAB program files that you intend to provide to other users. The `matlab.codetools.requiredFilesAndProducts` function provides you with requirements information to pass along with your files.

To ensure an accurate dependency report, `files` and dependencies must be on the MATLAB path. `matlab.codetools.requiredFilesAndProducts` does not return information about dependent files not on the path.

Example: `'myFile.m'` or `'C:\Program Files\MATLAB\R2014a\my_work\myFile.m'`

Example: `{'myFile.m','myOtherFile.m'}`

Example: `cellstr(ls('*.m'))`

Output Arguments

`fList` — List of user-authored MATLAB program files

cell array of strings

List of user-authored MATLAB program files required by `files`, returned as a cell array of strings. Each string indicates the full path of the required file. `fList` does not include built-in MATLAB files, since these files are installed with the products listed in `pList`.

`pList` — List of MathWorks products

structure or array of structures

List of MathWorks products possibly required by `files`, returned as a structure or array of structures. Each product is described by name (`Name` field), version (`Version` field), product number (`ProductNumber` field), and a certainty indicator (`Certain` field). The `Certain` field has a value of 1 if `matlab.codetools.requiredFilesAndProducts` determines the product is required by the specified program files, `files`, or a value of 0 if the product is possibly required.

The `matlab.codetools.requiredFilesAndProducts` function is intended to provide you with information to pass on to consumers of your MATLAB program files. The version numbers indicate the version of the products you have installed when you execute the function. `Version` is not an indicator of backward compatibility.

More About

- "Identify Program Dependencies"

Introduced in R2014a

matlab.io.MatFile class

Package: matlab.io

Load and save parts of variables in MAT-files

Description

The `matfile` function constructs a `matlab.io.MatFile` object that corresponds to a MAT-File, such as

```
matObj = matfile('myFile.mat')
```

Access variables in the MAT-file as properties of `matObj`, with dot notation similar to accessing fields of structs. The syntax for loading and saving part of variable `varName` in the MAT-file corresponding to `matObj` is:

```
loadedData = matObj.varName(indices);    % Load into workspace
matObj.varName(indices) = dataToSave;    % Save to file
```

When indexing, specify indices for all dimensions. Indices can be a single value, an equally spaced range of increasing values, or a colon `(:)`, such as

```
matObj.varName(100:500, 200:600)
matObj.varName(:, 501:1000)
matObj.varName(1:2:1000, 80)
```

Limitations

- Using the `end` keyword as part of an index causes MATLAB to load the entire variable into memory. For very large variables, this load operation results in `Out of Memory` errors. Rather than using `end`, determine the extent of a variable with the `size` method, such as:

  ```
  sizeMyVar = size(matObj,'myVar')
  ```
- `matfile` does not support linear indexing. You must specify indices for all dimensions.
- `matfile` does not support indexing into:
  - Cells of cell arrays

  - Fields of structs
  - User-defined classes
  - Sparse arrays
- You cannot assign complex values to an indexed portion of a real array.
- You cannot evaluate function handles using a `MatFile` object. For example, if your MAT-file contains function handle `myfunc`, the syntax `matObj.myfunc()` attempts to index into the function handle, and does not invoke the function.

Construction

`matObj = matfile(filename)` constructs a `matlab.io.MatFile` object that can load or save parts of variables in MAT-file `filename`. MATLAB does not load any data from the file into memory when creating the object.

`matObj = matfile(filename,'Writable',isWritable)` enables or disables write access to the file for object `matObj`. Possible values for `isWritable` are logical `true` (1) or `false` (0).

Input Arguments

`filename`

String enclosed in single quotation marks that specifies the name of a MAT-file.

`filename` can include a full or partial path, otherwise `matfile` searches for the file along the MATLAB search path. If `filename` does not include an extension, `matfile` appends `.mat`.

If the file does not exist, `matfile` creates a Version 7.3 MAT-file on the first assignment to a variable.

`matfile` only supports partial loading and saving for MAT-files in Version 7.3 format (described in MAT-File Versions). If you index into a variable in a Version 7 (the current default) or earlier MAT-file, MATLAB warns and temporarily loads the entire contents of the variable.

`'Writable'`

Parameter to use with the `isWritable` argument.

isWritable

Logical value that specifies whether to allow saving to the file. Possible values:

`true (1)`	Enable saving. If the file is read only, change the system permissions with `fileattrib`.
`false (0)`	Disable saving with `matfile`. MATLAB does not change the system permissions.

Default: `true` for new files, `false` for existing files

Properties

Properties.Source

String that contains the fully qualified path to the file. Read only.

Properties.Writable

Logical value that specifies whether to allow saving to the file. Possible values:

`true (1)`	Enable saving. If the file is read only, change the system permissions with `fileattrib`.
`false (0)`	Disable saving with `matfile`. MATLAB does not change the system permissions.

Default: `true` for new files, `false` for existing files

Methods

size	Array dimensions
who	Names of variables in MAT-file
whos	Names, sizes, and types of variables in MAT-file

You cannot access help for these methods using the `help` command. Find help on the methods from the command line using the `doc` command, such as `doc matlab.io.MatFile/size`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create `myFile.mat` in a temporary folder and save data to part of variable `savedVar`:

```
filename = fullfile(tempdir,'myFile.mat');
matObj = matfile(filename);
matObj.savedVar(81:100,81:100) = magic(20);
```

Load part of the data into variable `loadVar`:

```
loadVar = matObj.savedVar(85:94,85:94);
```

Load or save an entire variable by omitting the indices. For example, load variable `topo` from `topography.mat`:

```
filename = 'topography.mat';
matObj = matfile(filename);
topo = matObj.topo;
```

Determine the dimensions of a variable, and process one part of the variable at a time. In this case, calculate and store the average of each column of variable `stocks` in the example file `stocks.mat`:

```
filename = 'stocks.mat';
matObj = matfile(filename);
[nrows, ncols] = size(matObj,'stocks');

avgs = zeros(1,ncols);
for idx = 1:ncols
    avgs(idx) = mean(matObj.stocks(:,idx));
end
```

By default, `matfile` only supports loading data from existing files. To enable saving, set `Writable` to `true` either during construction of the object,

```
filename = 'myFile.mat';
matObj = matfile(filename,'Writable',true);
```

or in a separate step, by setting `Properties.Writable`:

```
filename = 'myFile.mat';
matObj = matfile(filename);
matObj.Properties.Writable = true;
```

See Also

`load` | `save` | `size` | `whos`

matlab.io.fits.closeFile

Close FITS file

Syntax

```
closeFile(fptr)
```

Description

`closeFile(fptr)` closes an open FITS file.

This function corresponds to the `fits_close_file (ffclos)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits','READONLY');
fits.closeFile(fptr);
```

See Also

`createFile` | `openFile`

matlab.io.fits.createFile

Create FITS file

Syntax

```
fptr = createFile(filename)
```

Description

`fptr = createFile(filename)` creates a FITS file. An error will be returned if the specified file already exists, unless the filename is prefixed with an exclamation point (!). In that case CFITSIO will overwrite (delete) any existing file with the same name.

This function corresponds to the `fits_create_file (ffinit)` function in the CFITSIO library C API.

Examples

Create a new FITS file, overwriting any existing file by the same name.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'uint8',[256 512]);
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

See Also

`closeFile` | `createImg` | `createTbl` | `openFile`

matlab.io.fits.deleteFile

Delete FITS file

Syntax

```
deleteFile(fptr)
```

Description

`deleteFile(fptr)` closes and deletes an open FITS file. This can be useful if a FITS file cannot be properly closed.

This function corresponds to the `fits_delete_file (ffdelt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fptr = fits.openFile('myfile.fits','readwrite');
fits.deleteFile(fptr);
fptrs = fits.getOpenFiles()
```

See Also

`closeFile` | `createFile`

matlab.io.fits.fileName

Name of FITS file

Syntax

```
name = fileName(fptr)
```

Description

`name = fileName(fptr)` returns the name of the FITS file associated with the file handle.

This function corresponds to the `fits_file_name (ffflnm)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits','READONLY');
name = fits.fileName(fptr);
fits.closeFile(fptr);
```

See Also

`createFile` | `openFile`

matlab.io.fits.fileMode

I/O mode of FITS file

Syntax

```
mode = fileMode(fptr)
```

Description

`mode = fileMode(fptr)` returns the I/O mode of the opened FITS file. Possible values returned for `mode` are `'READONLY'` or `'READWRITE'`.

This function corresponds to the `fits_file_mode (ffflmd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
mode = fits.fileMode(fptr);
fits.closeFile(fptr);
```

See Also

`createFile` | `openFile`

matlab.io.fits.openFile

Open FITS file

Syntax

```
fptr = openFile(filename)
fptr = openFile(filename,mode)
```

Description

`fptr = openFile(filename)` opens a existing FITS file in read-only mode and returns a file pointer, `fptr`, which references the primary array (first header data unit, or "HDU").

`fptr = openFile(filename,mode)` opens a existing FITS file according to the `mode`, which describes the type of access. `mode` may be either `'READONLY'` or `'READWRITE'`.

This function corresponds to the `fits_open_file (ffopen)` function in the CFITSIO library C API.

Examples

Open a file in read-only mode and read image data from the primary array.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
imagedata = fits.readImg(fptr);
fits.closeFile(fptr);
```

Open a file in read-write mode and add a comment to the primary array.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fptr = fits.openFile('myfile.fits','readwrite');
fits.writeComment(fptr,'This is just a comment.');
```

```
fits.closeFile(fptr);
```

See Also

`closeFile` | `createFile`

matlab.io.fits.createImg

Create FITS image

Syntax

```
createImg(fptr,bitpix,naxes)
```

Description

`createImg(fptr,bitpix,naxes)` creates a new primary image or image extension with a specified datatype `bitpix` and size `naxes`. If the FITS file is currently empty then a primary array is created, otherwise a new image extension is appended to the file.

The first two elements of `naxes` correspond to the NAXIS2 and NAXIS1 keywords, while any additional elements correspond to the NAXIS3, NAXIS4 ... NAXISn keywords.

The datatype `bitpix` may be given as either a CFITSIO name or as the corresponding MATLAB datatype.

`'byte_img'`	`'uint8'`
`'short_img'`	`'int16'`
`'long_img'`	`'int32'`
`'longlong_img'`	`'int64'`
`'float_img'`	`'single'`
`'double_img'`	`'double'`

This function corresponds to the `fits_create_imgll (ffcrimll)` function in the CFITSIO library C API.

Examples

Create two images in a new FITS file. There are 100 rows (NAXIS2 keyword) and 200 columns (NAXIS1 keyword) in the first image, and 256 rows (NAXIS2 keyword), 512 columns (NAXIS1 keyword), and 3 planes (NAXIS3 keyword) in the second image.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'int16',[100 200]);
fits.createImg(fptr,'byte_img',[256 512 3]);
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

See Also

createTbl | insertImg | readImg | setCompressionType | writeImg

matlab.io.fits.getImgSize

Size of image

Syntax

```
imagesize = getImgSize(fptr)
```

Description

`imagesize = getImgSize(fptr)` returns the number of rows and columns of an image. This function corresponds to the `fits_get_img_size (ffgisz)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*;
fptr = fits.openFile('tst0012.fits');
hdus = [1 3 4];
for j = hdus;
    htype = fits.movAbsHDU(fptr,j);
    sz = fits.getImgSize(fptr);
    fprintf('HDU %d:  "%s", [', j, htype);
    for k = 1:numel(sz)
        fprintf(' %d ', sz(k));
    end
    fprintf(']\n');
end
fits.closeFile(fptr);
```

See Also

`createImg` | `getImgType`

matlab.io.fits.getImgType

Data type of image

Syntax

```
datatype = getImgType(fptr)
```

Description

`datatype = getImgType(fptr)` gets the data type of an image. `datatype` can be one of the following strings:

```
'BYTE_IMG'
'SHORT_IMG'
'LONG_IMG'
'LONGLONG_IMG'
'FLOAT_IMG'
'DOUBLE_IMG'
```

This function corresponds to the `fits_get_img_type (ffgidt)` function in the CFITSIO library C API.

Examples

```
fptr = fits.openFile('tst0012.fits');
hdus = [1 3 4];
for j = hdus;
    htype = fits.movAbsHDU(fptr,j);
    dtype = fits.getImgType(fptr);
    fprintf('HDU %d:  "%s", "%s"\n', j, htype, dtype);
end
fits.closeFile(fptr);
```

See Also

`getImgSize`

matlab.io.fits.insertImg

Insert FITS image after current image

Syntax

```
insertImage(fptr,bitpix,naxes)
```

Description

`insertImage(fptr,bitpix,naxes)` inserts a new image extension immediately following the current HDU. If the file has just been created, a new primary array is inserted at the beginning of the file. Any following extensions in the file will be shifted down to make room for the new extension. If the current HDU is the last HDU in the file, then the new image extension will be appended to the end of the file.

This function corresponds to the `fits_insert_imgll (ffiimgll)` function in the CFITSIO library C API.

Examples

Create a 150x300 image between the 1st and 2nd images in a FITS file.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.createImg(fptr,'byte_img',[200 400]);
fits.movAbsHDU(fptr,1);
fits.insertImg(fptr,'byte_img',[150 300]);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','min');
```

See Also

`createImg`

matlab.io.fits.readImg

Read image data

Syntax

```
imgdata = readImg(fptr)
imgdata = readImg(fptr,fpixel,lpixel)
imgdata = readImg(fptr,fpixel,lpixel,inc)
```

Description

`imgdata = readImg(fptr)` reads the entire current image. The number of rows in `imgdata` will correspond to the value of the NAXIS2 keyword, while the number of columns will correspond to the value of the NAXIS1 keyword. Any further dimensions of `imgdata` will correspond to NAXIS3, NAXIS4, and so on.

`imgdata = readImg(fptr,fpixel,lpixel)` reads the subimage defined by pixel coordinates `fpixel` and `lpixel`. The `fpixel` argument is the coordinate of the first pixel and `lpixel` is the coordinate of the last pixel. `fpixel` and `lpixel` are one-based.

`imgdata = readImg(fptr,fpixel,lpixel,inc)` reads the subimage defined by `fpixel`, `lpixel`, and `inc`. The `inc` argument denotes the inter- element spacing along each extent.

This function corresponds to the `fits_read_subset (ffgsv)` function in the CFITSIO library C API.

Examples

Read an entire image.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
data = fits.readImg(fptr);
fits.closeFile(fptr);
```

Read a 70x80 image subset.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
img = fits.readImg(fptr,[11 11],[80 90]);
fits.closeFile(fptr);
```

See Also

`createImg` | `writeImg`

matlab.io.fits.setBscale

Reset image scaling

Syntax

```
setBscale(fptr,BSCALE,BZERO)
```

Description

`setBscale(fptr,BSCALE,BZERO)` resets the scaling factors in the primary array or image extension according to the equation

```
output = (FITS array) * BSCALE + BZERO
```

The inverse formula is used when writing data values to the FITS file.

This only affects the automatic scaling performed when the data elements are read. It does not change the BSCALE and BZERO keyword values.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.setBscale(fptr,2.0,0.5);
data = fits.readImg(fptr);
fits.closeFile(fptr);
```

See Also

`readImg`

matlab.io.fits.writeImg

Write to FITS image

Syntax

```
writeImg(fptr,data)
writeImg(fptr,data,fpixel)
```

Description

`writeImg(fptr,data)` writes an entire image to the FITS data array. The number of rows and columns in `data` must equal the values of the NAXIS2 and NAXIS1 keywords, respectively. Any further extents must correspond to the NAXIS3, NAXIS4 ... NAXISn keywords respectively.

`writeImg(fptr,data,fpixel)` writes a subset of an image to the FITS data array. `fpixel` gives the coordinate of the first pixel in the image region.

This function corresponds to the `fits_write_subset (ffpss)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'long_img',[256 512]);
data = reshape(1:256*512,[256 512]);
data = int32(data);
fits.writeImg(fptr,data);
fits.closeFile(fptr);
```

Create an 80x40 `uint8` image and set all but the outermost pixels to 1.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'uint8',[80 40]);
```

```
data = ones(78,38);
fits.writeImg(fptr,data,[1 1]);
fits.closeFile(fptr);
```

See Also

createImg | readImg

matlab.io.fits.deleteKey

Delete key by name

Syntax

```
deleteKey(fptr,keyname)
```

Description

`deleteKey(fptr,keyname)` deletes a keyword by name.

This function corresponds to the `fits_delete_key (ffdrec)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fprintf('Before key deletion...\n');
fitsdisp('myfile.fits','index',1);
fptr = fits.openFile('myfile.fits','readwrite');
fits.deleteKey(fptr,'DATE');
fits.closeFile(fptr);
fprintf('\n\n\nAfter key deletion...\n');
fitsdisp('myfile.fits','index',1);
```

See Also

`deleteRecord` | `writeKey`

matlab.io.fits.deleteRecord

Delete key by record number

Syntax

```
deleteRecord(fptr,keynum)
```

Description

`deleteRecord(fptr,keynum)` deletes a keyword by record number.

This function corresponds to the `fits_delete_record (ffdrec)` function in the CFITSIO library C API.

Examples

Delete the 18th keyword ("ORIGIN") in a primary array.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fptr = fits.openFile('myfile.fits','readwrite');
card = fits.readRecord(fptr,18);
fits.deleteRecord(fptr,18);
fits.closeFile(fptr);
```

See Also

`deleteKey` | `readRecord`

matlab.io.fits.getHdrSpace

Number of keywords in header

Syntax

```
[nkeys,morekeys] = fits.getHdrSpace(fptr)
```

Description

`[nkeys,morekeys] = fits.getHdrSpace(fptr)` returns the number of existing keywords (not counting the END keyword) and the amount of space currently available for more keywords. It returns `morekeys = -1` if the header has not yet been closed. Note that the CFITSIO library will dynamically add space if required when writing new keywords to a header so in practice there is no limit to the number of keywords that can be added to a header.

This function corresponds to the `fits_get_hdrspace (ffghsp)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
[nkeys,morekeys] = fits.getHdrSpace(fptr);
fits.closeFile(fptr);
```

matlab.io.fits.readCard

Header record of keyword

Syntax

```
card = readCard(fptr,keyname)
```

Description

`card = readCard(fptr,keyname)` returns the entire 80-character header record of the keyword, with any trailing blank characters stripped off.

This function corresponds to the `fits_read_card (ffgcrd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    card = fits.readCard(fptr,'NAXIS');
    fprintf('HDU %d:  ''%s''\n', j, card);
end
fits.closeFile(fptr);
```

See Also

`readKey` | `readRecord`

matlab.io.fits.readKey

Keyword

Syntax

```
[value,comment] = readKey(fptr,keyname)
```

Description

`[value,comment] = readKey(fptr,keyname)` returns the specified key and comment. `value` is returned as a string.

This function corresponds to the `fits_read_key_str (ffgkys)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    [key,comment] = fits.readKey(fptr,'NAXIS');
    fprintf('HDU %d:  NAXIS %s, "%s"\n', j, key, comment);
end
fits.closeFile(fptr);
```

See Also

`readKeyCmplx` | `readKeyDbl` | `readKeyLongLong`

matlab.io.fits.readKeyCmplx

Keyword as complex scalar value

Syntax

```
[value,comment] = readKeyCmplx(fptr,keyname)
```

Description

`[value,comment] = readKeyCmplx(fptr,keyname)` returns the specified key and comment. `value` is returned as a double precision complex scalar value.

This function corresponds to the `fits_read_key_dblcmp" (ffgkym)` function in the CFITSIO library C API.

See Also

`readKey` | `readKeyDbl` | `readKeyLongLong`

matlab.io.fits.readKeyDbl

Keyword as double precision value

Syntax

```
[value,comment] = readKeyDbl(fptr,keyname)
```

Description

`[value,comment] = readKeyDbl(fptr,keyname)` returns the specified key and comment.

This function corresponds to the `fits_read_key_dbl (ffgkyd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    [key,comment] = fits.readKeyDbl(fptr,'NAXIS');
    fprintf('HDU %d:  NAXIS %s, "%s"\n', j, key, comment);
end
fits.closeFile(fptr);
```

See Also

`readKey` | `readKeyCmplx` | `readKeyLongLong`

matlab.io.fits.readKeyLongLong

Keyword as `int64`

Syntax

```
[value,comment] = readKeyLongLong(fptr,keyname)
```

Description

`[value,comment] = readKeyLongLong(fptr,keyname)` returns the specified key and comment. `value` is returned an `int64` scalar value.

This function corresponds to the `fits_read_key_lnglng (ffgkyjj)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    [key,comment] = fits.readKeyLongLong(fptr,'NAXIS');
    fprintf('HDU %d:  NAXIS %d, "%s"\n', j, key, comment);
end
fits.closeFile(fptr);
```

See Also

`readKey` | `readKeyCmplx` | `readKeyDbl`

matlab.io.fits.readKeyLongStr

Long string value

Syntax

```
[value,comment] = readKeyLongStr(fptr,keyname)
```

Description

`[value,comment] = readKeyLongStr(fptr,keyname)` returns the specified long string value and comment.

This function corresponds to the `fits_read_key_longstr (ffgkls)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
idata = repmat(char(97:106),1,10);
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeKey(fptr,'mykey',idata);
odata1 = fits.readKey(fptr,'mykey');
odata2 = fits.readKeyLongStr(fptr,'mykey');
fits.closeFile(fptr);
```

See Also

`readKey`

matlab.io.fits.readKeyUnit

Physical units string from keyword

Syntax

```
units = readKeyUnit(fptr,keyname)
```

Description

`units = readKeyUnit(fptr,keyname)` returns the physical units string from an existing keyword. If no units are defined, `units` is returned as an empty string.

This function corresponds to the `fits_read_key_unit" (ffgunt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'long_img',[10 20]);
fits.writeKey(fptr,'VELOCITY',12.3,'orbital speed');
fits.writeKeyUnit(fptr,'VELOCITY','km/s');
units = fits.readKeyUnit(fptr,'VELOCITY');
fits.closeFile(fptr);
```

See Also

`readKey` | `writeKeyUnit`

matlab.io.fits.readRecord

Header record specified by number

Syntax

```
card = readRecord(fptr,keynum)
```

Description

`card = readRecord(fptr,keynum)` returns the entire 80-character header record identified by the numeric `keynum`. Trailing blanks are truncated.

This function corresponds to the `fits_read_record (ffgrec)` function in the CFITSIO library C API.

Examples

Read the second record in each HDU.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getHdrSpace(fptr);
for j = 1:n
    card = fits.readRecord(fptr,j);
    fprintf('record %d:  "%s"\n', j, card);
end
fits.closeFile(fptr);
```

See Also

`deleteRecord` | `readCard` | `readKey`

matlab.io.fits.writeComment

Write or append COMMENT keyword to CHU

Syntax

```
writeComment(fptr,comment)
```

Description

`writeComment(fptr,comment)` writes (appends) a COMMENT keyword to the CHU. The comment string will be continued over multiple keywords if it is longer than 70 characters.

This function corresponds to the `fits_write_comment (ffpcom)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeComment(fptr,'this is a comment');
fits.writeComment(fptr,'this is another comment');
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`writeDate` | `writeHistory`

matlab.io.fits.writeDate

Write DATE keyword to CHU

Syntax

```
writeDate(FPTR)
```

Description

`writeDate(FPTR)` writes the DATE keyword to the CHU.

This function corresponds to the `fits_write_date (ffpdat)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeDate(fptr);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`writeComment` | `writeHistory`

matlab.io.fits.writeKey

Update or add new keyword into current HDU

Syntax

```
writeKey(fptr,keyname,value,comment)
writeKey(fptr,keyname,value,comment,decimals)
```

Description

`writeKey(fptr,keyname,value,comment)` adds a new record in the current HDU, or updates it if it already exists. `comment` is optional.

`writeKey(fptr,keyname,value,comment,decimals)` adds a new floating point keyword in the current HDU, or updates it if it already exists. You must use this syntax to write a keyword with imaginary components. `decimals` is ignored otherwise.

If a character `value` exceeds 68 characters in length, the LONGWARN convention is automatically employed.

This function corresponds to the `fits_write_key (ffpky)` and `fits_update_key (ffuky)` family of functions in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeKey(fptr,'mykey1','a char value','with a comment');
fits.writeKey(fptr,'mykey2',int32(1));
fits.writeKey(fptr,'mykey3',5+7*j,'with another comment');
fits.writeKey(fptr,'mykey4',4/3,'with yet another comment',2);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`deleteKey` | `readKey` | `readRecord`

matlab.io.fits.writeKeyUnit

Write physical units string

Syntax

```
writeKeyUnit(fptr,keyname,unit)
```

Description

`writeKeyUnit(fptr,keyname,unit)` writes a physical units string into an existing keyword.

This function corresponds to the `fits_write_key_unit (ffpunt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myFitsFile.fits');
fits.createImg(fptr,'long_img',[10 20]);
fits.writeKey(fptr,'VELOCITY',12.3,'orbital speed');
fits.writeKeyUnit(fptr,'VELOCITY','km/s');
fits.closeFile(fptr);
```

See Also

`readKeyUnit`

matlab.io.fits.writeHistory

Write or append HISTORY keyword to CHU

Syntax

```
writeHistory(fptr,history)
```

Description

`writeHistory(fptr,history)` writes (appends) a HISTORY keyword to the CHU. The history string is continued over multiple keywords if it is longer than 70 characters.

This function corresponds to the `fits_write_history (ffphis)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeHistory(fptr,'this is a history keyword');
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`writeComment` | `writeDate`

matlab.io.fits.copyHDU

Copy current HDU from one file to another

Syntax

```
copyHDU(infptr,outfptr)
```

Description

`copyHDU(infptr,outfptr)` copies the current HDU from the FITS file associated with `infptr` and appends it to the end of the FITS file associated with `outfptr`.

This function corresponds to the `fits_copy_hdu (ffcopy)` function in the CFITSIO library C API.

Examples

Copy the first, third, and fifth HDUs from one file to another.

```
import matlab.io.*
infptr = fits.openFile('tst0012.fits');
outfptr = fits.createFile('myfile.fits');
fits.copyHDU(infptr,outfptr);
fits.movAbsHDU(infptr,3);
fits.copyHDU(infptr,outfptr);
fits.movAbsHDU(infptr,5);
fits.copyHDU(infptr,outfptr);
fits.closeFile(infptr);
fits.closeFile(outfptr);
fitsdisp('tst0012.fits','mode','min','index',[1 3 5]);
fitsdisp('myfile.fits','mode','min');
```

See Also

`deleteHDU`

matlab.io.fits.deleteHDU

Delete current HDU in FITS file

Syntax

```
HDU_TYPE = deleteHDU(fptr)
```

Description

`HDU_TYPE = deleteHDU(fptr)` deletes the current HDU in the FITS file. Any following HDUs will be shifted forward in the file, filling the gap created by the deleted HDU. In the case of deleting the primary array (the first HDU in the file) then the current primary array will be replaced by a null primary array containing the minimum set of required keywords and no data. If there are more HDUs in the file following the HDU being deleted, then the current HDU will be redefined to point to the following HDU. If there are no following HDUs then the current HDU will be redefined to point to the previous HDU. `HDU_TYPE` returns the type of the new current HDU.

This function corresponds to the `fits_delete_hdu (ffdhdu)` function in the CFITSIO library C API.

Examples

Delete the second HDU in a FITS file.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fitsdisp('myfile.fits','mode','min');
fptr = fits.openFile('myfile.fits','readwrite');
fits.movAbsHDU(fptr,2);
new_current_hdu = fits.deleteHDU(fptr);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','min');
```

See Also

copyHDU

matlab.io.fits.getHDUnum

Number of current HDU in FITS file

Syntax

```
N = getHDUnum(fptr)
```

Description

`N = getHDUnum(fptr)` returns the number of the current HDU in the FITS file. The primary array has HDU number 1.

This function corresponds to the `fits_get_hdu_num (ffghdn)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getHDUnum(fptr);
fits.closeFile(fptr);
```

See Also

`getHDUtype` | `getNumHDUs`

matlab.io.fits.getHDUtype

Type of current HDU

Syntax

```
htype = getHDUtype(fptr)
```

Description

`htype = getHDUtype(fptr)` returns the type of the current HDU in the FITS file. The possible values for `htype` are:

```
'IMAGE_HDU'
'ASCII_TBL'
'BINARY_TBL'
```

This function corresponds to the `fits_get_hdu_type (ffghdt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.getHDUtype(fptr);
end
fits.closeFile(fptr);
```

See Also

`getHDUnum`

matlab.io.fits.getNumHDUs

Total number of HDUs in FITS file

Syntax

```
N = getNumHDUs(fptr)
```

Description

`N = getNumHDUs(fptr)` returns the number of completely defined HDUs in a FITS file. If a new HDU has just been added to the FITS file, then that last HDU will only be counted if it has been closed, or if data has been written to the HDU. The current HDU remains unchanged by this routine.

This function corresponds to the `fits_get_num_hdus (ffthdu)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
fits.closeFile(fptr);
```

See Also

`getHDUnum`

matlab.io.fits.movAbsHDU

Move to absolute HDU number

Syntax

```
htype = fits.movAbsHDU(fptr,HDUNUM)
```

Description

`htype = fits.movAbsHDU(fptr,HDUNUM)` moves to a specified absolute HDU number (starting with 1 for the primary array) in the FITS file. The possible values for `htype` are:

```
'IMAGE_HDU'
'ASCII_TBL'
'BINARY_TBL'
```

This function corresponds to the `fits_move_abs_hdu` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    htype = fits.movAbsHDU(fptr,j);
    fprintf('HDU %d:  "%s"\n',j,htype);
end
fits.closeFile(fptr);
```

See Also

`getNumHDUs` | `movNamHDU` | `movRelHDU`

matlab.io.fits.movNamHDU

Move to first HDU having specific type and keyword values

Syntax

```
movNamHDU(fptr,hdutype,EXTNAME,EXTVER)
```

Description

`movNamHDU(fptr,hdutype,EXTNAME,EXTVER)` moves to the first HDU which has the specified extension type and `EXTNAME` and `EXTVER` keyword values (or `HDUNAME` and `HDUVER` keywords).

The `hdutype` parameter may have a value of:

```
'IMAGE_HDU'
'ASCII_TBL'
'BINARY_TBL'
'ANY_HDU'
```

If `hdutype` is `'ANY_HDU'`, only the `EXTNAME` and `EXTVER` values are used to locate the correct extension. If the input value of `EXTVER` is 0, then the `EXTVER` keyword is ignored and the first HDU with a matching `EXTNAME` (or `HDUNAME`) keyword will be found.

This function corresponds to the `fits_movnam_hdu (ffmnhd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movNamHDU(fptr,'IMAGE_HDU','quality',1);
fits.closeFile(fptr);
```

See Also

`movAbsHDU` | `movRelHDU`

matlab.io.fits.movRelHDU

Move relative number of HDUs from current HDU

Syntax

```
htype = moveRelHDU(fptr,nmove)
```

Description

`htype = moveRelHDU(fptr,nmove)` moves a relative number of HDUs forward or backward from the current HDU and returns the HDU type, `htype`, of the resulting HDU. The possible values for `htype` are:

```
'IMAGE_HDU'
'ASCII_TBL'
'BINARY_TBL'
```

This function corresponds to the `fits_movrel_hdu (ffmrhd)` function in the CFITSIO library C API.

Examples

Move through each HDU in succession, then move backwards twice by two HDUs.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    htype = fits.movAbsHDU(fptr,j);
    fprintf('HDU %d:  "%s"\n',j,htype);
end
htype = fits.movRelHDU(fptr,-2);
n = fits.getHDUnum(fptr);
fprintf('HDU %d:  "%s"\n',n,htype);
htype = fits.movRelHDU(fptr,-2);
```

```
n = fits.getHDUnum(fptr);
fprintf('HDU %d:  "%s"\n',n,htype);
fits.closeFile(fptr);
```

See Also

movAbsHDU | movNamHDU

matlab.io.fits.writeChecksum

Compute and write checksum for current HDU

Syntax

```
writeChecksum(fptr)
```

Description

`writeChecksum(fptr)` computes and writes the `DATASUM` and `CHECKSUM` keyword values for the current HDU into the current header. If the keywords already exist, their values are updated only if necessary (for example, if the file has been modified since the original keyword values were computed).

This function corresponds to the `fits_write_chksum (ffpcks)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'long_img',[10 20]);
fits.writeChecksum(fptr)
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`fitsdisp`

matlab.io.fits.imgCompress

Compress HDU from one file into another

Syntax

```
imgCompress(infptr,outfptr)
```

Description

`imgCompress(infptr,outfptr)` initializes the output HDU, copies all the keywords, and loops through the input image, compressing the data and writing the compressed data to the output HDU.

This function corresponds to the `fits_img_compress` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
infptr = fits.openFile('tst0012.fits');
outfptr = fits.createFile('myfile.fits');
fits.setCompressionType(outfptr,'rice');
fits.imgCompress(infptr,outfptr);
fits.closeFile(infptr);
fits.closeFile(outfptr);
```

See Also

`setCompressionType`

matlab.io.fits.isCompressedImg

Determine if current image is compressed

Syntax

```
TF = isCompressedImg(fptr)
```

Description

`TF = isCompressedImg(fptr)` returns true if the image in the current HDU is compressed.

This function corresponds to the `fits_is_compressed_image` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
bool = fits.isCompressedImg(fptr);
fits.closeFile(fptr);
```

See Also

`setCompressionType`

matlab.io.fits.setCompressionType

Set image compression type

Syntax

```
setCompressionType(fptr,comptype)
```

Description

`setCompressionType(fptr,comptype)` specifies the image compression algorithm that should be used when writing a FITS image.

Supported values for `comptype` include:

```
'GZIP'
'GZIP2'
'RICE'
'PLIO'
'HCOMPRESS'
'NOCOMPRESS'
```

This function corresponds to the `fits_set_compression_type` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'GZIP2');
fits.createImg(fptr,'long_img',[256 512]);
data = reshape(1:256*512,[256 512]);
data = int32(data);
fits.writeImg(fptr,data);
fits.closeFile(fptr);
```

```
fitsdisp('myfile.fits','mode','full');
```

See Also

createImg | setTileDim

matlab.io.fits.setHCompScale

Set scale parameter for HCOMPRESS algorithm

Syntax

```
setHCompScale(fptr,scale)
```

Description

`setHCompScale(fptr,scale)` sets the scale parameter to be used with the HCOMPRESS compression algorithm. Setting the scale parameter causes the algorithm to operate in lossy mode.

This function corresponds to the `fits_set_hcomp_scale` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
data = 50*ones(256,512,'double') + 10 * rand([256 512]);
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'HCOMPRESS_1');
fits.setHCompScale(fptr,2.5);
fits.createImg(fptr,'double_img',[256 512]);
fits.writeImg(fptr,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`setCompressionType` | `setHCompSmooth`

matlab.io.fits.setHCompSmooth

Set smoothing for images compressed with HCOMPRESS

Syntax

```
setHCompSmooth(fptr,smooth)
```

Description

`setHCompSmooth(fptr,smooth)` sets the smoothing to be used when compressing an image with the HCOMPRESS algorithm. Setting either the scale or smoothing parameter causes the algorithm to operate in lossy mode.

This function corresponds to the `fits_set_hcomp_smooth` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
data = int32(50*ones(256,512,'double') + 10 * rand([256 512]));
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'HCOMPRESS');
fits.setHCompSmooth(fptr,1);
fits.createImg(fptr,'long_img',[256 512]);
fits.writeImg(fptr,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`setCompressionType` | `setHCompScale`

matlab.io.fits.setTileDim

Set tile dimensions

Syntax

```
fits.setTileDim(fptr,tiledims)
```

Description

`fits.setTileDim(fptr,tiledims)` specifies the size of the image compression tiles to be used when creating a compressed image.

This function corresponds to the `fits_set_tile_dim` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'RICE_1');
fits.setTileDim(fptr,[64 128]);
fits.createImg(fptr,'byte_img',[256 512]);
data = ones(256,512,'uint8');
fits.writeImg(fptr,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`setCompressionType`

matlab.io.fits.createTbl

Create new ASCII or binary table extension

Syntax

```
fptr = createTbl(fptr,tbltype,nrows,ttype,tform,tunit,extname)
```

Description

`fptr = createTbl(fptr,tbltype,nrows,ttype,tform,tunit,extname)` creates a new ASCII or bintable table extension. `ttype` must be either `'binary'` or `'ascii'`. The `nrows` argument gives the initial number of rows to be created in the table and should normally be zero. `tunit` specifies the units for each column, but can be an empty cell array if no units are desired. `extname` specifies the extension name, but can be omitted.

`tform` specifies the format of the column. For binary tables, the values should be in the form of `'rt'`, where `'r'` is the repeat count and `'t'` is one of the following letters.

`'A'`	ASCII character
`'B'`	Byte or `uint8`
`'C'`	Complex (single precision)
`'D'`	Double precision
`'E'`	Single precision
`'I'`	`int16`
`'J'`	`int32`
`'K'`	`int64`
`'L'`	Logical
`'M'`	Complex (double precision)
`'X'`	Bit (`int8` zeros and ones)

A column can also be specified as having variable-width if the `tform` value has the form `'1Pt'` or `'1Qt'`, where `'t'` specifies the data type as above.

For ASCII tables, the `tform` values take the form:

Iw	`int16` column with width `'w'`
Aw	ASCII column with width `'w'`
Fww.dd	Fixed point
Eww.dd	Single precision with width `'ww'` and precision `'dd'`
Dww.dd	Double precision with width `'ww'` and precision `'dd'`

This function corresponds to the `fits_create_tbl (ffcrtb)` function in the CFITSIO library C API.

Examples

Create a binary table. The first column contains strings of nine characters each. The second column contains four-element sequences of bits. The third column contains three-element sequences of `uint8` values. The fourth column contains double-precision scalars.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2','Col3','Col4'};
tform = {'9A','4X','3B','1D'};
tunit = {'m/s','kg','kg/m^3','candela'};
fits.createTbl(fptr,'binary',10,ttype,tform,tunit,'my-table');
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

Create a two-column table where the first column has a single double-precision value, but the second column has a variable-length double-precision value.

```
import matlab.io.*
fptr = fits.createFile('myfile2.fits');
ttype = {'Col1','Col2'};
tform = {'1D','1PD'};
fits.createTbl(fptr,'binary',0,ttype,tform);
fits.closeFile(fptr);
fitsdisp('myfile2.fits');
```

See Also

`createImg` | `insertATbl` | `insertBTbl` | `readCol` | `writeCol`

matlab.io.fits.deleteCol

Delete column from table

Syntax

```
deleteCol(fptr,colnum)
```

Description

`deleteCol(fptr,colnum)` deletes the column from an ASCII or binary table.

This function corresponds to the `fits_delete_col (ffdcol)` function in the CFITSIO library C API.

Examples

Delete the second column in a binary table.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fprintf('Before:  '); fitsdisp('myfile.fits','index',2,'mode','min');
fptr = fits.openFile('myfile.fits','readwrite');
fits.movAbsHDU(fptr,2);
fits.deleteCol(fptr,2);
fits.closeFile(fptr);
fprintf('After :  '); fitsdisp('myfile.fits','index',2,'mode','min');
```

See Also

`deleteRows`

matlab.io.fits.deleteRows

Delete rows from table

Syntax

```
deleteRows(fptr,firstrow,nrows)
```

Description

`deleteRows(fptr,firstrow,nrows)` deletes rows from an ASCII or binary table.

This function corresponds to the `fits_delete_rows (ffdrow)` function in the CFITSIO library C API.

Examples

Delete the second, third, and fourth rows in a binary table (second HDU).

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fprintf('Before:  '); fitsdisp('myfile.fits','index',2,'mode','min');
fptr = fits.openFile('myfile.fits','readwrite');
fits.movAbsHDU(fptr,2);
fits.deleteRows(fptr,2,2);
fits.closeFile(fptr);
fprintf('After :  '); fitsdisp('myfile.fits','index',2,'mode','min');
```

See Also

`deleteCol` | `insertRows`

matlab.io.fits.insertRows

Insert rows into table

Syntax

```
insertRows(fptr,firstrow,nrows)
```

Description

`insertRows(fptr,firstrow,nrows)` inserts rows into an ASCII or binary table. `firstrow` is a one-based number.

This function corresponds to the `fits_insert_rows (ffirow)` function in the CFITSIO library C API.

Examples

Insert five rows into an empty table.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2'};
tform = {'3A','1D'};
tunit = {'m/s','candela'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
fits.insertRows(fptr,1,5);
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2);
```

See Also

`deleteRows` | `insertCol`

matlab.io.fits.getAColParms

ASCII table information

Syntax

```
[ttype,tbcol,tunit,tform,scale,zero,nulstr,tdisp] =
getAColParms(fptr,colnum)
```

Description

`[ttype,tbcol,tunit,tform,scale,zero,nulstr,tdisp] = getAColParms(fptr,colnum)` gets information about an existing ASCII table column.

This function corresponds to the `fits_get_acolparms (fffacl)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,5);
[ttype,tbcol,tunit,tform,scale,zero,nulstr,tdisp] = fits.getAColParms(fptr,2);
fits.closeFile(fptr);
```

See Also

`getBColParms`

matlab.io.fits.getBColParms

Binary table information

Syntax

```
[ttype,tunit,typechar,repeat,scale,zero,nulval,tdisp] =
getBColParms(fptr,colnum)
```

Description

`[ttype,tunit,typechar,repeat,scale,zero,nulval,tdisp] = getBColParms(fptr,colnum)` gets information about an existing binary table column.

This function corresponds to the `fits_get_bcolparms (ffgbcl)` function in the CFITSIO library C API.

Examples

Get information about the second column in a binary table.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[ttype,tunit,typechar,repeat,scale,zero,nulval,tdisp]= fits.getBColParms(fptr,2);
fits.closeFile(fptr);
```

See Also

`getAColParms`

matlab.io.fits.getColName

Table column name

Syntax

```
[colnum,colname] = getColNum(fptr,templt,casesen)
```

Description

`[colnum,colname] = getColNum(fptr,templt,casesen)` gets the table column numbers and names of the columns whose names match an input template name. If `casesen` is true, then the column name match is case-sensitive. `casesen` defaults to false.

The input column name template may be either the exact name of the column to be searched for, or it may contain wildcard characters (*, ?, or #), or it may contain the integer number of the desired column (with the first column = 1). The `'*'` wildcard character matches any sequence of characters (including zero characters) and the `'?'` character matches any single character. The # wildcard matches any consecutive string of decimal digits (0-9).

Examples

Return all the columns starting with the letter `'C'`.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[nums,names] = fits.getColName(fptr,'C*');
fits.closeFile(fptr);
```

See Also

`getAColParms` | `getBColParms`

matlab.io.fits.getColType

Scaled column data type, repeat value, width

Syntax

```
[dtype,repeat,width] = getColType(fptr,colnum)
```

Description

`[dtype,repeat,width] = getColType(fptr,colnum)` returns the data type, vector repeat value, and the width in bytes of a column in an ASCII or binary table.

This function corresponds to the `fits_get_coltypell (ffgtclll)` function in the CFITSIO library C API.

Examples

Get information about the 'FLUX' column in the second HDU.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[dtype,repeat,width] = fits.getColType(fptr,5);
fits.closeFile(fptr);
```

See Also

`getEqColType`

matlab.io.fits.getEqColType

Column data type, repeat value, width

Syntax

```
[dtype,repeat,width] = getColType(fptr,colnum)
```

Description

`[dtype,repeat,width] = getColType(fptr,colnum)` returns the scaled data type needed to store the scaled column data type, the vector repeat value, and the width in bytes of a column in an ASCII or binary table.

This function corresponds to the `fits_get_eqcoltypell (ffeqtyll)` function in the CFITSIO library C API.

Examples

Get information about the 'FLUX' column in the second HDU.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[dtype,repeat,width] = fits.getEqColType(fptr,5);
fits.closeFile(fptr);
```

See Also

`getColType`

matlab.io.fits.getNumCols

Number of columns in table

Syntax

```
ncols = getNumCols(fptr)
```

Description

`ncols = getNumCols(fptr)` gets the number of columns in the current FITS table. This function corresponds to the `fits_get_num_cols (ffgncl)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
ncols = fits.getNumCols(fptr);
nrows = fits.getNumRows(fptr);
fits.closeFile(fptr);
```

See Also

`getNumRows`

matlab.io.fits.getNumRows

Number of rows in table

Syntax

```
nrows = getNumRows(fptr)
```

Description

`nrows = getNumRows(fptr)` gets the number of rows in the current FITS table. This function corresponds to the `fits_get_num_rowsll` (ffgnrwll) function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
ncols = fits.getNumCols(fptr);
nrows = fits.getNumRows(fptr);
fits.closeFile(fptr);
```

See Also

`getNumCols`

matlab.io.fits.insertCol

Insert column into table

Syntax

```
insertCol(fptr,colnum,ttype,tform)
```

Description

`insertCol(fptr,colnum,ttype,tform)` inserts a column into an ASCII or binary table.

This function corresponds to the `fits_insert_col (fficol)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2'};
tform = {'3A','1D'};
tunit = {'m/s','candela'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
fits.insertCol(fptr,3,'Col3','3D');
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2);
```

See Also

`insertRows`

matlab.io.fits.insertATbl

Insert ASCII table after current HDU

Syntax

```
insertATbl(fptr,rowlen,nrows,ttype,tbcol,tform,tunit,extname)
```

Description

`insertATbl(fptr,rowlen,nrows,ttype,tbcol,tform,tunit,extname)` inserts a new ASCII table extension immediately following the current HDU. Any following extensions are shifted down to make room for the new extension. If there are no other following extensions, then the new table extension is simply appended to the end of the file. If the FITS file is currently empty then this routine creates a dummy primary array before appending the table to it. The new extension becomes the current HDU. If `rowlen` is 0, then CFITSIO calculates the default `rowlen` based on the `tbcol` and `ttype` values.

`tform` can take the following forms. In each case, `'w'` and `'ww'` represent the widths of the ASCII columns.

Iw	`int16` column
Aw	ASCII column
Fww.dd	Fixed point with `'dd'` digits after the decimal point
Eww.dd	Single precision with `'dd'` digits of precision
Dww.dd	Double precision with `'dd'` digits of precision

Binary tables are recommended instead of ASCII tables.

This function corresponds to the `fits_insert_atbl (ffitab)` function in the CFITSIO library C API.

Examples

Create an ASCII table between two images.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'uint8',[20 30]);
fits.createImg(fptr,'int16',[30 40]);
fits.movRelHDU(fptr,-1);
ttype = {'Name','Short','Fix','Double'};
tbcol = [1 17 28 43];
tform = {'A15','I10','F14.2','D12.4'};
tunit = {'','m**2','cm','km/s'};
fits.insertATbl(fptr,0,0,ttype,tbcol,tform,tunit,'my-table');
fits.writeCol(fptr,1,1,char('abracadabra','hocus-pocus'));
fits.writeCol(fptr,2,1,int16([0; 1]));
fits.writeCol(fptr,3,1,[12.4; 4/3]);
fits.writeCol(fptr,4,1,[12.4; 4e8/3]);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','min');
```

See Also

`createTbl` | `insertBTbl`

matlab.io.fits.insertBTbl

Insert binary table after current HDU

Syntax

```
insertBTbl(fptr,nrows,ttype,tform,tunit,extname,pcount)
```

Description

`insertBTbl(fptr,nrows,ttype,tform,tunit,extname,pcount)` inserts a new binary table extension immediately following the current HDU. Any following extensions are shifted down to make room for the new extension. If there are no other following extensions then the new table extension is simply appended to the end of the file. If the FITS file is currently empty then this routine creates a dummy primary array before appending the table to it. The new extension becomes the CHDU. If there are following extensions in the file and if the table contains variable-length array columns then `pcount` must specify the expected final size of the data heap. Otherwise, `pcount` must be zero.

This function corresponds to the `fits_insert_btbl (ffibin)` function in the CFITSIO library C API.

Examples

Create a table following the primary array. Then, insert a new table just before it.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2'};
tform = {'9A','1D'};
tunit = {'m/s','candela'};
fits.createTbl(fptr,'binary',10,ttype,tform,tunit,'my-table');
fits.movRelHDU(fptr,-1);
fits.insertBTbl(fptr,5,ttype,tform,tunit,'my-new-table',0);
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

See Also

`createTbl` | `insertATbl`

matlab.io.fits.readATblHdr

Read header information from current ASCII table

Syntax

```
[rowlen,nrows,ttype,tbcol,tform,tunit,extname] = readATblHdr(fptr)
```

Description

`[rowlen,nrows,ttype,tbcol,tform,tunit,extname] = readATblHdr(fptr)` reads header information for the current ASCII table.

This function corresponds to the `fits_read_atblhdrll (ffghtbll)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,5);
[rowlen,nrows,ttype,tbcol,tform,tunit,extname] = fits.readATblHdr(fptr);
fits.closeFile(fptr);
```

See Also

`readBTblHdr`

matlab.io.fits.readBTblHdr

Read header information from current binary table

Syntax

```
[nrows,ttype,tform,tunit,extname,pcount] = readBTblHdr(fptr)
```

Description

`[nrows,ttype,tform,tunit,extname,pcount] = readBTblHdr(fptr)` reads header information for the current binary table.

This function corresponds to the `fits_read_btblhdrll (ffghbnll)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[nrows,ttype,tform,tunit,extname,pcount] = fits.readBTblHdr(fptr);
fits.closeFile(fptr);
```

See Also

`readATblHdr`

matlab.io.fits.readCol

Read rows of ASCII or binary table column

Syntax

```
[coldata,nullval] = readCol(fptr,colnum)
[coldata,nullval] = readCol(fptr,colnum,firstrow,numrows)
```

Description

`[coldata,nullval] = readCol(fptr,colnum)` reads an entire column from an ASCII or binary table column. `nullval` is a logical array specifying if a particular element of `coldata` should be treated as undefined. It is the same size as `coldata`.

`[coldata,nullval] = readCol(fptr,colnum,firstrow,numrows)` reads a subsection of rows from an ASCII or binary table column.

The MATLAB data type returned by `readCol` corresponds to the data type returned by `getEqColType`.

This function corresponds to the `fits_read_col (ffgcv)` function in the CFITSIO library C API.

Examples

Read an entire column.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
colnum = fits.getColName(fptr,'flux');
fluxdata = fits.readCol(fptr,colnum);
fits.closeFile(fptr);
```

Read the first five rows in a column.

```
import matlab.io.*
```

```
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
colnum = fits.getColName(fptr,'flux');
fluxdata = fits.readCol(fptr,colnum,1,5);
fits.closeFile(fptr);
```

See Also

writeCol

matlab.io.fits.setTscale

Reset image scaling

Syntax

```
setTscale(fptr,colnum,tscale,tzero)
```

Description

`setTscale(fptr,colnum,tscale,tzero)` resets the scaling factors for a table column according to the equation:

```
output = (FITS array) * tscale + tzero
```

The inverse formula is used when writing data values to the FITS file.

This only affects the automatic scaling performed when the data elements are read. It does not change the `tscale` and `tzero` keyword values.

Examples

Turn off automatic scaling in a table column where the `tscale` and `tzero` keywords are present.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
scaled_data = fits.readCol(fptr,3);
fits.setTscale(fptr,3,1.0,0.0);
unscaled_data = fits.readCol(fptr,3);
fits.closeFile(fptr);
```

See Also

`readImg`

matlab.io.fits.writeCol

Write elements into ASCII or binary table column

Syntax

```
writeCol(fptr,colnum,firstrow,coldata)
```

Description

`writeCol(fptr,colnum,firstrow,coldata)` writes elements into an ASCII or binary table extension column.

When writing rows of data to a variable length field, `coldata` must be a cell array.

This function corresponds to the `fits_write_col" (ffpcl)` function in the CFITSIO library C API.

Examples

Write to a table with ASCII, `uint8`, double-precision, and variable-length double-precision columns.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2','Col3','Col4'};
tform = {'3A','3B','1D','1PD'};
tunit = {'m/s','kg/m^3','candela','parsec'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
fits.writeCol(fptr,1,1,['dog'; 'cat']);
fits.writeCol(fptr,2,1,[0 1 2; 3 4 5; 6 7 8; 9 10 11]);
fits.writeCol(fptr,3,1,[1; 2; 3; 4]);
fits.writeCol(fptr,4,1,{1;[1 2];[1 2 3];[1 2 3 4]});
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2,'mode','full');
```

Write to a table with logical, bit, double precision, and variable-length complex single-precision columns.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2','Col3','Col4'};
tform = {'2L','3X','1D','1PC'};
tunit = {'','kg/m^3','candela','parsec'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
fits.writeCol(fptr,1,1,[false false; true false]);
fits.writeCol(fptr,2,1,int8([0 1 1; 1 1 1; 1 1 1; 1 0 1]));
fits.writeCol(fptr,3,1,[1; 2; 3; 4]);
data = cell(4,1);
data{1} = single(1);
data{2} = single(1+2j);
data{3} = single([1j 2 3+j]);
data{4} = single([1 2+3j 3 4]);
fits.writeCol(fptr,4,1,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2,'mode','full');
```

See Also

createTbl | readCol

matlab.io.fits.getConstantValue

Numeric value of named constant

Syntax

```
N = getConstantValue(name)
```

Description

`N = getConstantValue(name)` returns the numeric value corresponding to the named CFITSIO constant.

Examples

```
import matlab.io.*
n = fits.getConstantValue('BYTE_IMG');
```

matlab.io.fits.getVersion

Revision number of the CFITSIO library

Syntax

```
V = getVersion()
```

Description

`V = getVersion()` returns the revision number of the CFITSIO library. This function corresponds to the `fits_get_version (ffvers)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
v = fits.getVersion();
```

matlab.io.fits.getOpenFiles

List of open FITS files

Syntax

```
fptrs = getOpenFiles()
```

Description

`fptrs = getOpenFiles()` returns a list of file pointers of all open FITS files.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
clear fptr;
fptr = fits.getOpenFiles();
fits.closeFile(fptr);
```

See Also

`closeFile` | `openFile`

matlab.io.hdf4.sd

Interact directly with HDF4 multifile scientific data set (SD) interface

Description

To use these MATLAB functions, you should be familiar with the HDF SD C API. In most cases, the syntax of the MATLAB function is similar to the syntax of the corresponding HDF library function. The functions are implemented as the package `matlab.io.hdf4.sd`. To use this package, prefix the function name with a package path, or use the import function to add the package to the current import list, prior to calling the function, for example,

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','read');
```

Access

matlab.io.hdf4.sd.close	Terminate access to SD interface
matlab.io.hdf4.sd.endAccess	Terminate access to data set
matlab.io.hdf4.sd.getFilename	Name of file
matlab.io.hdf4.sd.select	Identifier of data set with specified index
matlab.io.hdf4.sd.setExternalFile	Store data in external file
matlab.io.hdf4.sd.start	Open HDF file and initialize SD interface

Read/Write

matlab.io.hdf4.sd.create	Create new data set

matlab.io.hdf4.sd.readData	Read subsample of data
matlab.io.hdf4.sd.setFillMode	Set current fill mode of file
matlab.io.hdf4.sd.writeData	Write to data set

Inquiry

matlab.io.hdf4.sd.fileInfo	Number of data sets and global attributes in file
matlab.io.hdf4.sd.getCompInfo	Information about data set compression
matlab.io.hdf4.sd.getFillValue	Fill value for data set
matlab.io.hdf4.sd.getInfo	Information about data set
matlab.io.hdf4.sd.idToRef	Reference number corresponding to data set identifier
matlab.io.hdf4.sd.idType	Type of object
matlab.io.hdf4.sd.isCoordVar	Determine if data set is a coordinate variable
matlab.io.hdf4.sd.isRecord	Determine if data set is appendable
matlab.io.hdf4.sd.nameToIndex	Index value of named data set
matlab.io.hdf4.sd.nameToIndices	List of data sets with same name
matlab.io.hdf4.sd.refToIndex	Index of data set corresponding to reference number

Dimensions

matlab.io.hdf4.sd.dimInfo	Information about dimension
matlab.io.hdf4.sd.getDimID	Dimension identifier
matlab.io.hdf4.sd.getDimScale	Scale data for dimension
matlab.io.hdf4.sd.setDimName	Associate name with dimension
matlab.io.hdf4.sd.setDimScale	Set scale values for dimension

User-defined Attributes

matlab.io.hdf4.sd.attrInfo	Information about attribute
matlab.io.hdf4.sd.findAttr	Index of specified attribute
matlab.io.hdf4.sd.readAttr	Read attribute value
matlab.io.hdf4.sd.setAttr	Write attribute value

Predefined Attributes

matlab.io.hdf4.sd.getCal	Data set calibration information
matlab.io.hdf4.sd.getDataStrs	Predefined attribute strings for data set
matlab.io.hdf4.sd.getDimStrs	Predefined attribute strings for dimension

matlab.io.hdf4.sd.getFillValue	Fill value for data set
matlab.io.hdf4.sd.getRange	Maximum and minimum range values
matlab.io.hdf4.sd.setCal	Set data set calibration information
matlab.io.hdf4.sd.setDataStrs	Set predefined attributes for data set
matlab.io.hdf4.sd.setDimStrs	Set label, unit, and format attribute strings
matlab.io.hdf4.sd.setFillValue	Set fill value for data set
matlab.io.hdf4.sd.setRange	Set maximum and minimum range value for data set

Chunking/Tiling Operations

matlab.io.hdf4.sd.getChunkInfo	Chunk size for data set
matlab.io.hdf4.sd.readChunk	Read chunk from data set
matlab.io.hdf4.sd.setChunk	Set chunk size and compression method of data set
matlab.io.hdf4.sd.writeChunk	Write chunk to data set

Compression

matlab.io.hdf4.sd.setCompress	Set compression method of data set

matlab.io.hdf4.sd.setNBitDataSet	Specify nonstandard bit length for data set values

matlab.io.hdf4.sd.attrInfo

Package: matlab.io.hdf4.sd

Information about attribute

Syntax

```
[name,datatype,nelts] = attrInfo(objID,idx)
```

Description

`[name,datatype,nelts] = attrInfo(objID,idx)` returns the name, data type, and number of elements in the specified attribute. The attribute is specified by its zero-based index value. `objID` can be either an SD interface identifier, a data set identifier, or a dimension identifier.

This function corresponds to the `SDattrinfo` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.findAttr(sdID,'creation_date');
[name,datatype,nelts] = sd.attrInfo(sdID,idx);
data = sd.readAttr(sdID,idx);
sd.close(sdID);
```

See Also

`sd.findAttr`

matlab.io.hdf4.sd.close

Package: matlab.io.hdf4.sd

Terminate access to SD interface

Syntax

```
sd.close(sdID)
```

Description

`sd.close(sdID)` closes the file identified by `sdID`.

This function corresponds to the `SDend` function in the HDF C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.start`

matlab.io.hdf4.sd.create

Package: matlab.io.hdf4.sd

Create new data set

Syntax

```
sdsID = create(sdID,name,datatype,dims)
```

Description

`sdsID = create(sdID,name,datatype,dims)` creates a data set with the given name `name`, data type `datatype`, and dimension sizes `dims`.

To create a data set with an unlimited dimension, the last value in `dims` should be set to 0.

This function corresponds to the `SDcreate` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `dims` parameter is reversed with respect to the C library API.

Examples

Create a 3D data set with an unlimited dimension.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20 0]);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.endAccess`

matlab.io.hdf4.sd.dimInfo

Package: matlab.io.hdf4.sd

Information about dimension

Syntax

```
[name,dimlen,datatype,nattrs] = dimInfo(dimID)
```

Description

`[name,dimlen,datatype,nattrs] = dimInfo(dimID)` returns the name, length, data type, and number of attributes of the specified dimension.

This function corresponds to the `SDdiminfo` function in the HDF library C API.

Examples

Read a 2-by-3 portion of a data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'latitude');
sdsID = sd.select(sdID,idx);
dimID = sd.getDimID(sdsID,0);
[name,dimlen,datatype,nattrs] = sd.dimInfo(dimID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDimID`

matlab.io.hdf4.sd.endAccess

Package: matlab.io.hdf4.sd

Terminate access to data set

Syntax

```
sd.endAccess(sdsID)
```

Description

`sd.endAccess(sdsID)` terminates access to the data set identified by `sdsID`. Failing to call this function after all operations on the specified data set are complete may result in loss of data.

This function corresponds to the `SDendaccess` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.close` | `sd.select`

matlab.io.hdf4.sd.fileInfo

Package: matlab.io.hdf4.sd

Number of data sets and global attributes in file

Syntax

```
[ndatasets,ngatts] = fileInfo(sdID)
```

Description

`[ndatasets,ngatts] = fileInfo(sdID)` returns the number of data sets `ndatasets` and the number of global attributes `ngatts` in the file identified by `sdID`.

`ndatasets` includes the number of coordinate variable data sets.

This function corresponds to the `SDfileinfo` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
[ndatasets,ngatts] = sd.fileInfo(sdID);
sd.close(sdID);
```

See Also

`sd.getInfo`

matlab.io.hdf4.sd.findAttr

Package: matlab.io.hdf4.sd

Index of specified attribute

Syntax

```
idx = findAttr(objID,attrname)
```

Description

`idx = findAttr(objID,attrname)` returns the index of the attribute specified by `attrname`. The `objID` input can be either an SD interface identifier, a data set identifier, or a dimension identifier.

The function corresponds to the `SDfindattr` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.findAttr(sdID,'creation_date');
data = sd.readAttr(sdID,idx);
sd.close(sdID);
```

See Also

`sd.getDimID` | `sd.readAttr` | `sd.select` | `sd.start`

matlab.io.hdf4.sd.getCal

Package: matlab.io.hdf4.sd

Data set calibration information

Syntax

```
[cal,calErr,offset,offsetErr,datatype] = getCal(sdsID)
```

Description

`[cal,calErr,offset,offsetErr,datatype] = getCal(sdsID)` retrieves the calibration information associated with a data set.

This function corresponds to the `SDgetcal` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[cal,calErr,offset,offsetErr,dtype] = sd.getCal(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setCal`

matlab.io.hdf4.sd.getChunkInfo

Package: matlab.io.hdf4.sd

Chunk size for data set

Syntax

```
chunkDims = getChunkInfo(sdsID)
```

Description

`chunkDims = getChunkInfo(sdsID)` returns the chunk size for the data set specified by `sdsID`. If a data set is chunked, the dimensions of the chunks is returned in `chunkDims`. Otherwise `chunkDims` is `[]`.

This function corresponds to the `SDgetchunkinfo` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `chunkDims` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
cdims = sd.getChunkInfo(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getCompInfo` | `sd.setChunk`

matlab.io.hdf4.sd.getCompInfo

Package: matlab.io.hdf4.sd

Information about data set compression

Syntax

```
[comptype,compparms] = getCompType(sdsID)
```

Description

`[comptype,compparms] = getCompType(sdsID)` retrieves the compression type and compression information for a data set. `comptype` can be one of the following strings.

`'none'`	No compression
`'rle'`	Run-length encoding
`'nbit'`	NBIT compression
`'skphuff'`	Skipping Huffman compression
`'deflate'`	GZIP compression
`'szip'`	SZIP compression

If `comptype` is `'none'` or `'rle'`, then `compparms` is `[]`.

If `comptype` is `'nbit'`, then `compparms` is a 4-element array.

`compparm(1)`	sign_ext
`compparm(2)`	fill_one
`compparm(3)`	start_bit
`compparm(4)`	bit_len

If `comptype` is `'deflate'`, then `compparms` contains the deflation value, a number between 0 and 9.

If `comptype` is `'szip'`, them `compparms` is a 5-element array. Consult the HDF Reference Manual for details on SZIP compression.

This function corresponds to the `SDgetcompinfo` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[100 50]);
sd.setCompress(sdsID,'deflate',5);
[comptype,compparm] = sd.getCompInfo(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setCompress` | `sd.setNBitDataSet`

matlab.io.hdf4.sd.getDataStrs

Package: matlab.io.hdf4.sd

Predefined attribute strings for data set

Syntax

```
[label,unit,format,coordsys] = getDataStrs(sdsID)
[label,unit,format,coordsys] = getDataStrs(sdsID,maxlen)
```

Description

`[label,unit,format,coordsys] = getDataStrs(sdsID)` returns the label, unit, format, and coordsys attributes for the data set identified by `sdsID`.

`[label,unit,format,coordsys] = getDataStrs(sdsID,maxlen)` returns the label, unit, format, and coordsys attributes for the data set identified by `sdsID`. The `maxlen` input is the maximum length of any of the attribute strings. It defaults to 1000 if not specified.

This function corresponds to the `SDgetdatastrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[label,unit,fmt,coordsys] = sd.getDataStrs(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setDataStrs`

matlab.io.hdf4.sd.getDimID

Package: matlab.io.hdf4.sd

Dimension identifier

Syntax

```
dimID = getDimID(sdsID,dimnumber)
```

Description

`dimID = getDimID(sdsID,dimnumber)` returns the identifier of the dimension given its index.

Note: MATLAB uses Fortran-style indexing while the HDF library uses C-style indexing. The order of the dimension identifiers retrieved with `sd.getDimID` are reversed from what would be retrieved via the C API.

This function corresponds to the `SDgetdimid` function in the HDF library C API.

Examples

Read an entire data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dimID0 = sd.getDimID(sdsID,0);
dimID1 = sd.getDimID(sdsID,1);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setDimName`

matlab.io.hdf4.sd.getDimScale

Package: matlab.io.hdf4.sd

Scale data for dimension

Syntax

```
scale = getDimScale(dimID)
```

Description

`scale = getDimScale(dimID)` returns the scale values of the dimension identified by `dimID`.

This function corresponds to the `SDgetdimscale` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',20);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'x');
sd.setDimScale(dimID,0:5:95);
sd.endAccess(sdsID);
sd.close(sdID);
sdID = sd.start('myfile.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dimID = sd.getDimID(sdsID,0);
scale = sd.getDimScale(dimID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.dimInfo` | `sd.setDimScale`

matlab.io.hdf4.sd.getDimStrs

Package: matlab.io.hdf4.sd

Predefined attribute strings for dimension

Syntax

```
[label,unit,format] = getDimStrs(dimID)
```

Description

`[label,unit,format] = getDimStrs(dimID)` returns the label, unit, and format attributes for the dimension identified by `dimID`.

This function corresponds to the `SDgetdimstrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',20);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'x');
sd.setDimStrs(dimID,'xdim','none','%d');
sd.endAccess(sdsID);
sd.close(sdID);
sdID = sd.start('myfile.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dimID = sd.getDimID(sdsID,0);
[label,unit,fmt] = sd.getDimStrs(dimID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setDimStrs`

matlab.io.hdf4.sd.getFilename

Package: matlab.io.hdf4.sd

Name of file

Syntax

```
filename = getFilename(sdID)
```

Description

`filename = getFilename(sdID)` retrieves the name of a file previously opened with the `sd` package with identifier `sdID`.

This function corresponds to the `SDgetfilename` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
filename = sd.getFilename(sdID);
sd.close(sdID);
```

See Also

`sd.getInfo` | `sd.start`

matlab.io.hdf4.sd.getFillValue

Package: matlab.io.hdf4.sd

Fill value for data set

Syntax

```
fillvalue = getFillValue(sdsID)
```

Description

`fillvalue = getFillValue(sdsID)` returns the fill value for a data set.

This function corresponds to the `SDgetfillvalue` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
fillvalue = sd.getFillValue(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setFillValue`

matlab.io.hdf4.sd.getInfo

Package: matlab.io.hdf4.sd

Information about data set

Syntax

```
[name,dims,datatype,nattrs] = getInfo(sdsID)
```

Description

`[name,dims,datatype,nattrs] = getInfo(sdsID)` returns the name, extents, and number of attributes of the data set identified by `sdsID`.

This function corresponds to the `SDgetinfo` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the dims parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[name,dims,datatype,nattrs] = sd.getInfo(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.attrInfo` | `sd.dimInfo` | `sd.fileInfo`

matlab.io.hdf4.sd.getRange

Package: matlab.io.hdf4.sd

Maximum and minimum range values

Syntax

```
[maxval,minval] = getRange(sdsID)
```

Description

`[maxval,minval] = getRange(sdsID)` retrieves the "valid_range" two-element attribute value.

This function corresponds to the `SDgetrange` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[maxval,minval] = sd.getRange(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setRange`

matlab.io.hdf4.sd.idToRef

Package: matlab.io.hdf4.sd

Reference number corresponding to data set identifier

Syntax

```
ref = idToRef(sdsID)
```

Description

`ref = idToRef(sdsID)` returns the reference number corresponding to the data set.

This function corresponds to the `SDidtoref` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
ref = sd.idToRef(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.refToIndex`

matlab.io.hdf4.sd.idType

Package: matlab.io.hdf4.sd

Type of object

Syntax

```
objtype = idType(objID)
```

Description

`objtype = idType(objID)` returns the type of object that `objID` represents. Possible values for `objtype` are:

`'NOT_SDAPI_ID'`	The object is not an HDF SD identifier.
`'SD_ID'`	The object is an SD identifier (file handle) .
`'SDS_ID'`	The object is a data set identifier.
`'DIM_ID'`	The object is a dimension identifier.

This function corresponds to the `SDidtype` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
objType = sd.idType(sdID);
sd.close(sdID);
```

matlab.io.hdf4.sd.isCoordVar

Package: matlab.io.hdf4.sd

Determine if data set is a coordinate variable

Syntax

```
TF = isCoordVar(sdsID)
```

Description

`TF = isCoordVar(sdsID)` returns `true` if a data set is a coordinate variable and returns `false` otherwise.

This function corresponds to the `SDiscoordvar` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
ndataset = sd.fileInfo(sdID);
for idx = 0:ndataset-1
 sdsID = sd.select(sdID,idx);
 sdsName = sd.getInfo(sdsID);
 fprintf('%s (index %d) ', sdsName, idx);
 if ( sd.isCoordVar(sdsID) )
  fprintf('is a coordinate variable.\n');
 else
  fprintf('is not a coordinate variable.\n');
 end
 sd.endAccess(sdsID);
end
sd.close(sdID);
```

See Also

`sd.isRecord`

matlab.io.hdf4.sd.isRecord

Package: matlab.io.hdf4.sd

Determine if data set is appendable

Syntax

```
TF = isRecord(sdsID)
```

Description

`TF = isRecord(sdsID)` determines if the data set specified by `sdsID` is appendable, meaning that the slowest changing dimension is unlimited.

This function corresponds to the `SDisrecord` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
ndataset = sd.fileInfo(sdID);
for idx = 0:ndataset-1
    sdsID = sd.select(sdID,idx);
    sdsName = sd.getInfo(sdsID);
    if sd.isRecord(sdsID)
        fprintf('%s is a record variable.\n',sdsName);
    else
        fprintf('%s is not a record variable.\n',sdsName);
    end
    sd.endAccess(sdsID);
end
sd.close(sdID);
```

See Also

`sd.isCoordVar`

matlab.io.hdf4.sd.nameToIndex

Package: matlab.io.hdf4.sd

Index value of named data set

Syntax

```
idx = nameToIndex(sdID,sdsname)
```

Description

`idx = nameToIndex(sdID,sdsname)` returns the index of the data set with the name specified by `sdsname`. If there is more than one data set with the same name, the routine returns the index of the first one.

This function corresponds to the `SDnametoindex` function in the HDF C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf','read');
idx = sd.nameToIndex(sdID,'temperature');
sd.close(sdID);
```

See Also

`sd.select`

matlab.io.hdf4.sd.nameToIndices

Package: matlab.io.hdf4.sd

List of data sets with same name

Syntax

```
varstruct = nameToIndices(sdID,sdsname)
```

Description

`varstruct = nameToIndices(sdID,sdsname)` returns a structure array for all data sets with the same name. Each element of `varstruct` has two fields.

`'index'`	Index of data set
`'type'`	Type of data set, either `'SDSVAR'`, `'COORDVAR'`, or `'UNKNOWN'`

This function corresponds to the `SDnametoindices` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
varlist = sd.nameToIndices(sdID,'latitude');
sd.close(sdID);
```

See Also

`sd.isCoordVar` | `sd.setDimScale`

matlab.io.hdf4.sd.readAttr

Package: matlab.io.hdf4.sd

Read attribute value

Syntax

```
data = readAttr(objID,idx)
```

Description

`data = readAttr(objID,idx)` reads the value of the attribute specified by index `idx`. The `objID` input can be an SD interface identifier, a data set identifier, or a dimension identifier. `idx` is a zero-based index.

This function corresponds to the `SDreadattr` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.findAttr(sdID,'creation_date');
data = sd.readAttr(sdID,idx);
sd.close(sdID);
```

See Also

`sd.findAttr` | `sd.setAttr`

matlab.io.hdf4.sd.readChunk

Package: matlab.io.hdf4.sd

Read chunk from data set

Syntax

```
datachunk = readChunk(sdsID,origin)
```

Description

`datachunk = readChunk(sdsID,origin)` reads an entire chunk of data from the data set identified by `sdsID`. The `origin` input specifies the location of the chunk in zero-based chunking coordinates, not in data set coordinates.

This function corresponds to the `SDreadchunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the origin parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dataChunk = sd.readChunk(sdsID,[0 1]);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.writeChunk` | `sd.writeData`

matlab.io.hdf4.sd.readData

Package: matlab.io.hdf4.sd

Read subsample of data

Syntax

```
data = readData(sdsID)
data = readData(sdsID,start,count)
data = readData(sdsID,start,count,stride)
```

Description

`data = readData(sdsID)` reads all of the data for the data set identified by `sdsID`.

`data = readData(sdsID,start,count)` reads a contiguous hyperslab of data from the data set identified by `sdsID`. The `start` input specifies the starting position from where the hyperslab is read. `count` specifies the number of values to read along each data set dimension.

`data = readData(sdsID,start,count,stride)` reads a strided hyperslab of data from the data set identified by `sdsID`.

`start`, `count`, and `stride` use zero-based indexing.

This function corresponds to the `SDreaddata` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `start`, `count`, and `stride` parameters are reversed with respect to the C library API.

Examples

Read an entire data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
```

```
sdsID = sd.select(sdID,idx);
data = sd.readData(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

Read a 2-by-3 portion of a data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
data = sd.readData(sdsID,[0 0],[2 3]);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

sd.writeData

matlab.io.hdf4.sd.refToIndex

Package: matlab.io.hdf4.sd

Index of data set corresponding to reference number

Syntax

```
idx = refToIndex(sdID,ref)
```

Description

`idx = refToIndex(sdID,ref)` returns the index of the data set identified by its reference number `ref`. The `idx` output can then be passed to `sd.select`, to obtain a data set identifier.

This function corresponds to the `SDreftoindex` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf','read');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
ref = sd.idToRef(sdsID);
idx2 = sd.refToIndex(sdID,ref);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.idToRef` | `sd.select`

matlab.io.hdf4.sd.select

Package: matlab.io.hdf4.sd

Identifier of data set with specified index

Syntax

```
sdsID = select(sdID,IDX)
```

Description

`sdsID = select(sdID,IDX)` returns the identifier of the data set specified by its index.

This function corresponds to the `SDselect` function in the HDF C library.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf','read');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.endAccess` | `sd.nametoIndex`

matlab.io.hdf4.sd.setAttr

Package: matlab.io.hdf4.sd

Write attribute value

Syntax

```
setAttr(objID,name,value)
```

Description

`setAttr(objID,name,value)` attaches an attribute to the object specified by `objID`. If `objID` is the SD interface identifier, then a global attribute is created. If a data identifier is specified, then the attribute is attached to the data set. If a dimension identifier is specified, then the attribute is attached to the dimension.

This function corresponds to the `SDsetattr` function in the HDF library C API.

Examples

Attach attributes to a file, a data set, and to a dimension.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sd.setAttr(sdID,'creation_date',datestr(now));
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setAttr(sdsID,'long_name','Temperature in sunlight.');
dimID0 = sd.getDimID(sdsID,0);
sd.setAttr(dimID0,'long_name','latitude');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.findAttr` | `sd.readAttr`

matlab.io.hdf4.sd.setCal

Package: matlab.io.hdf4.sd

Set data set calibration information

Syntax

```
setCal(sdsID,cal,calErr,offset,offsetErr,datatype)
```

Description

`setCal(sdsID,cal,calErr,offset,offsetErr,datatype)` sets the calibration information for a data set.

This function corresponds to the `SDsetcal` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setDataStrs(sdsID,'Temperature','degrees_kelvin','%.3f','spherical');
sd.setCal(sdsID,1,0,273,0,'double');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getCal`

matlab.io.hdf4.sd.setChunk

Package: matlab.io.hdf4.sd

Set chunk size and compression method of data set

Syntax

```
setChunk(sdsID,chunkSize,comptype,compparm)
```

Description

`setChunk(sdsID,chunkSize,comptype,compparm)` makes the data set specified by `sdsID` a chunked data set with chunk size given by `chunkSize` and compression specified by `comptype` and `compparm`. The `comptype` input can be one of the following strings.

`'none'`	No compression
`'skphuff'`	Skipping Huffman compression
`'deflate'`	GZIP compression
`'rle'`	Run-length encoding

- If `comptype` is `'none'` or `'rle'`, then `compparm` need not be specified.
- If `comptype` is `'skphuff'`, then `compparm` is the skipping size.
- If `comptype` is `'deflate'`, then `compparm` is the deflate level, which must be between 0 and 9.

This function corresponds to the `SDsetchunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `chunkSize` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
```

```
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[200 100]);
sd.setChunk(sdsID,[20 10],'skphuff',16);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

sd.readChunk | sd.writeChunk

matlab.io.hdf4.sd.setCompress

Package: matlab.io.hdf4.sd

Set compression method of data set

Syntax

```
setCompress(sdsID,comptype,compparm)
```

Description

`setCompress(sdsID,comptype,compparm)` sets the compression scheme for the specified data set. The compression must be done before writing the data set. `comptype` can be one of the following strings.

`'none'`	No compression
`'skphuff'`	Skipping Huffman compression
`'deflate'`	GZIP compression
`'rle'`	Run-length encoding

- If `comptype` is `'none'` or `'rle'`, then `compparm` need not be specified.
- If `comptype` is `'skphuff'`, then `compparm` is the skipping size.
- If `comptype` is `'deflate'`, then `compparm` is the deflate level, which must be between 0 and 9.

This function corresponds to the `SDsetcompress` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[200 100]);
sd.setCompress(sdsID,'deflate',5);
data = rand(200,100);
```

```
sd.writeData(sdsID,[0 0],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

sd.setChunk

matlab.io.hdf4.sd.setDataStrs

Package: matlab.io.hdf4.sd

Set predefined attributes for data set

Syntax

```
setDataStrs(sdsID,label,unit,format,coordsys)
```

Description

`setDataStrs(sdsID,label,unit,format,coordsys)` sets the predefined attributes `'long_name'`, `'units'`, `'format'`, and `'coordsys'` for a data set.

This function corresponds to the `SDsetdatastrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setDataStrs(sdsID,'degrees_celsius','degrees_east','','geo');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDataStrs` | `sd.setDimStrs`

matlab.io.hdf4.sd.setDimName

Package: matlab.io.hdf4.sd

Associate name with dimension

Syntax

```
setDimName(dimID,dimname)
```

Description

`setDimName(dimID,dimname)` sets the name of the dimension identified by `dimID` to `dimname`.

This function corresponds to the `SDsetdimname` function in the HDF library C API.

Examples

Create a 2D data set with dimensions `'lat'` and `'lon'`.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'lat');
dimID = sd.getDimID(sdsID,1);
sd.setDimName(dimID,'lon');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.dimInfo`

matlab.io.hdf4.sd.setDimScale

Package: matlab.io.hdf4.sd

Set scale values for dimension

Syntax

```
setDimScale(dimID,scaledata)
```

Description

`setDimScale(dimID,scaledata)` sets the scale values for a dimension.

This function corresponds to the `SDsetdimscale` function in the HDF library C API.

Examples

Create a 2D data set with dimensions `'lat'` and `'lon'`.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'lat');
sd.setDimScale(dimID,0:10:90);
dimID = sd.getDimID(sdsID,1);
sd.setDimName(dimID,'lon');
sd.setDimScale(dimID, -180:18:179);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDimScale`

matlab.io.hdf4.sd.setDimStrs

Package: matlab.io.hdf4.sd

Set label, unit, and format attribute strings

Syntax

```
setDimStrs(dimID,label,unit,format)
```

Description

`setDimStrs(dimID,label,unit,format)` sets the label, unit, and format attributes for the dimension identified by `dimID`.

This function corresponds to the `SDsetdimstrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'lat');
dimID = sd.getDimID(sdsID,1);
sd.setDimName(dimID,'lon');
sd.setDimStrs(dimID,'Degrees of Longitude','degrees_east','%.2f');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDimStrs`

matlab.io.hdf4.sd.setExternalFile

Package: matlab.io.hdf4.sd

Store data in external file

Syntax

```
setExternalFile(sdsID,extfile,offset)
```

Description

`setExternalFile(sdsID,extfile,offset)` moves data values (not metadata) into the external data file `extfile` starting at the byte offset, `offset`.

Data can only be moved once for any given data set. The external file should be kept with the main file.

This function corresponds to the `SDsetexternalfile` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setExternalFile(sdsID,'myExternalFile.dat',0);
sd.writeData(sdsID,[0 0],rand(10,20));
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.create` | `sd.writeData`

matlab.io.hdf4.sd.setFillMode

Package: matlab.io.hdf4.sd

Set current fill mode of file

Syntax

```
prevmode = setFillMode(sdID,fillmode)
```

Description

`prevmode = setFillMode(sdID,fillmode)` returns the previous fill mode of a file and resets it to `fillmode`. This setting applies to all data sets contained in the file identified by `sdID`.

Possible values of `fillmode` are `'fill'`, and `'nofill'`. `'fill'` is the default mode and indicates that fill values will be written when the data set is created. `'nofill'` indicates that the fill values will not be written.

When a fixed-size data set is created, the first call to `sd.writeData` will fill the entire data set with the default or user-defined fill value if `fillmode` is `'fill'`. In data sets with an unlimited dimension, if a new write operation takes place along the unlimited dimension beyond the last location of the previous write operation, the array locations between these written areas will be initialized to the user-defined fill value, or the default fill value if a user-defined fill value has not been specified.

If it is certain that all data set values will be written before any read operation takes place, there is no need to write the fill values. Calling `sd.setFillMode` with `'nofill'` can improve performance in this case.

This function corresponds to the `SDsetfillmode` function in the HDF library C API.

Examples

Write two partial records. Write the first in `'nofill'` mode, and the second with `'fill'` mode.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sd.setFillMode(sdID,'nofill');
sdsID = sd.create(sdID,'temperature','double',[10 10 0]);
sd.writeData(sdsID,[0 0 0], rand(5,5));
sd.setFillMode(sdID,'fill');
sd.setFillValue(sdsID,-999);
sd.writeData(sdsID,[0 0 1], rand(5,5));
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

sd.getFillValue | sd.setFillValue

matlab.io.hdf4.sd.setFillValue

Package: matlab.io.hdf4.sd

Set fill value for data set

Syntax

```
setFillValue(sdsID,fillValue)
```

Description

`setFillValue(sdsID,fillValue)` sets the fill value for a data set. The fill value must have the same data type as the data set.

This function corresponds to the `SDsetfillvalue` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setFillValue(sdsID,-999);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getFillValue`

matlab.io.hdf4.sd.setNBitDataSet

Package: matlab.io.hdf4.sd

Specify nonstandard bit length for data set values

Syntax

```
setNBitDataSet(sdsID,startBit,bitlen,ext,fillone)
```

Description

`setNBitDataSet(sdsID,startBit,bitlen,ext,fillone)` specifies that the integer data set identified by `sdsID` contains data of a non-standard length defined by `startBit` and `bitlen`.

Any length between 1 and 32 bits can be specified. After `setNBitDataset` has been called for the data set array, any read or write operation will involve conversion between the new data length of the data set array and the data length of the read or write buffer.

Bit lengths of all data types are counted from the right of the bit field starting with 0. In a bit field containing the values 01111011, bits 2 and 7 are set to 0 and all the other bits are set to 1. The least significant bit is bit 0.

The `startBit` parameter specifies the left-most position of the variable-length bit field to be written. For example, in the bit field described in the preceding paragraph a `startBit` parameter set to 4 would correspond to the fourth bit value of 1 from the right.

The parameter `bitlen` specifies the number of bits of the variable-length bit field to be written. This number includes the starting bit and the count proceeds toward the right end of the bit field - toward the lower-bit numbers. For example, starting at bit 5 and writing 4 bits of the bit field described in the preceding paragraph would result in the bit field 1110 being written to the data set. This would correspond to a `startBit` value of 5 and a `bitlen` value of 4.

The parameter `ext` specifies whether to use the left-most bit of the variable-length bit field to sign-extend to the left-most bit of the data set data. For example, if 9-bit signed integer data is extracted from bits 17-25 and the bit in position 25 is 1, then when the

data is read back from disk, bits 26-31 will be set to 1. Otherwise bit 25 will be 0 and bits 26-31 will be set to 0. The `ext` parameter can be set to `true` (or 1) or `false` (or 0); specify `true` to sign-extend.

The parameter `fillone` specifies whether to fill the "background" bits with the value 1 or 0. This parameter is also set to either `true` (or 1) or `false` (or 0).

The "background" bits of a non-standard length data set are the bits that fall outside of the non-standard length bit field stored on disk. For example, if five bits of an unsigned 16-bit integer data set located in bits 5 to 9 are written to disk with the parameter `fillone` set to `true` (or 1), then when the data is reread into memory bits 0 to 4 and 10 to 15 would be set to 1. If the same 5-bit data was written with a `fillone` value of `false` (or 0), then bits 0 to 4 and 10 to 15 would be set to 0.

The operation on `fillone` is performed before the operation on `ext`. For example, using the `ext` example above, bits 0 to 16 and 26 to 31 will first be set to the background bit value, and then bits 26 to 31 will be set to 1 or 0 based on the value of the 25th bit.

This function corresponds to the `SDsetnbitdataset` in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','int32',[10 20]);
sd.setNBitDataSet(sdsID,6,4,0,0);
data = int32([1:200]);
data = reshape(data,10,20);
sd.writeData(sdsID,[0 0],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setCompress`

matlab.io.hdf4.sd.setRange

Package: matlab.io.hdf4.sd

Set maximum and minimum range value for data set

Syntax

```
setRange(sdsID,maxval,minval)
```

Description

`setRange(sdsID,maxval,minval)` sets the maximum and minimum range values of the data set identified by `sdsID`. These values form the "valid_range" attribute for `sdsID`.

The actual maximum and minimum values of the data set are not computed. The "valid_range" attribute is for informational purposes only.

This function corresponds to the `SDsetrange` function in the HDF library C interface.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setDataStrs(sdsID,'Temperature','degrees_celsius','%.2f','');
sd.setRange(sdsID,1000,-273.15);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getRange`

matlab.io.hdf4.sd.start

Package: matlab.io.hdf4.sd

Open HDF file and initialize SD interface

Syntax

```
sdID = start(filename)
sdID = start(filename,access)
```

Description

`sdID = start(filename)` opens the file `filename` in read-only mode. This routine must be called for each file before any other `sd` calls can be made on that file.

`sdID = start(filename,access)` opens the file `filename` with the access mode specified by `access`. This routine must be called before any other SD interface operations can be made on that file. `access` can be one of the following strings:

- `'read'`
- `'write'`
- `'create'`

`access` defaults to `'read'` if not supplied.

This function corresponds to the `SDstart` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
sd.close(sdID);
```

See Also

`sd.close`

matlab.io.hdf4.sd.writeChunk

Package: matlab.io.hdf4.sd

Write chunk to data set

Syntax

```
writeChunk(sdsID,origin,dataChunk)
```

Description

`writeChunk(sdsID,origin,dataChunk)` writes an entire chunk of data to the data set identified by `sdsID`. The `origin` input specifies the location of the chunk in chunking coordinates, not in data set coordinates.

This function corresponds to the `SDwritechunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the origin parameter is reversed with respect to the C library API.

Examples

Write to a 2D chunked and compressed data set. The chunked layout constitutes a 10-by-5 grid.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[100 50]);
sd.setChunk(sdsID,[10 10],'deflate',5);
for j = 0:9
    for k = 0:4
        origin = [j k];
        data = (1:100) + k*1000 + j*10000;
        data = reshape(data,10,10);
        sd.writeChunk(sdsID,origin,data);
    end
end
```

```
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.readChunk` | `sd.writeData`

matlab.io.hdf4.sd.writeData

Package: matlab.io.hdf4.sd

Write to data set

Syntax

```
writeData(sdsID,data)
writeData(sdsID,start,data)
writeData(sdsID,start,stride,data)
```

Description

`writeData(sdsID,data)` writes all the data to the data set identified by `sdsID`.

`writeData(sdsID,start,data)` writes a contiguous hyperslab to the data set. `start` specifies the zero-based starting index. The number of values along each dimension is inferred from the size of `data`.

`writeData(sdsID,start,stride,data)` writes a strided hyperslab of data to a grid datafield. The number of elements to write along each dimension is inferred either from the size of `data` or from the data set itself.

`start` and `stride` use zero-based indexing.

This function corresponds to the `SDreadchunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `start` and `stride` parameters are reversed with respect to the C library API.

Examples

Write to a 2D data set.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
```

```
data = rand(10,20);
sd.writeData(sdsID,[0 0],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

Write to a 2D unlimited data set.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 0]);
data = rand(10,20);
sd.writeData(sdsID,[0 0],data);
data = rand(10,30);
sd.writeData(sdsID,[0 20],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.readData`

matlab.io.hdfeos.gd

Low-level access to HDF-EOS grid data

Description

To use these MATLAB functions, you must be familiar with the HDF-EOS library C interface. In most cases, the syntax of the MATLAB function is similar to the syntax of the corresponding HDF-EOS library function. The functions are implemented as the package `matlab.io.hdfeos.gd`. To use this package, prefix the function name with a package path, or use the import function to add the package to the current import list, prior to calling the function, for example,

```
import matlab.io.hdfeos.*
gfid = gd.open(filename,'read');
```

Access

matlab.io.hdfeos.gd.attach	Attach to existing grid
matlab.io.hdfeos.gd.close	Close HDF-EOS grid file
matlab.io.hdfeos.gd.detach	Detach from existing grid
matlab.io.hdfeos.gd.open	Open grid file

Definition

matlab.io.hdfeos.gd.create	Create new grid structure
matlab.io.hdfeos.gd.defComp	Set grid field compression
matlab.io.hdfeos.gd.defDim	Define new dimension within grid

matlab.io.hdfeos.gd.defField	Define new data field within grid
matlab.io.hdfeos.gd.defOrigin	Define origin of pixels in grid
matlab.io.hdfeos.gd.defPixReg	Define pixel registration within grid
matlab.io.hdfeos.gd.defProj	Define grid projection
matlab.io.hdfeos.gd.writeBlkSomOffset	Write Block SOM offset

Basic I/O

matlab.io.hdfeos.gd.getFillValue	Fill value for specified field
matlab.io.hdfeos.gd.readAttr	Read grid attribute
matlab.io.hdfeos.gd.readField	Read data from grid field
matlab.io.hdfeos.gd.setFillValue	Set fill value for specified field
matlab.io.hdfeos.gd.writeAttr	Write grid attribute
matlab.io.hdfeos.gd.writeField	Write data to grid field

Inquiry

matlab.io.hdfeos.gd.compInfo	Compression information for field
matlab.io.hdfeos.gd.dimInfo	Length of dimension

matlab.io.hdfeos.gd.fieldInfo	Information about data field
matlab.io.hdfeos.gd.gridInfo	Position and size of grid
matlab.io.hdfeos.gd.inqAttrs	Names of grid attributes
matlab.io.hdfeos.gd.inqDims	Information about dimensions defined in grid
matlab.io.hdfeos.gd.inqFields	Information about data fields defined in grid
matlab.io.hdfeos.gd.inqGrid	Names of grids in file
matlab.io.hdfeos.gd.nEntries	Number of specified objects
matlab.io.hdfeos.gd.originInfo	Origin code
matlab.io.hdfeos.gd.pixRegInfo	Pixel registration code
matlab.io.hdfeos.gd.projInfo	GCTP projection information about grid
matlab.io.hdfeos.gd.readBlkSomOffset	Read Block SOM offset

Subsetting

matlab.io.hdfeos.gd.defBoxRegion	Define region of interest by latitude and longitude
matlab.io.hdfeos.gd.defVrtRegion	Define vertical subset region
matlab.io.hdfeos.gd.extractRegion	Read region of interest from field

matlab.io.hdfeos.gd.getPixels	Pixel rows and columns for latitude/ longitude pairs
matlab.io.hdfeos.gd.getPixValues	Read data values for specified pixels
matlab.io.hdfeos.gd.regionInfo	Information about subsetted region

Tiling

matlab.io.hdfeos.gd.defTile	Define tiling parameters
matlab.io.hdfeos.gd.readTile	Read single tile of data from field
matlab.io.hdfeos.gd.setTileComp	Set tiling and compression for field with fill value
matlab.io.hdfeos.gd.tileInfo	Tile size of grid field
matlab.io.hdfeos.gd.writeTile	Write tile to field

Utility

matlab.io.hdfeos.gd.ij2ll	Convert row and column space to latitude and longitude
matlab.io.hdfeos.gd.ll2ij	Convert latitude and longitude to row and column space
matlab.io.hdfeos.gd.sphereCodeToName	Name corresponding to GCTP sphere code

matlab.io.hdfeos.gd.sphereNameToCode	Numeric GCTP code corresponding to sphere name

matlab.io.hdfeos.gd.attach

Package: matlab.io.hdfeos.gd

Attach to existing grid

Syntax

```
gridID = attach(gfID,gridName)
```

Description

`gridID = attach(gfID,gridName)` attaches to the grid dataset identified by `gridName` in the file identified by `gfID`. The `gridID` output is the identifier for the grid dataset.

This function corresponds to the `GDattach` function in the HDF-EOS library C API.

Examples

Attach to the grid named `'PolarGrid'` in the file `'grid.hdf'`.

```
import matlab.io.hdfeos.*
gfID = gd.open('grid.hdf');
gridID = gd.attach(gfID,'PolarGrid');
gd.detach(gridID);
gd.close(gfID);
```

See Also

`gd.detach` | `gd.inqGrid` | `gd.readField`

matlab.io.hdfeos.gd.close

Package: matlab.io.hdfeos.gd

Close HDF-EOS grid file

Syntax

```
close(gfID)
```

Description

`close(gfID)` closes an HDF-EOS grid file identified by `gfID`.

This function corresponds to the `GDclose` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfID = gd.open('grid.hdf');
gd.close(gfID);
```

See Also

`gd.create` | `gd.open`

matlab.io.hdfeos.gd.compInfo

Package: matlab.io.hdfeos.gd

Compression information for field

Syntax

```
[compCode,parms] = compInfo(gridID,fieldname)
```

Description

`[compCode,parms] = compInfo(gridID,fieldname)` returns the compression code and compression parameters for a given field. Refer to `gd.defComp` for a description of various compression schemes and parameters.

This function corresponds to the `GDcompinfo` function in the HDF-EOS library C API.

Examples

Get compression information for the `ice_temp` field.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[compCode,compParms] = gd.compInfo(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defComp`

matlab.io.hdfeos.gd.create

Package: matlab.io.hdfeos.gd

Create new grid structure

Syntax

```
gridID = create(gfID,gridName,xdim,ydim,upLeft,lowRight)
```

Description

`gridID = create(gfID,gridName,xdim,ydim,upLeft,lowRight)` creates a new grid structure where `gfID` is the grid file identifier. `gridName` is the name of the new grid. `xdim` and `ydim` define the size of the grid. `upLeft` is a two-element vector containing the location of the upper left pixel, and `lowRight` is a two-element vector containing the location of the lower right pixel.

Note: `upLeft` and `lowRight` are in units of meters for all GCTP projections other than the geographic and bcea projections, which should have units of packed degrees.

Note: For certain projections, `upLeft` and `lowRight` can be given as `[]`.

- Polar Stereographic projection of an entire hemisphere.
- Goode Homolosine projection of the entire globe.
- Lambert Azimuthal entire polar or equatorial projection.

Note: MATLAB uses Fortran-style ordering, but the HDF-EOS library uses C-style ordering.

This function corresponds to the `GDcreate` function in the HDF-EOS library C API.

Examples

Create a polar stereographic grid of the northern hemisphere.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
gd.detach(gridID);
gd.close(gfid);
```

Create a UTM grid bounded by 54 E to 60 E longitude and 20 N to 30 N latitude. Divide the grid into 120 bins along the x-axis and 200 bins along the y-axis.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
uplft = [210584.50041 3322395.95445];
lowrgt = [813931.10959 2214162.53278];
gridID = gd.create(gfid,'UTMGrid',120,200,uplft,lowrgt);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defProj` | `gd.detach` | `gd.gridInfo`

matlab.io.hdfeos.gd.defBoxRegion

Package: matlab.io.hdfeos.gd

Define region of interest by latitude and longitude

Syntax

```
regionID = defBoxRegion(gridID,cornerLat,cornerLon)
```

Description

`regionID = defBoxRegion(gridID,cornerLat,cornerLon)` defines a latitude-longitude box region as a subset region for a grid. `regionID` can be used to read all the entries of a data field within the region.

This function corresponds to the `GDdefboxregion` function in the HDF-EOS library C API.

Examples

Define a region of interest between 20 and 50 degrees latitude and between -90 and -60 degrees longitude.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
regionID = gd.defBoxRegion(gridID,cornerlat,cornerlon);
data = gd.extractRegion(gridID,regionID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.extractRegion`

matlab.io.hdfeos.gd.defComp

Package: matlab.io.hdfeos.gd

Set grid field compression

Syntax

```
defComp(gridID,compscheme,compparm)
```

Description

`defComp(gridID,compscheme,compparm)` sets the HDF field compression for subsequent field definitions. The compression scheme does not apply to one-dimensional fields. `compscheme` can be one of the following strings.

`'rle'`	Run-length encoding
`'skphuff'`	Skipping Huffman
`'deflate'`	Gzip deflate
`'none'`	No compression

When the compression scheme is `'deflate'`, `compparm` is the deflate compression level, an integer between 0 and 9. `compparm` can be omitted for the other compression schemes.

If a field is defined with compression, it must be written with a single call to `gd.writeField`. If this is not possible, you should consider using tiling.

This function corresponds to the `GDdefcomp` function in the HDF-EOS library C API.

Examples

Create a grid with a polar stereographic `Pressure` field using run-length encoding, and then an `Opacity` field with deflate compression.

```
import matlab.io.hdfeos.*
```

```
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
dims = { 'XDim', 'YDim' };
gd.defComp(gridID,'rle');
gd.defField(gridID,'Pressure',dims,'float');
gd.defComp(gridID,'deflate',5);
gd.defField(gridID,'Opacity',dims,'float');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.defField | gd.defTile

matlab.io.hdfeos.gd.defDim

Package: matlab.io.hdfeos.gd

Define new dimension within grid

Syntax

```
defDim(gridID,dimname,dimlen)
```

Description

`defDim(gridID,dimname,dimlen)` defines a new dimension named `dimname` with length `dimlen` in the grid structure identified by `gridID`.

To specify an unlimited dimension, you can use either 0 or `'unlimited'` for `dimlen`.

This function corresponds to the `GDdefdim` function in the HDF-EOS library C API.

Examples

Define a dimension `'Band'` with length of 15 and an unlimited dimension `'Time'`.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
gd.defDim(gridID,'Band',15);
gd.defDim(gridID,'Time',0);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defField` | `gd.dimInfo`

matlab.io.hdfeos.gd.defField

Package: matlab.io.hdfeos.gd

Define new data field within grid

Syntax

```
defField(gridID,fieldname,dimlist,dtype)
defField(gridID,fieldname,dimlist,dtype,mergeCode)
```

Description

`defField(gridID,fieldname,dimlist,dtype)` defines data fields for a grid specified by `gridID`. The `fieldname` input is the name of the new field. `dimlist` is a cell array of geolocation dimensions and should be listed in FORTRAN-style order, that is, the fastest varying dimension should be listed first. `dimlist` can also be a string if there is only one dimension. `dtype` is the data type of the field.

`defField(gridID,fieldname,dimlist,dtype,mergeCode)` defines a data field with a specific merge code. `mergeCode` can be either `'nomerge'` or `'automerge'`. The `mergeCode` input defaults to `'nomerge'` if not provided.

This function corresponds to the `GDdeffield` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

Define a single precision grid field `'Temperature'` with dimensions `'XDim'` and `'YDim'`. Then define a single precision field `'Spectra'` with dimensions `'XDim'`, `'YDim'`, and `'Bands'`.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
xdim = 120; ydim = 200;
gridID = gd.create(gfid,'geo',xdim,ydim,[],[]);
```

```
gd.defProj(gridID,'geo',[],[],[]);
dimlist = {'XDim','YDim'};
gd.defField(gridID,'Temperature',dimlist,'single');
gd.defDim(gridID,'Bands',3);
dimlist = {'XDim','YDim','Bands'};
gd.defField(gridID,'Spectra',dimlist,'uint8');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.create | gd.defDim

matlab.io.hdfeos.gd.defOrigin

Package: matlab.io.hdfeos.gd

Define origin of pixels in grid

Syntax

```
defOrigin(gridID,originCode)
```

Description

`defOrigin(gridID,originCode)` defines the origin of pixels in a grid. `gridID` is the identifier of the grid, and `originCode` can be one of the following four strings.

`'ul'`	Upper-left
`'ur'`	Upper-right
`'ll'`	Lower-left
`'lr'`	Lower-right

You can select any corner of the grid pixel as the origin. If this routine is not invoked, the grid defaults to using the upper-left corner for the origin.

This function corresponds to the `GDdeforigin` function in the HDF-EOS library C API.

Examples

Create a polar stereographic grid with the origin of the grid pixel in the lower right corner.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
```

```
gd.defOrigin(gridID,'lr');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.defPixReg | gd.originInfo

matlab.io.hdfeos.gd.defPixReg

Package: matlab.io.hdfeos.gd

Define pixel registration within grid

Syntax

```
defPixReg(gridID,pixRegCode)
```

Description

`defPixReg(gridID,pixRegCode)` defines whether the pixel center or pixel corner is used when requesting the location (longitude and latitude) of a given pixel. `pixRegCode` can be one of the following strings.

`'center'`	Center of pixel cell
`'corner'`	Corner of pixel cell

If this routine is not invoked, the pixel registration is `'center'`.

This function corresponds to the `GDdefpixreg` function in the HDF-EOS library.

Examples

Define a grid with pixel registration in the center.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
gd.defPixReg(gridID,'corner');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defOrigin` | `gd.pixRegInfo`

matlab.io.hdfeos.gd.defProj

Package: matlab.io.hdfeos.gd

Define grid projection

Syntax

```
defProj(gridID,projCode,zoneCode,sphereCode,projParm)
```

Description

`defProj(gridID,projCode,zoneCode,sphereCode,projParm)` defines a GCTP projection on the grid specified by `gridID`. The `projCode` argument can be one of the following strings.

`'geo'`	Geographic
`'utm'`	Universal Transverse Mercator
`'albers'`	Albers Canonical Equal Area
`'lamcc'`	Lambert Conformal Conic
`'ps'`	Polar Stereographic
`'polyc'`	Polyconic
`'tm'`	Transverse Mercator
`'lamaz'`	Lambert Azimuthal Equal Area
`'snsoid'`	Sinusoidal
`'hom'`	Hotine Oblique Mercator
`'som'`	Space Oblique Mercator
`'good'`	Interrupted Goode Homolosine
`'cea'`	Cylindrical Equal Area
`'bcea'`	Behrmann Cylindrical Equal Area
`'isinus'`	Integerized Sinusoidal

If `projCode` is `'geo'`, then `zoneCode`, `sphereCode`, and `projParm` should be specified as `[]`. Any other values for these parameters are ignored.

`zoneCode` is the Universal Transverse Mercator zone code. It should be specified as -1 for other projections.

`sphereCode` is the name of the GCTP spheroid or the corresponding numeric code.

`projParm` is a vector of up to 13 elements containing projection-specific parameters. For more details about `projCode`, `zoneCode`, `sphereCode`, and `projParm`, see Chapter 6 of HDF-EOS Library Users Guide for the ECS Project, Volume 1: Overview and Examples.

This function corresponds to the `GDdefproj` function in the HDF library C API.

Examples

Create a UTM grid bounded by 54 E to 60 E longitude and 20 N to 30 N latitude (zone 40). Divide the grid into 120 bins along the x-axis and 200 bins along the y-axis.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
uplft = [210584.50041 3322395.95445];
lowrgt = [813931.10959 2214162.53278];
gridID = gd.create(gfid,'UTMGrid',120,200,uplft,lowrgt);
gd.defProj(gridID,'utm',40,'Clarke 1866',[]);
gd.detach(gridID);
gd.close(gfid);
```

Add a polar stereographic projection of the northern hemisphere with true scale at 90 N, 0 longitude below the pole using the WGS 84 spheroid.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.create` | `gd.projInfo` | `gd.sphereCodeToName`

matlab.io.hdfeos.gd.defTile

Package: matlab.io.hdfeos.gd

Define tiling parameters

Syntax

```
defTile(gridID,tileDims)
```

Description

`defTile(gridID,tileDims)` defines tiling dimensions for subsequent field definitions. If `tileDims` is `[]`, then subsequently defined fields will have no tiling.

This function corresponds to the `GDdeftile` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileDims` parameter is reversed with respect to the C library API.

Examples

Define a field with tiling, then a subsequent field with no tiling.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'GeoGrid',120,200,[],[]);
gd.defDim(gridID,'Bands',3);
gd.defProj(gridID,'geo',[],[],[]);
gd.defTile(gridID,[30 50 1]);
dimlist = {'XDim','YDim','Bands'};
gd.defField(gridID,'Spectra',dimlist,'float');
gd.defTile(gridID,[]);
dimlist = {'XDim','YDim'};
gd.defField(gridID,'Temperature',dimlist,'int32');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defField` | `gd.tileInfo`

matlab.io.hdfeos.gd.defVrtRegion

Package: matlab.io.hdfeos.gd

Define vertical subset region

Syntax

```
out_RID = defVrtRegion(gridID,regionID,vobj,vRange)
```

Description

`out_RID = defVrtRegion(gridID,regionID,vobj,vRange)` defines a vertical subset region and can be used on either a monotonic field or contiguous elements of a dimension.

`regionID` should be `'noprevsub'` if no prior subsetting has occurred. Otherwise it should be a value as returned from a previous subsetting routine.

`vobj` is the name of either the dimension or field to subset. If `vobj` is a dimension, it should be prefixed with `'DIM:'`.

`vRange` is the minimum and maximum range for the vertical subset.

This function corresponds to the `GDdefvrtregion` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
range = [333 667];
regionID = gd.defVrtRegion(gridID,'noprevsub','Height',range);
data = gd.extractRegion(gridID,regionID,'pressure');
gd.detach(gridID);
gd.close(gfid);
```

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
range = [3 5];
regionID = gd.defVrtRegion(gridID,'noprevsub','DIM:Height',range);
data = gd.extractRegion(gridID,regionID,'pressure');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.extractRegion

matlab.io.hdfeos.gd.detach

Package: matlab.io.hdfeos.gd

Detach from existing grid

Syntax

```
detach(gridID)
```

Description

`detach(gridID)` detaches from the grid identified by `gridID`.

This function corresponds to the `GDdetach` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfID = gd.open('grid.hdf');
gridID = gd.attach(gfID,'PolarGrid');
gd.detach(gridID);
gd.close(gfID);
```

See Also

`gd.attach`

matlab.io.hdfeos.gd.dimInfo

Package: matlab.io.hdfeos.gd

Length of dimension

Syntax

```
dimlen = diminfo(gridID,dimname)
```

Description

`dimlen = diminfo(gridID,dimname)` retrieves the length of the specified user-defined dimension.

Please note that the two extents used to create the grid are not considered user-defined dimensions. To retrieve the length of `XDim` and `YDim`, use `gd.gridInfo`. This function corresponds to the `GDdiminfo` function in the HDF-EOS library C API.

Examples

Inquire about a `'Bands'` dimension.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
dimlen = gd.dimInfo(gridID,'Height');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defDim` | `gd.gridInfo`

matlab.io.hdfeos.gd.extractRegion

Package: matlab.io.hdfeos.gd

Read region of interest from field

Syntax

```
data = extractRegion(gridID,regionID,fieldname)
```

Description

`data = extractRegion(gridID,regionID,fieldname)` extract data from a subsetted region.

This routine corresponds to the `GDextractregion` function in the HDF-EOS library C API.

Examples

Define and extract a region of interest between 20 and 50 degrees latitude and between -90 and -60 degrees longitude.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
regionID = gd.defBoxRegion(gridID,cornerlat,cornerlon);
data = gd.extractRegion(gridID,regionID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defBoxRegion` | `gd.defVrtRegion`

matlab.io.hdfeos.gd.fieldInfo

Package: matlab.io.hdfeos.gd

Information about data field

Syntax

```
[dims,ntype,dimlist] = fieldInfo(gridID,fieldname)
```

Description

`[dims,ntype,dimlist] = fieldInfo(gridID,fieldname)` returns information about a specific geolocation or data field in the grid. `dims` is a vector containing the dimension sizes of the field. `ntype` is a string containing the HDF number type of the field. `dimlist` is a cell array of strings containing the list of dimension names.

This function corresponds to the `GDfieldinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
[dims,ntype,dimlist] = gd.fieldInfo(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defField`

matlab.io.hdfeos.gd.getFillValue

Package: matlab.io.hdfeos.gd

Fill value for specified field

Syntax

```
fillvalue = getFillValue(gridID,fieldname)
```

Description

`fillvalue = getFillValue(gridID,fieldname)` retrieves the fill value for the specified field.

This function corresponds to the `GDgetfillvalue` function in the HDF-EOS library C API.

Examples

Return the fill value for the `'ice_temp'` field in the `'PolarGrid'` grid.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
fillvalue = gd.getFillValue(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.setFillValue`

matlab.io.hdfeos.gd.getPixels

Package: matlab.io.hdfeos.gd

Pixel rows and columns for latitude/longitude pairs

Syntax

```
[row,col] = getPixels(gridID,lat,lon)
```

Description

`[row,col] = getPixels(gridID,lat,lon)` converts latitude/longitude pairs into zero-based pixel row and column coordinates. The origin is the upper left-hand corner of the grid pixel. If the latitude/longitude pairs are outside the grid, then `row` and `col` are -1.

This function corresponds to the `GDgetpixels` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
[row,col] = gd.getPixels(gridID,cornerlat,cornerlon);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.getPixValues`

matlab.io.hdfeos.gd.getPixValues

Package: matlab.io.hdfeos.gd

Read data values for specified pixels

Syntax

```
data = getPixValues(gridID,rows,cols,fieldname)
```

Description

`data = getPixValues(gridID,rows,cols,fieldname)` reads data values for the pixels specified by the zero-based `rows` and `cols` coordinates. All entries along the non-geographic dimensions, i.e. NOT `XDim` and `YDim`, are returned.

This function corresponds to the `GDgetpixvalues` function in the HDF-EOS library C API.

Examples

Read the grid field's corner values.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
rows = [0 99 99  0];
cols = [0  0 99 99];
data = gd.getPixValues(gridID,rows,cols,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defBoxRegion` | `gd.extractRegion` | `gd.getPixels` | `gd.readField`

matlab.io.hdfeos.gd.gridInfo

Package: matlab.io.hdfeos.gd

Position and size of grid

Syntax

```
[xDim,yDim,upLeft,lowRight] = gridInfo(gridID)
```

Description

`[xDim,yDim,upLeft,lowRight] = gridInfo(gridID)` returns the size of a grid as well as the upper left and lower right corners of the grid.

Note: `upLeft` and `lowRight` are in units of meters for all GCTP projections other than the geographic and bcea projections, which will have units of packed degrees.

This function corresponds to the `GDgridinfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[xdimsize,ydimsize,upleft,lowright] = gd.gridInfo(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.create`

matlab.io.hdfeos.gd.ij2ll

Package: matlab.io.hdfeos.gd

Convert row and column space to latitude and longitude

Syntax

```
[lat,lon] = ij2ll(gridID,row,col)
```

Description

`[lat,lon] = ij2ll(gridID,row,col)` converts a grid's row and column coordinates to latitude and longitude in decimal degrees.

`row` and `col` are zero-based and defined such that `col` increases monotonically with the `XDim` dimension and `row` increases monotonically with the `YDim` dimension in the HD-EOS library.

This routine corresponds to the `GDij2ll` function in the HDF-EOS C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[xdim,ydim] = gd.gridInfo(gridID);
r = 0:(xdim-1);
c = 0:(ydim-1);
[Col,Row] = meshgrid(c,r);
[lat,lon] = gd.ij2ll(gridID,Row,Col);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.ll2ij` | `gd.readField`

matlab.io.hdfeos.gd.inqAttrs

Package: matlab.io.hdfeos.gd

Names of grid attributes

Syntax

```
attrList = inqAttrs(gridID)
```

Description

`attrList = inqAttrs(gridID)` returns the list of grid attribute names. `attrList` is a cell array.

This function corresponds to the `GDinqattrs` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
attrList = gd.inqAttrs(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.readAttr` | `gd.writeAttr`

matlab.io.hdfeos.gd.inqDims

Package: matlab.io.hdfeos.gd

Information about dimensions defined in grid

Syntax

```
[dimnames,dimlens] = inqDims(gridID)
```

Description

`[dimnames,dimlens] = inqDims(gridID)` returns the names of the dimensions `dimnames` in a cell array and their respective lengths `dimlens`. This does not include the grid extent dimensions `XDim` and `YDim`.

This function corresponds to the `GDinqdims` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimnames` and `dimlens` parameters are reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
[dims,dimlens] = gd.inqDims(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defDim`

matlab.io.hdfeos.gd.inqFields

Package: matlab.io.hdfeos.gd

Information about data fields defined in grid

Syntax

```
[fldList,fldRank,fldType] = inqFields(gridID)
```

Description

`[fldList,fldRank,fldType] = inqFields(gridID)` returns the list of fields `fldList` as a cell array. `fldRank` contains the rank of each data field. `fldType` is a cell array containing the data type of each data field.

This function corresponds to the `GDinqfields` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[fldlist,fldrank,fldtype] = gd.inqFields(gridID);
gd.detach(gridID);
gd.close(gfid);
for j = 1:numel(fldrank)
    fprintf('%s: Rank %d, datatype %s\n', fldlist{j},fldrank(j),fldtype{j});
end
```

See Also

`gd.defField`

matlab.io.hdfeos.gd.inqGrid

Package: matlab.io.hdfeos.gd

Names of grids in file

Syntax

```
grids = inqGrid(filename)
```

Description

`grids = inqGrid(filename)` returns the names of all grids in the given file. `grids` is a cell array.

This function corresponds to the `GDinqgrid` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
grids = gd.inqGrid('grid.hdf');
```

See Also

`gd.create` | `sw.inqSwath`

matlab.io.hdfeos.gd.ll2ij

Package: matlab.io.hdfeos.gd

Convert latitude and longitude to row and column space

Syntax

```
[row,col] = ll2ij(gridID,lat,lon)
```

Description

`[row,col] = ll2ij(gridID,lat,lon)` converts latitude and longitude coordinates to a pre-defined grid's row and column coordinates.

`row` and `col` are zero-based and defined such that `col` increases monotonically with the `XDim` dimension and `row` increases monotonically with the `YDim` dimension in the HD-EOS library.

This routine corresponds to the `GDll2ij` function in the HDF-EOS C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
lat = [46 46 42 42];
lon = [-71 -67 -67 -71];
[row,col] = gd.ll2ij(gridID,lat,lon);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.ij2ll`

matlab.io.hdfeos.gd.nEntries

Package: matlab.io.hdfeos.gd

Number of specified objects

Syntax

```
nentries = nEntries(gridID,entType)
```

Description

`nentries = nEntries(gridID,entType)` returns the number of specified objects in a grid. `entType` can be either `'dims'` or `'fields'`.

This function corresponds to the `GDnentries` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
fid = gd.open('grid.hdf');
gridID = gd.attach(fid,'PolarGrid');
ndims = gd.nEntries(gridID,'dims');
nflds = gd.nEntries(gridID,'fields');
gd.detach(gridID);
gd.close(fid);
fprintf('The number of dimensions is %d.\n', ndims);
fprintf('The number of fields is %d.\n', nflds);
```

See Also

`gd.inqGrid`

matlab.io.hdfeos.gd.open

Package: matlab.io.hdfeos.gd

Open grid file

Syntax

```
gfid = open(filename,access)
```

Description

`gfid = open(filename,access)` opens or creates an HDF-EOS grid file identified by `filename` and returns a file ID. `access` can be one of the following string values:

`'read'`	Read-only
`'rdwr'`	Read-write
`'create'`	Creates a file, deleting it if it already exists

If `access` is not provided, it defaults to `'read'`.

This function corresponds to the `GDopen` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gd.close(gfid);
```

See Also

`gd.attach` | `gd.close`

matlab.io.hdfeos.gd.originInfo

Package: matlab.io.hdfeos.gd

Origin code

Syntax

```
originCode = originInfo(gridID)
```

Description

`originCode = originInfo(gridID)` retrieves the origin code for the grid specified by `gridID`. The `originCode` output is one of the following four string values.

`'ul'`	Upper-left
`'ur'`	Upper-right
`'ll'`	Lower-left
`'lr'`	Lower-right

This function corresponds to the `GDorigininfo` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
origin = gd.originInfo(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defOrigin`

matlab.io.hdfeos.gd.pixRegInfo

Package: matlab.io.hdfeos.gd

Pixel registration code

Syntax

```
pixRegCode = pixRegInfo(gridID)
```

Description

`pixRegCode = pixRegInfo(gridID)` retrieve the pixel registration code for the grid identified by `gridID`. The `pixRegCode` output can be one of the following strings.

`'center'`	Center of pixel cell
`'corner'`	Corner of pixel cell

This function corresponds to the `GDpixreginfo` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
code = gd.pixRegInfo(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defPixReg`

matlab.io.hdfeos.gd.projInfo

Package: matlab.io.hdfeos.gd

GCTP projection information about grid

Syntax

```
[projCode,zoneCode,sphereName,projParm] = projInfo(gridID)
```

Description

`[projCode,zoneCode,sphereName,projParm] = projInfo(gridID)` returns the GCTP projection code, zone code, spheroid, and projection parameters for the grid identified by `gridID`.

`zoneCode` is -1 if `projCode` is anything other than `'UTM'`.

This function corresponds to the `GDprojinfo` function in the HDF-EOS library C API.

For details about the GCTP projection code, zone code, spheroid code, and projection parameters, please consult the HDF-EOS User's Guide.

Examples

```
import matlab.io.hdfeos.*
fid = gd.open('grid.hdf');
gridID = gd.attach(fid,'PolarGrid');
[projCode,zoneCode,sphereCode,projParm] = gd.projInfo(gridID);
gd.detach(gridID);
gd.close(fid);
```

See Also

`gd.defProj` | `gd.sphereCodeToName` | `gd.sphereNameToCode`

matlab.io.hdfeos.gd.readAttr

Package: matlab.io.hdfeos.gd

Read grid attribute

Syntax

```
data = readAttr(gridID,attrname)
```

Description

`data = readAttr(gridID,attrname)` reads a grid attribute.

This function corresponds to the `GDreadattr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
data = gd.readAttr(gridID,'creation_date');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.writeAttr`

matlab.io.hdfeos.gd.readBlkSomOffset

Package: matlab.io.hdfeos.gd

Read Block SOM offset

Syntax

```
offset = readBlkSomOffset(GID)
```

Description

`offset = readBlkSomOffset(GID)` reads the block SOM offset values, in pixels, from a standard SOM (Space Oblique Mercator) projection. `offset` is a vector of offset values for SOM projection data. This routine can only be used with grids that use the SOM projection.

This function corresponds to the `GDblkSOMoffset` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
lowright = [30521379.68485 1152027.64253];
upleft = [-11119487.42844 8673539.24806];
gridID = gd.create(gfid,'SOM',120,60,upleft,lowright);
projparm(1) = 6378137;
projparm(2) = 0.006694348;
projparm(4) = 98096360;  % 98.161 in DDDMMMSSS
projparm(5) = 87069061;  % 87.112 in DDDMMMSSS
projparm(9) = 0.068585416*1440;
projparm(10) = 0.0;
projparm(12) = 6;
gd.defProj(gridID,'som',[],[],projparm);
gd.writeBlkSomOffset(gridID,[5 10 12 8 2]);
gd.detach(gridID);
```

```
gd.close(gfid);
gfid = gd.open('myfile.hdf');
gridID = gd.attach(gfid,'SOM');
blk = gd.readBlkSomOffset(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.writeBlkSomOffset

matlab.io.hdfeos.gd.readField

Package: matlab.io.hdfeos.gd

Read data from grid field

Syntax

```
data = readField(gridID,fieldname)
data = readField(gridID,fieldname,start,count)
data = readField(gridID,fieldname,start,count,stride)
[data,lat,lon] = readField( ___ )
```

Description

`data = readField(gridID,fieldname)` reads the entire grid field identified by `fieldname` in the grid identified by `gridID`.

`data = readField(gridID,fieldname,start,count)` reads a contiguous hyperslab of data from the field. `start` specifies the zero-based starting index of the hyperslab. `count` specifies the number of values to read along each dimension.

`data = readField(gridID,fieldname,start,count,stride)` reads a strided hyperslab of data from the field. `stride` specifies the inter-element spacing along each dimension.

`[data,lat,lon] = readField( ___ )` reads the data and the associated geo-coordinates from the grid field. This syntax is only allowed when the leading two dimensions of the grid are `'XDim'` and `'YDim'`.

This function corresponds to the `GDreadfield` function in the HDF-EOS library C API.

Examples

Read the data, latitude, and longitude for the `'ice_temp'` field.

```
import matlab.io.hdfeos.*
```

```
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[data,lat,lon] = gd.readField(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

Read only the first 4x4 hyperslab of data, latitude, and longitude for the `'ice_temp'` field.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[data2,lat2,lon2] = gd.readField(gridID,'ice_temp',[0 0], [4 4]);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.writeField`

matlab.io.hdfeos.gd.readTile

Package: matlab.io.hdfeos.gd

Read single tile of data from field

Syntax

```
data = readTile(gridID,fieldname,tileCoords)
```

Description

`data = readTile(gridID,fieldname,tileCoords)` reads a single of data from a field. If the data is to be read tile by tile, this routine is more efficient than `gd.readField`. In all other cases, use `gd.readField`. The `tileCoords` argument has the form `[rownum colnum]` and is defined in terms of the tile coordinates, not the data elements.

This function corresponds to the `GDreadtile` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileCoords` parameter is reversed with respect to the C library API.

Examples

Define a field with a 2-by-3 tiling scheme.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
for h = 0:9
    data = gd.readTile(gridID,'pressure',[0 0 h]);
end
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.tileInfo` | `gd.writeTile`

matlab.io.hdfeos.gd.regionInfo

Package: matlab.io.hdfeos.gd

Information about subsetted region

Syntax

```
[dims,upLeft,lowRight] = regionInfo(gridID,regionID,fieldname)
```

Description

`[dims,upLeft,lowRight] = regionInfo(gridID,regionID,fieldname)` returns the dimensions and corner points for the specified field of a subsetted region identified by `regionID` in the grid identified by `gridID`.

This function corresponds to the `GDregioninfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
regionID = gd.defBoxRegion(gridID,cornerlat,cornerlon);
[dims,upleft,lowright] = gd.regionInfo(gridID,regionID,'ice_temp');
data = gd.extractRegion(gridID,regionID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defBoxRegion` | `gd.defVrtRegion`

matlab.io.hdfeos.gd.setFillValue

Package: matlab.io.hdfeos.gd

Set fill value for specified field

Syntax

```
setFillValue(gridID,fieldname,fillvalue)
```

Description

`setFillValue(gridID,fieldname,fillvalue)` sets the fill value for the specified field. The fill value should have the same data type as the field.

This function corresponds to the `GDsetfillvalue` function in the HDF-EOS library C API.

Examples

Create a new double-precision field with a fill value of -1.

```
import matlab.io.hdfeos.*
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','grid.hdf');
copyfile(srcFile,'myfile.hdf');
fileattrib('myfile.hdf','+w');
gfid = gd.open('myfile.hdf','rdwr');
gridID = gd.attach(gfid,'PolarGrid');
gd.defComp(gridID,'none');
gd.defField(gridID,'newfield',{'XDim','YDim'},'double');
gd.setFillValue(gridID,'newfield',-1);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.getFillValue`

matlab.io.hdfeos.gd.setTileComp

Package: matlab.io.hdfeos.gd

Set tiling and compression for field with fill value

Syntax

```
setTileComp(gridID,fieldname,tilesize,compCode,compParm)
```

Description

`setTileComp(gridID,fieldname,tilesize,compCode,compParm)` sets the tiling and compression for a field that had a fill value. This function must be applied after `gd.defField` and `gd.setFillValue`. The `compCode` argument can be one of the following strings.

`'rle'`	Run-length encoding
`'skphuff'`	Skipping Huffman
`'deflate'`	Deflate
`'none'`	No compression

`compParm` need only be specified when the compression scheme is `'deflate'`, and then must be an integer between 0 and 9.

This function corresponds to the `GDsettilecomp` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tilesize` parameter is reversed with respect to the C library API.

Examples

Define a temperature field with a 2-by-2 tiling scheme, a fill value of -999, and deflate compression.

```
import matlab.io.hdfeos.*
```

```
gfid = gd.open('myfile.hdf','create');
upleft = [210584.50041 3322395.95445];
lowright = [813931.10959 2214162.53278];
gridID = gd.create(gfid,'UTMGrid',120,200,upleft,lowright);
spherecode = 0; zonecode = 40;
projparm = zeros(1,13);
gd.defProj(gridID,'utm',zonecode,spherecode,projparm);
gd.defDim(gridID,'Time',10);
gd.defField(gridID,'Pollution',{'XDim','YDim','Time'},'float');
gd.setFillValue(gridID,'Pollution',single(7));
gd.setTileComp(gridID,'Pollution',[40 20 1],'deflate',5);
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.defComp | gd.defTile

matlab.io.hdfeos.gd.sphereCodeToName

Package: matlab.io.hdfeos.gd

Name corresponding to GCTP sphere code

Syntax

```
name = sphereCodeToName(code)
```

Description

`name = sphereCodeToName(code)` returns the name for the spheroid corresponding to the spheroid code. The list of supported GCTP spheroids is as follows:

GCTP Spheroid Code	Spheroid Name
0	Clarke 1866
1	Clarke 1880
2	Bessel
3	International 1967
4	International 1909
5	WGS 72
6	Everest
7	WGS 66
8	GRS 1980
9	Airy
10	Modified Airy
11	Modified Everest
12	WGS 84
13	Southeast Asia
14	Australian National

GCTP Spheroid Code	Spheroid Name
15	Krassovsky
16	Hough
17	Mercury 1960
18	Modified Mercury 1968
19	Sphere of radius 6370997m
20	Sphere of radius 6371228m
21	Sphere of radius 6371007.181m

See Also

`gd.defProj` | `gd.sphereNameToCode`

matlab.io.hdfeos.gd.sphereNameToCode

Package: matlab.io.hdfeos.gd

Numeric GCTP code corresponding to sphere name

Syntax

```
code = sphereCodeToName(name)
```

Description

`code = sphereCodeToName(name)` returns the numeric GCTP code corresponding to the named spheroid. The list of supported GCTP spheroids is as follows:

GCTP Spheroid Code	Spheroid Name
0	Clarke 1866
1	Clarke 1880
2	Bessel
3	International 1967
4	International 1909
5	WGS 72
6	Everest
7	WGS 66
8	GRS 1980
9	Airy
10	Modified Airy
11	Modified Everest
12	WGS 84
13	Southeast Asia
14	Australian National

GCTP Spheroid Code	Spheroid Name
15	Krassovsky
16	Hough
17	Mercury 1960
18	Modified Mercury 1968
19	Sphere of radius 6370997m
20	Sphere of radius 6371228m
21	Sphere of radius 6371007.181m

See Also

`gd.defProj` | `gd.sphereCodeToName`

matlab.io.hdfeos.gd.tileInfo

Package: matlab.io.hdfeos.gd

Tile size of grid field

Syntax

```
tileDims = tileInfo(gridID,fieldname)
```

Description

`tileDims = tileInfo(gridID,fieldname)` returns the tile dimensions of the field specified by `fieldname` in the grid specified by `gridID`. If the field is not tiled, then `tileDims` is `[]`.

This function corresponds to the `GDtileinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileDims` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
tileDims = gd.tileInfo(gridID,'pressure');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defTile`

matlab.io.hdfeos.gd.writeAttr

Package: matlab.io.hdfeos.gd

Write grid attribute

Syntax

```
writeAttr(gridID,attrname,data)
```

Description

`writeAttr(gridID,attrname,data)` writes an attribute to a grid. If the attribute does not exist, it is created. If the attribute exists, it can be modified in place, but it cannot be recreated with a different data type or length.

This function corresponds to the `GDwriteattr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','grid.hdf');
copyfile(srcFile,'myfile.hdf');
fileattrib('myfile.hdf','+w');
gfid = gd.open('myfile.hdf','rdwr');
gridID = gd.attach(gfid,'PolarGrid');
gd.writeAttr(gridID,'modification_date',datestr(now));
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.readAttr`

matlab.io.hdfeos.gd.writeBlkSomOffset

Package: matlab.io.hdfeos.gd

Write Block SOM offset

Syntax

```
writeBlkSomOffset(gridID,offset)
```

Description

`writeBlkSomOffset(gridID,offset)` writes the block SOM offset values n pixels for a standard Solar Oblique Mercator (SOM) projection. `offset` is a vector of offset values for SOM projection data. This routine can only be used with grids that use the SOM projection. You must take care to use this function properly in conjunction with `gd.defProj`. The 12th element of the projection parameters must be set to the total number of blocks to be defined. `offset` starts by listing the offset to the second block, so the 12th element of the projection parameters is always one more than the length of `offset`.

All fields defined after writing the block SOM offset values will automatically include "SOMBlockDim" as the slowest varying dimension.

This function corresponds to the `GDblkSOMoffset` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
lowright = [30521379.68485 1152027.64253];
upleft = [-11119487.42844 8673539.24806];
gridID = gd.create(gfid,'SOM',120,60,upleft,lowright);
projparm(1) = 6378137;
projparm(2) = 0.006694348;
projparm(4) = 98096360;  % 98.161 in DDDMMMSSS
```

```
projparm(5) = 87069061;  % 87.112 in DDDMMMSSS
projparm(9) = 0.068585416*1440;
projparm(10) = 0.0;
projparm(12) = 6;
gd.defProj(gridID,'som',[],[],projparm);
gd.writeBlkSomOffset(gridID,[5 10 12 8 2]);
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.readBlkSomOffset

matlab.io.hdfeos.gd.writeField

Package: matlab.io.hdfeos.gd

Write data to grid field

Syntax

```
writeField(gridID,fieldname,data)
writeField(gridID,fieldname,start,data)
writeField(gridID,fieldname,start,stride,data)
```

Description

`writeField(gridID,fieldname,data)` writes all the data to a grid field. The field is identified by `fieldname` and the grid is identified by `gridID`.

`writeField(gridID,fieldname,start,data)` writes a contiguous hyperslab to the grid field. `start` specifies the zero-based starting index.

`writeField(gridID,fieldname,start,stride,data)` writes a strided hyperslab of data to a grid data field. `stride` specifies the inter-element spacing along each dimension. The number of elements to write along each dimension is inferred from the size of `data`.

This function corresponds to the `GDwritefield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `start` and `stride` parameters are reversed with respect to the C library API.

Examples

Write all the data to a grid field.

```
import matlab.io.hdfeos.*
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','grid.hdf');
copyfile(srcFile,'myfile.hdf');
fileattrib('myfile.hdf','+w');
```

```
gfid = gd.open('myfile.hdf','rdwr');
gridID = gd.attach(gfid,'PolarGrid');
data = zeros(100,100,'uint16');
gd.writeField(gridID,'ice_temp',data);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.readField`

matlab.io.hdfeos.gd.writeTile

Package: matlab.io.hdfeos.gd

Write tile to field

Syntax

```
writeTile(gridID,fieldname,tileCoords,data)
```

Description

`writeTile(gridID,fieldname,tileCoords,data)` writes a single tile of data to a field. If the field data can be arranged tile by tile, this routine is more efficient than `gd.writeField`. In all other cases, use `gd.writeField`. The `tileCoords` argument has the form `[rownum colnum]` and is defined in terms of the tile coordinates, not the data elements.

This function corresponds to the `GDwritetile` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileCoords` parameter is reversed with respect to the C library API.

Examples

Define a field with a 2-by-3 tiling scheme.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
xdim = 200; ydim = 180;
gridID = gd.create(gfid,'PolarGrid',xdim,ydim,[],[]);
zonecode = 40;
spherecode = 0;
projParm = zeros(1,13);
projParm(6) =  90000000;
gd.defProj(gridID,'ps',[],spherecode,projParm);
tileSize = [100 60];
gd.defTile(gridID,tileSize);
```

```
dimlist = {'XDim','YDim'};
gd.defField(gridID,'Pressure',dimlist,'int32');
for c = 0:2
    for r = 0:1
        data = (r+c)*ones(tileSize,'int32');
        gd.writeTile(gridID,'Pressure',[r c],data);
    end
end
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.readTile

matlab.io.hdfeos.sw

Low-level access to HDF-EOS swath files

Description

To use these MATLAB functions, you must be familiar with the HDF-EOS library C interface. In most cases, the syntax of the MATLAB function is similar to the syntax of the corresponding HDF-EOS library function. The functions are implemented as the package `matlab.io.hdfeos.sw`. To use this package, prefix the function name with a package path, or use the import function to add the package to the current import list, prior to calling the function, for example,

```
import matlab.io.hdfeos.*
fileId = sw.open(filename);
```

Access

matlab.io.hdfeos.sw.attach	Attach to swath data set
matlab.io.hdfeos.sw.close	Close swath file
matlab.io.hdfeos.sw.create	Create new swath structure
matlab.io.hdfeos.sw.detach	Detach from swath
matlab.io.hdfeos.sw.open	Open swath file

Definition

matlab.io.hdfeos.sw.defComp	Set grid field compression

matlab.io.hdfeos.sw.defDataField	Define new data field within swath
matlab.io.hdfeos.sw.defDim	Define new dimension within swath
matlab.io.hdfeos.sw.defDimMap	Define mapping between geolocation and data dimensions
matlab.io.hdfeos.sw.defGeoField	Define new data field within swath

Basic I/O

matlab.io.hdfeos.sw.getFillValue	Fill value for specified field
matlab.io.hdfeos.sw.readAttr	Read swath attribute
matlab.io.hdfeos.sw.readField	Read data from swath field
matlab.io.hdfeos.sw.setFillValue	Set fill value for the specified field
matlab.io.hdfeos.sw.writeAttr	Write swath attribute
matlab.io.hdfeos.sw.writeField	Write data to swath field

Inquiry

matlab.io.hdfeos.sw.compInfo	Compression information for field
matlab.io.hdfeos.sw.dimInfo	Size of dimension
matlab.io.hdfeos.sw.fieldInfo	Information about swath field

matlab.io.hdfeos.sw.geoMapInfo	Type of dimension mapping for named dimension
matlab.io.hdfeos.sw.idxMapInfo	Indexed array of geolocation mapping
matlab.io.hdfeos.sw.inqAttrs	Names of swath attributes
matlab.io.hdfeos.sw.inqDataFields	Information about geolocation fields
matlab.io.hdfeos.sw.inqDims	Information about dimensions defined in swath
matlab.io.hdfeos.sw.inqGeoFields	Information about geolocation fields
matlab.io.hdfeos.sw.inqIdxMaps	Information about swath indexed geolocation mapping
matlab.io.hdfeos.sw.inqMaps	Information about swath geolocation relations
matlab.io.hdfeos.sw.inqSwath	Names of swaths in file
matlab.io.hdfeos.sw.mapInfo	Offset and increment of specific geolocation mapping
matlab.io.hdfeos.sw.nEntries	Number of entries for specific type

Subsetting

matlab.io.hdfeos.sw.defBoxRegion	Define latitude-longitude region for swath
matlab.io.hdfeos.sw.defTimePeriod	Define time period of interest

matlab.io.hdfeos.sw.defVrtRegion	Subset on monotonic field or dimension
matlab.io.hdfeos.sw.extractPeriod	Read data from subsetted time period
matlab.io.hdfeos.sw.extractRegion	Read subsetted region
matlab.io.hdfeos.sw.periodInfo	Information about subsetted period
matlab.io.hdfeos.sw.regionInfo	Information about subsetted region

matlab.io.hdfeos.sw.attach

Package: matlab.io.hdfeos.sw

Attach to swath data set

Syntax

```
swathID = attach(swfID,swathname)
```

Description

`swathID = attach(swfID,swathname)` attaches to the swath identified by `swathname` in the file identified by `swfID`. The `swathID` output is the identifier for the named swath.

This function corresponds to the `SWattach` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.detach`

matlab.io.hdfeos.sw.close

Package: matlab.io.hdfeos.sw

Close swath file

Syntax

```
close(swfID)
```

Description

`close(swfID)` closes an HDF-EOS swath file identified by `swfID`.

This function corresponds to the `SWclose` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'ExampleSwath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.create` | `sw.open`

matlab.io.hdfeos.sw.compInfo

Package: matlab.io.hdfeos.sw

Compression information for field

Syntax

```
[code,parms] = compInfo(swathID,fieldname)
```

Description

`[code,parms] = compInfo(swathID,fieldname)` returns the compression code and compression parameters for a given field. Refer to `sw.defComp` for a description of various compression schemes and parameters.

This function corresponds to the `SWcompinfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[compCode,parms] = sw.compInfo(swathID,'Spectra');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defComp`

matlab.io.hdfeos.sw.create

Package: matlab.io.hdfeos.sw

Create new swath structure

Syntax

```
swathID = create(swfID,swathname)
```

Description

`swathID = create(swfID,swathname)` creates a new swath structure where `swfID` is the swath file identifier and `swathname` is the name of the new swath. The swath is created as a Vgroup with the HDF file with the name `swathname` and HDF Vgroup class `'SWATH'`.

This function corresponds to the `SWcreate` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'ExampleSwath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.detach`

matlab.io.hdfeos.sw.defBoxRegion

Package: matlab.io.hdfeos.sw

Define latitude-longitude region for swath

Syntax

```
regionID = defBoxRegion(swathID,lat,lon,mode)
```

Description

`regionID = defBoxRegion(swathID,lat,lon,mode)` defines a latitude-longitude box region for a swath. `lat` and `lon` are two-element arrays containing the latitude and longitude in decimal degrees of the box corners. A cross track is determined to be within the box if a condition is met according to the value of `mode`:

`'MIDPOINT'`	The cross track midpoint is within the box.
`'ENDPOINT'`	Either endpoint is within the box.
`'ANYPOINT'`	Any point of the cross track is within the box.

All elements of a cross track are within the region if the condition is met. The swath must have both Longitude and Latitude (or Colatitude) defined.

`regionID` is an identifier to be used by `sw.extractRegion` to read all the entries of a data field within the region.

This function corresponds to the `SWdefboxregion` and `SWregionindex` functions in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
lat = [34 44];
```

```
lon = [16 24];
regionID = sw.defBoxRegion(swathID,lat,lon,'MIDPOINT');
data = sw.extractRegion(swathID,regionID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.extractRegion

matlab.io.hdfeos.sw.defComp

Package: matlab.io.hdfeos.sw

Set grid field compression

Syntax

```
defComp(swathID,compscheme,compparm)
```

Description

`defComp(swathID,compscheme,compparm)` sets the field compression for subsequent definitions. The compression scheme does not apply to one-dimensional fields. `compscheme` can be one of the following strings.

`'rle'`	Run-length encoding
`'skphuff'`	Skipping Huffman
`'deflate'`	Gzip compression
`'none'`	No compression

When the compression scheme is `'deflate'`, the `compparm` input is the deflate compression level, an integer between 0 and 9. `compparm` can be omitted for the other compression schemes.

Fields defined with compression must be written with a single call to `sw.writeField`.

This function corresponds to the `SWdefcomp` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'Track',4000);
```

```
sw.defDim(swathID,'Xtrack',2000);
sw.defDim(swathID,'Bands',3);
sw.defComp(swathID,'rle');
dims = {'Xtrack','Track'};
sw.defDataField(swathID,'Pressure',dims,'float');
sw.defComp(swathID,'deflate',5);
sw.defDataField(swathID,'Opacity',dims,'float');
sw.defComp(swathID,'skphuff');
dims = {'Xtrack','Track','Bands'};
sw.defDataField(swathID,'Spectra',dims,'float');
sw.defComp(swathID,'none');
dims = {'Xtrack','Track'};
sw.defDataField(swathID,'Temperature',dims,'float');
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.compInfo

matlab.io.hdfeos.sw.defDataField

Package: matlab.io.hdfeos.sw

Define new data field within swath

Syntax

```
defDataField(swathID,fieldname,dimlist,dtype)
defDataField(swathID,fieldname,dimlist,dtype,mergeCode)
```

Description

`defDataField(swathID,fieldname,dimlist,dtype)` defines a data field to be stored in the swath identified by `swathID`. The `dimlist` input can be a cell array of dimension names, or a single char if there is only one dimension. `dtype` is the data type of the field and can be one of the following strings.

- `'double'`
- `'single'`
- `'int32'`
- `'uint32'`
- `'int16'`
- `'uint16'`
- `'int8'`
- `'uint8'`
- `'char'`

`dimlist` should be ordered such that the fastest varying dimension is listed first. This is opposite from the order in which the dimensions are listed in the C API.

`defDataField(swathID,fieldname,dimlist,dtype,mergeCode)` defines a data field that can be merged with other data fields according to the value of `mergeCode`. The `mergeCode` input can be one of two strings, `'automerge'` and `'nomerge'`. If `mergeCode` is `'automerge'`, then the HDF-EOS library will attempt to merge swath

fields into a single object. This should not be done if you wish to access the swath fields individually with the another interface. By default, `mergeCode` is `'nomerge'`.

Note: To assure that the fields defined by `sw.defDataField` are properly established in the file, the swath should be detached and then reattached before writing to any fields.

This function corresponds to the `SWdefdatafield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.defDim(swathID,'Bands',3);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
dims = {'GeoXtrack','GeoTrack'};
sw.defGeoField(swathID,'Longitude',dims,'float');
sw.defGeoField(swathID,'Latitude',dims,'float');
dims = {'DataXtrack','DataTrack','Bands'};
sw.defDataField(swathID,'Spectra',dims,'float');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defGeoField` | `sw.inqDataFields`

matlab.io.hdfeos.sw.defDim

Package: matlab.io.hdfeos.sw

Define new dimension within swath

Syntax

```
defDim(swathID,dimname,dimlen)
```

Description

`defDim(swathID,dimname,dimlen)` defines a new dimension named `dimname` with length `dimlen` in the swath structure identified by `swathID`.

To specify an unlimited dimension, use either 0 or `'unlimited'` for `dimlen`.

This function corresponds to the `SWdefdim` function in the HDF-EOS library.

Examples

Define a dimension `'Band'` with length of 15 and an unlimited dimension `'Time'`.

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.dimInfo`

matlab.io.hdfeos.sw.defDimMap

Package: matlab.io.hdfeos.sw

Define mapping between geolocation and data dimensions

Syntax

```
defDimMap(swathID,geoDim,dataDim,offset,increment)
```

Description

`defDimMap(swathID,geoDim,dataDim,offset,increment)` defines a monotonic mapping between the geolocation and data dimensions, which usually have differing lengths. `offset` gives the index of the data element corresponding to the first geolocation element, and `increment` gives the number of data elements to skip for each geolocation element. If the geolocation dimension begins "before" the data dimension, then `offset` is negative. Similarly, if the geolocation dimension has higher resolution than the data dimension, then `increment` is negative.

This function corresponds to the `SWdefdimmap` function in the HDF-EOS library.

Examples

Create a dimension mapping such that the first element of the GeoTrack dimension corresponds to the first element of the DataTrack Dimension and such that the data dimension has twice the resolution as the geolocation dimension. Also create a dimension mapping such that the first element of the GeoXtrack dimension corresponds to the second element of the DataXtrack dimensions and such that the data dimension has twice the resolution as the geolocation dimension.

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
```

```
sw.defDim(swathID,'DataXtrack',2000);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDim` | `sw.mapInfo`

matlab.io.hdfeos.sw.defGeoField

Package: matlab.io.hdfeos.sw

Define new data field within swath

Syntax

```
defGeoField(swathID,fieldname,dimlist,dtype)
defGeoField(swathID,fieldname,dimlist,dtype,mergeCode)
```

Description

`defGeoField(swathID,fieldname,dimlist,dtype)` defines a geolocation field to be stored in the swath identified by `swathID`. The `dimlist` argument can be a cell array of dimension names or a single char if there is only one dimension. `dtype` is the data type of the field

`dimlist` should be ordered such that the fastest varying dimension is listed first. This is opposite from the order in which the dimensions are listed in the C API.

`defGeoField(swathID,fieldname,dimlist,dtype,mergeCode)` defines a geolocation field that may be merged with other geolocation fields according to the value of `mergeCode`. The `mergeCode` argument can be one of two strings, `'automerge'` and `'nomerge'`. If `mergeCode` is `'automerge'`, then the HDF-EOS library will attempt to merge swath fields into a single object. This should not be done if you wish to access the swath fields individually with the another interface. By default, `mergeCode` is `'nomerge'`.

This function corresponds to the `SWdefgeofield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
```

```
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
dims = {'GeoXtrack','GeoTrack'};
sw.defGeoField(swathID,'Longitude',dims,'float');
sw.defGeoField(swathID,'Latitude',dims,'float');
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.defDataField | sw.inqGeoFields

matlab.io.hdfeos.sw.defTimePeriod

Package: matlab.io.hdfeos.sw

Define time period of interest

Syntax

```
outpID = defTimePeriod(swathID,start,stop,mode)
```

Description

`outpID = defTimePeriod(swathID,start,stop,mode)` defines a time period for a swath. `outpID` is a swath period ID that can be used to read all the entries of a data field within the time period. The swath structure must have the `'Time'` field defined. A cross track is within a time period if a condition is met according to the value of `mode`:

`'MIDPOINT'`	The midpoint is within the time period.
`'ENDPOINT'`	Either endpoint is within the time period.

This function corresponds to the `SWdeftimeperiod` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
starttime =  25;
stoptime = 425;
periodID = sw.defTimePeriod(swathID,starttime,stoptime,'MIDPOINT');
data = sw.extractPeriod(swathID,periodID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defBoxRegion` | `sw.defVrtRegion` | `sw.extractPeriod`

matlab.io.hdfeos.sw.defVrtRegion

Package: matlab.io.hdfeos.sw

Subset on monotonic field or dimension

Syntax

```
regionID_out = defVrtRegion(swathID,regionID,vertObj,range)
```

Description

`regionID_out = defVrtRegion(swathID,regionID,vertObj,range)` subsets on a monotonic field or contiguous elements of a dimension. Whereas `defBoxRegion` and `defTimePeriod` subset along the `'Track'` dimension, this routine allows the user to subset along any dimension. `regionID` specifies the subsetted region from a previous call. `vertObj` specifies the dimension by which to subset. `range` specifies the minimum and maximum values for `vertObj`.

If there is no current subsetted region, `regionID` should be `'noprevsub'`.

`vertObj` can be either a dimension or a field. If it is a dimension, then `range` should consist of dimension indices. If `vertObj` corresponds to a field, then `range` should consist of the minimum and maximum field values. `vertObj` must be one-dimensional in this case, and the its values must be monotonic.

This function corresponds to the `SWdefvrtregion` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
regionID = sw.defVrtRegion(swathID,'noprevsub','Bands',[450 600]);
data = sw.extractRegion(swathID,regionID,'Spectra');
sw.detach(swathID);
```

```
sw.close(swfid);
```

See Also

sw.defBoxRegion | sw.defTimePeriod

matlab.io.hdfeos.sw.detach

Package: matlab.io.hdfeos.sw

Detach from swath

Syntax

```
detach(swathID)
```

Description

`detach(swathID)` detaches from the swath identified by `swathID`.

This function corresponds to the `SWdetach` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.attach` | `sw.create`

matlab.io.hdfeos.sw.dimInfo

Package: matlab.io.hdfeos.sw

Size of dimension

Syntax

```
dimlen = dimInfo(swathID,dimname)
```

Description

`dimlen = dimInfo(swathID,dimname)` returns the length of the specified dimension.

This function corresponds to the `SWdiminfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
dimlen = sw.dimInfo(swathID,'GeoTrack');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDim`

matlab.io.hdfeos.sw.extractPeriod

Package: matlab.io.hdfeos.sw

Read data from subsetted time period

Syntax

```
data = extractPeriod(swathID,periodID,fieldname)
```

Description

`data = extractPeriod(swathID,periodID,fieldname)` reads data for the given field for the time period specified by `periodID`.

This routine corresponds to the `SWextractperiod` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
starttime =  25;
stoptime = 425;
periodID = sw.defTimePeriod(swathID,starttime,stoptime,'MIDPOINT');
data = sw.extractPeriod(swathID,periodID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defTimePeriod`

matlab.io.hdfeos.sw.extractRegion

Package: matlab.io.hdfeos.sw

Read subsetted region

Syntax

```
data = extractRegion(swathID,regionID,fieldname)
```

Description

`data = extractRegion(swathID,regionID,fieldname)` reads data for a specified field from a subsetted region identified by `regionID`.

This function corresponds to the `SWextractregion` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
lat = [34 44];
lon = [16 24];
regionID = sw.defBoxRegion(swathID,lat,lon,'MIDPOINT');
data = sw.extractRegion(swathID,regionID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defBoxRegion` | `sw.defVrtRegion`

matlab.io.hdfeos.sw.fieldInfo

Package: matlab.io.hdfeos.sw

Information about swath field

Syntax

```
[dimsizes,ntype,dimlist] = fieldInfo(swathID,fieldname)
```

Description

`[dimsizes,ntype,dimlist] = fieldInfo(swathID,fieldname)` returns the size, data type, and list of named dimensions for the specified swath geolocation or data field.

This function corresponds to the `SWfieldinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[fieldSize,ntype,dimlist] = sw.fieldInfo(swathID,'Spectra');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.inqDataFields` | `sw.inqGeoFields`

matlab.io.hdfeos.sw.geoMapInfo

Package: matlab.io.hdfeos.sw

Type of dimension mapping for named dimension

Syntax

```
mappingType = geoMapInfo(swathID,dimname)
```

Description

`mappingType = geoMapInfo(swathID,dimname)` returns the type of dimension mapping for the named dimension. `mappingType` is one of `'indexed'`, `'regular'`, or `'unmapped'`.

This routine corresponds to the `SWgeomapinfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
maptype = sw.geoMapInfo(swathID,'GeoTrack');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDimMap`

matlab.io.hdfeos.sw.getFillValue

Package: matlab.io.hdfeos.sw

Fill value for specified field

Syntax

```
fillvalue = getFillValue(swathID,fieldname)
```

Description

`fillvalue = getFillValue(swathID,fieldname)` returns the fill value for the specified field.

This function corresponds to the `SWgetfillvalue` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
fv = sw.getFillValue(swathID,'Spectra');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.setFillValue`

matlab.io.hdfeos.sw.idxMapInfo

Package: matlab.io.hdfeos.sw

Indexed array of geolocation mapping

Syntax

```
idx = idxMapInfo(swathID,geodim,datadim)
```

Description

`idx = idxMapInfo(swathID,geodim,datadim)` retrieves the indexed elements of the geolocation mapping between `geodim` and `datadim`.

This function corresponds to the `SWidxmapinfo` function in the HDF-EOS C library API.

See Also

`sw.geoMapInfo`

matlab.io.hdfeos.sw.inqAttrs

Package: matlab.io.hdfeos.sw

Names of swath attributes

Syntax

```
attrlist = inqAttrs(swathID)
```

Description

`attrlist = inqAttrs(swathID)` returns the list of swath attribute names. `attrlist` is a cell array.

This function corresponds to the `SWinqattrs` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
attrList = sw.inqAttrs(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.readAttr` | `sw.writeAttr`

matlab.io.hdfeos.sw.inqDataFields

Package: matlab.io.hdfeos.sw

Information about geolocation fields

Syntax

```
[fields,rank,datatype] = inqDataFields(swathID)
```

Description

`[fields,rank,datatype] = inqDataFields(swathID)` returns the list of geolocation field names, the rank of each field, and the data type of each field.

This function corresponds to the `SWinqdatafields` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `fields` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[fields,rank,datatype] = sw.inqDataFields(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDataField` | `sw.inqGeoFields`

matlab.io.hdfeos.sw.inqDims

Package: matlab.io.hdfeos.sw

Information about dimensions defined in swath

Syntax

```
[dimnames,dimlens] = inqDims(swathID)
```

Description

`[dimnames,dimlens] = inqDims(swathID)` returns the names of the dimensions `dimnames` as a cell array. The length of each respective dimension is returned in `dimlens`.

This function corresponds to the `SWinqdims` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[dimnames,dimlens] = sw.inqDims(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDim`

matlab.io.hdfeos.sw.inqGeoFields

Package: matlab.io.hdfeos.sw

Information about geolocation fields

Syntax

```
[fields,rank,datatype] = inqGeoFields(swathID)
```

Description

`[fields,rank,datatype] = inqGeoFields(swathID)` returns the list of geolocation fields `fields`, the rank of each field, and the data type of each field.

This function corresponds to the `SWinqgeofields` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `fields` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[fields,rank,datatypes] = sw.inqGeoFields(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defGeoField` | `sw.inqDataFields`

matlab.io.hdfeos.sw.inqIdxMaps

Package: matlab.io.hdfeos.sw

Information about swath indexed geolocation mapping

Syntax

```
[idxMap,idxSize] = inqIdxMaps(swathID)
```

Description

`[idxMap,idxSize] = inqIdxMaps(swathID)` retrieves all indexed geolocation/data mappings defined in the swath. `idxMap` is a cell array with each element consisting of the names of the dimensions of a mapping, separated by a `'/'`. `idxSize` contains the size of the index arrays corresponding to each mapping.

This function corresponds to the `SWinqidxmaps` routine in the HDF-EOS library.

See Also

`sw.inqMaps`

matlab.io.hdfeos.sw.inqMaps

Package: matlab.io.hdfeos.sw

Information about swath geolocation relations

Syntax

```
[map,offset,increment] = inqMaps(swathID)
```

Description

`[map,offset,increment] = inqMaps(swathID)` returns the dimension mapping list, the offset of each geolocation relation, and the increment of each geolocation relation. These mappings are not indexed. `map` is a cell array where each element contains the names of the dimensions for each mapping, separated by a slash. `offset` and `increment` contain the offset and increment of each geolocation relation.

This function corresponds to the `SWinqmaps` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[dimmap,offset,increment] = sw.inqMaps(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDimMap` | `sw.inqDims` | `sw.inqIdxMaps`

matlab.io.hdfeos.sw.inqSwath

Package: matlab.io.hdfeos.sw

Names of swaths in file

Syntax

```
swaths = inqSwath(filename)
```

Description

`swaths = inqSwath(filename)` returns a cell array containing the names of all the swaths in a file.

This function corresponds to the `SWinqswath` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swaths = sw.inqSwath('swath.hdf');
```

See Also

`gd.inqGrid`

matlab.io.hdfeos.sw.mapInfo

Package: matlab.io.hdfeos.sw

Offset and increment of specific geolocation mapping

Syntax

```
[offset,increment] = mapInfo(swathID,geodim,datadim)
```

Description

`[offset,increment] = mapInfo(swathID,geodim,datadim)` retrieves the offset and increment of the geolocation mapping between the specified geolocation dimension and the specified data dimension.

This function corresponds to the `SWmapinfo` function in the HDF-EOS C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
sw.detach(swathID);
sw.close(swfid);
swfid = sw.open('myfile.hdf','read');
swathID = sw.attach(swfid,'MySwath');
[offset,increment] = sw.mapInfo(swathID,'GeoTrack','DataTrack');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDimMap`

matlab.io.hdfeos.sw.nEntries

Package: matlab.io.hdfeos.sw

Number of entries for specific type

Syntax

```
nEnts = nEntries(swathID,type)
```

Description

`nEnts = nEntries(swathID,type)` returns the number of entries in a swath. Valid inputs for `type` include:

`'dims'`	or `'HDFE_NENTDIM'`
`'maps'`	or `'HDFE_NENTMAP'`
`'imaps'`	or `'HDFE_NENTIMAP'`
`'geofields'`	or `'HDFE_NENTGFLD'`
`'datafields'`	or `'HDFE_NENTFLD'`

This function corresponds to the `SWnentries` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
ndims = sw.nEntries(swathID,'dims');
sw.detach(swathID);
sw.close(swfid);
```

matlab.io.hdfeos.sw.open

Package: matlab.io.hdfeos.sw

Open swath file

Syntax

```
swfID = open(filename)
swfID = open(filename,access)
```

Description

`swfID = open(filename)` opens an HDF-EOS swath file for read-only access.

`swfID = open(filename,access)` opens or creates an HDF-EOS swath file identified by `filename` and returns a file ID. `access` can be one of the following string values.

`'read'` (default)	Read-only
`'rdwr'`	Read-write
`'create'`	Creates a file, deleting it if it already exists

This routine corresponds to the `SWopen` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
sw.close(swfid);
```

See Also

`sw.close`

matlab.io.hdfeos.sw.periodInfo

Package: matlab.io.hdfeos.sw

Information about subsetted period

Syntax

```
[datatype,dims] = periodInfo(swathID,periodID,fieldname)
```

Description

`[datatype,dims] = periodInfo(swathID,periodID,fieldname)` retrieves information about the period defined for the given field. `datatype` is the data type of the field. `dims` is the dimensions of the subsetted region.

This function corresponds to the `SWperiodinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dims` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
starttime =  25;
stoptime = 425;
periodID = sw.defTimePeriod(swathID,starttime,stoptime,'MIDPOINT');
[ntype,dims] = sw.periodInfo(swathID,periodID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defTimePeriod` | `sw.extractPeriod`

matlab.io.hdfeos.sw.readAttr

Package: matlab.io.hdfeos.sw

Read swath attribute

Syntax

```
data = readAttr(swathID,attrname)
```

Description

`data = readAttr(swathID,attrname)` reads a swath attribute.

This function corresponds to the `SWreadAttr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
value = sw.readAttr(swathID,'creation_date');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.writeAttr`

matlab.io.hdfeos.sw.readField

Package: matlab.io.hdfeos.sw

Read data from swath field

Syntax

```
data = readField(swathID,fieldname)
data = readField(swathID,fieldname,start,count)
data = readField(swathID,fieldname,start,count,stride)
```

Description

`data = readField(swathID,fieldname)` reads an entire swath field.

`data = readField(swathID,fieldname,start,count)` reads a contiguous hyperslab of data from the swath field `fieldname`. The `start` input specifies the zero-based index of the first element to be read. `count` specifies the number of elements along each dimension to read.

`data = readField(swathID,fieldname,start,count,stride)` reads a strided hyperslab of data from the swath field `fieldname`. The `stride` input specifies the inter-element spacing along each dimension.

This function corresponds to the `SWreadfield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `start`, `count`, and `stride` parameters are reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
data = sw.readField(swathID,'Longitude');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.writeField`

matlab.io.hdfeos.sw.regionInfo

Package: matlab.io.hdfeos.sw

Information about subsetted region

Syntax

```
[datatype,extent] = regionInfo(swathID,regionID,fieldname)
```

Description

`[datatype,extent] = regionInfo(swathID,regionID,fieldname)` returns the data type and extent of a subsetted region of a field. `regionID` is the identifier for the subsetted region.

This function corresponds to the `SWregioninfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `extent` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
lat = [34 44];
lon = [16 24];
regionID = sw.defBoxRegion(swathID,lat,lon,'MIDPOINT');
[ntype,dims] = sw.regionInfo(swathID,regionID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defBoxRegion` | `sw.defVrtRegion`

matlab.io.hdfeos.sw.setFillValue

Package: matlab.io.hdfeos.sw

Set fill value for the specified field

Syntax

```
setFillValue(swathID,fieldname,fillvalue)
```

Description

`setFillValue(swathID,fieldname,fillvalue)` sets the fill value for the specified field. The field must have more than two dimensions.

This function corresponds to the `SWsetfillvalue` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'Track',400);
sw.defDim(swathID,'Xtrack',200);
dims = {'Track','Xtrack'};
sw.defDataField(swathID,'Temperature',dims,'float');
sw.setFillValue(swathID,'Temperature',single(-999));
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.getFillValue`

matlab.io.hdfeos.sw.writeAttr

Package: matlab.io.hdfeos.sw

Write swath attribute

Syntax

```
writeAttr(swathID,attrname,data)
```

Description

`writeAttr(swathID,attrname,data)` writes an attribute to a swath. If the attribute does not exist, it is created. If the attribute exists, it can be modified in place, but it cannot be recreated with a different data type or length.

This function corresponds to the `SWwriteattr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.writeAttr(swathID,'creation_date', datestr(now));
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.readAttr`

matlab.io.hdfeos.sw.writeField

Package: matlab.io.hdfeos.sw

Write data to swath field

Syntax

```
writeField(swathID,fieldname,data)
writeField(swathID,fieldname,start,data)
writeField(swathID,fieldname,start,stride,data)
```

Description

`writeField(swathID,fieldname,data)` writes an entire swath data field.

`writeField(swathID,fieldname,start,data)` writes a contiguous hyperslab to a swath field. `start` specifies the index of the first element to write. The number of elements along each dimension is inferred from either the size of `data` or from the swath field itself.

`writeField(swathID,fieldname,start,stride,data)` writes a strided hyperslab to a swath field. `stride` specifies the inter-element spacing along each dimension.

This function corresponds to the `SWwritefield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `start` and `stride` parameters are reversed with respect to the C library API.

Examples

Write data to a geolocation field `'Longitude'`.

```
lon = [-50:49];
data = repmat(lon(:),1,100);
data = single(data);
import matlab.io.hdfeos.*
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','swath.hdf');
```

```
copyfile(srcFile,'myfile.hdf');
fileattrib('myfile.hdf','+w');
swfid = sw.open('myfile.hdf','rdwr');
swathID = sw.attach(swfid,'Example Swath');
sw.writeField(swathID,'Longitude',data);
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.readField

matlab.io.saveVariablesToScript

Save workspace variables to MATLAB script

Syntax

```
matlab.io.saveVariablesToScript(filename)
matlab.io.saveVariablesToScript(filename,varnames)
matlab.io.saveVariablesToScript(filename,Name,Value)
[r1,r2] = matlab.io.saveVariablesToScript(filename)
```

Description

`matlab.io.saveVariablesToScript(filename)` saves variables in the current workspace to a MATLAB script named `filename.m`. The filename can include the `.m` suffix. If you do not include it, the function adds it when it creates the file.

Variables that MATLAB cannot generate code for are saved to a MAT-file named `filename.mat`.

If a file with the same name already exists, it is overwritten.

`matlab.io.saveVariablesToScript(filename,varnames)` saves only workspace variables specified by `varnames` to the MATLAB script.

`matlab.io.saveVariablesToScript(filename,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

`[r1,r2] = matlab.io.saveVariablesToScript(filename)` additionally returns two cell arrays:

- `r1` for variables that were saved to the MATLAB script
- `r2` for variables that were saved to a MAT-file

Examples

Save Workspace Variables to MATLAB Script

Save variables from a workspace to a MATLAB script, `test.m`.

```
matlab.io.saveVariablesToScript('test.m')
```

Save Specific Workspace Variables to MATLAB Script

Create and save variable `myVar` from a workspace to a MATLAB script, `test.m`.

```
myVar = 55.3;
matlab.io.saveVariablesToScript('test.m','myVar')
```

Append Specific Variables to Existing MATLAB Script

Create two variables, `a` and `b`, and save them to an existing MATLAB script `abfile.m`.

```
a = 72.3;
b = pi;
matlab.io.saveVariablesToScript('abfile.m',{'a','b'},...
'SaveMode','append')
```

Update Specific Variables in Existing MATLAB Script

Update and save two variables, `y` and `z`, to an existing MATLAB script `yzfile.m`.

```
y = 15.7;
z = 3 * pi;
matlab.io.saveVariablesToScript('yzfile.m',{'y','z'},...
'SaveMode','update')
```

Specify MATLAB Script Configuration for Saving Variable

Update and save variable `resistance` to an existing MATLAB script `designData.m` while specifying the configuration of the script file.

```
resistance = [10 20.5 11 13.7 15.1 7.7];
matlab.io.saveVariablesToScript('designData.m','resistance',...
'SaveMode','Update','MaximumArraySize',5,...
'MaximumNestingLevel',5,'MaximumTextWidth',30)
```

Specify 2-D Slice for Saving 3-D Array in MATLAB Script

Specify a 2-D slice for the output of the 3-D array `my3Dtable`, such that the 2-D slice expands along the first and third dimensions. Save the 2-D slice in the MATLAB script `sliceData.m`.

```
level1 = [1 2; 3 4];
```

```
level2 = [5 6; 7 8];
my3Dtable( :, :, 1) = level1;
my3Dtable( :, :, 2) = level2;
matlab.io.saveVariablesToScript('sliceData.m','MultidimensionalFormat',[1,3])
```

The resulting MATLAB code is similar to the following:

```
level1 = ...
 [1 2;
 3 4];

level2 = ...
 [5 6;
 7 8];
my3Dtable = zeros(2, 2, 2);
my3Dtable(:,1,:) = ...
 [1 5;
 3 7];

my3Dtable(:,2,:) = ...
 [2 6;
 4 8];
```

Save Variables Matching a Regular Expression

Save variables that match the expression `autoL*` to a MATLAB script `autoVariables.m`.

```
matlab.io.saveVariablesToScript('autoVariables.m','RegExp','autoL*')
```

Save Variables to Version 7.3 MATLAB Script

Create two variables, `p` and `q`, and save them to a version 7.3 MATLAB script `version73.m`.

```
p = 49;
q = 35.5;
matlab.io.saveVariablesToScript('version73.m','p','q',...
'MATFileVersion','v7.3')
```

Return Variables Saved to MATLAB Script

Save variables that were saved to a MATLAB script to the variable `r1`, and those that were saved to a MAT-file to the variable `r2`.

```
[r1,r2] = matlab.io.saveVariablesToScript('mydata.m')

r1 =

    'level1'
    'level2'
    'level3'
    'my3Dtable'


r2 =

   Empty cell array: 0-by-1
```

Input Arguments

`filename` — Name of MATLAB script for saving variables

filename | variable

Name of MATLAB script for saving variables, specified as a string giving a file name or a variable containing the file name.

Example: `matlab.io.saveVariablesToScript('myVariables.m')`

`varnames` — Name of variables to save

string | cell array

Name of variables to save, specified as a string or a cell array.

Example: `{'X','Y','Z'}`

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaximumArraySize',500,'MATFileVersion','v4'` specifies that the maximum number of array elements to save is `500` using MATLAB version 4 syntax.

'MATFileVersion' — MATLAB version whose syntax to use
'v7.3' (default) | 'v4' | 'v6' | 'v7'

MATLAB version whose syntax to use for saving MAT-files, specified as the comma-separated pair consisting of 'MATFileVersion' and one of the following version numbers:

- 'v4'
- 'v6'
- 'v7'
- 'v7.3'

Example: 'MATFileVersion','v6'

Data Types: char

'MaximumArraySize' — Maximum array elements to save
1000 (default) | integer

Maximum array elements to save, specified as the comma-separated pair consisting of 'MaximumArraySize' and an integer in the range of 1 to 10,000.

Example: 'MaximumArraySize',1050

'MaximumNestingLevel' — Maximum number of object levels or array hierarchies to save
20 (default) | integer

Maximum number of object levels or array hierarchies to save, specified as the comma-separated pair consisting of 'MaximumNestingLevel' and an integer in the range of 1 to 200.

Example: 'MaximumNestingLevel',67

'MaximumTextWidth' — Text wrap width during save
76 (default) | integer

Text wrap width during save, specified as the comma-separated pair consisting of 'MaximumTextWidth' and an integer in the range of 32 to 256.

Example: 'MaximumTextWidth',82

'MultidimensionalFormat' — Dimensions of 2-D slices that represent n-D arrays of char, logic, or numeric data
'rowvector' (default) | integer cell array

Dimensions of 2-D slices that represent n-D arrays of char, logic, or numeric data, specified as the comma-separated pair consisting of `'MultidimensionalFormat'` and one of these values:

- `'rowvector'` — Save multidimensional variables as a single row vector.
- `integer cell array` — Save a 2-D slice of multidimensional variables, where the dimensions satisfy all the following criteria:
  - Two positive integers represent dimensions.
  - The two integers are less than or equal to the dimensions of the n-D array.
  - The second integer is greater than the first.

Example: `'MultidimensionalFormat',[1,3]`

`'RegExp'` — Regular expression for matching

string

Regular expression for matching, specified as the comma-separated pair consisting of `'RegExp'` and one or more expressions given as a string.

Example: `'RegExp','level*'`

Data Types: `char`

`'SaveMode'` — Mode to save MATLAB script

`'create'` (default) | `'update'` | `'append'`

Mode to save MATLAB script, specified as the comma-separated pair consisting of `SaveMode` and one of these values:

- `'create'` — Save variables to a new MATLAB script.
- `'update'` — Only update variables that are already present in a MATLAB script.
- `'append'` — Update variables that are already present in a MATLAB script and append new variables to the end of the script.

Example: `'SaveMode','Update'`

Output Arguments

`r1` — Variables that were saved to a MATLAB script

cell array

Variables that were saved to a MATLAB script, returned as a cell array of variable names.

`r2` — Variables that were saved to a MAT-file
cell array

Variables that were saved to a MAT-file, returned as a cell array of variable names.

Limitations

- `matlab.io.saveVariablesToScript` does not save the following variables to a MATLAB script or a MAT-file.
  - Java objects
  - .NET objects
  - Python objects
- `matlab.io.saveVariablesToScript` saves the following variables only to a MAT-file.
  - MATLAB objects
  - Function handles
  - Anonymous functions

matlab.lang.makeUniqueStrings

Construct unique strings from input strings

Syntax

```
U = matlab.lang.makeUniqueStrings(S)

U = matlab.lang.makeUniqueStrings(S,excludedStrings)
U = matlab.lang.makeUniqueStrings(S,whichStrings)
U = matlab.lang.makeUniqueStrings(S, ___ , maxStringLength)

[U, modified] = matlab.lang.makeUniqueStrings( ___ )
```

Description

`U = matlab.lang.makeUniqueStrings(S)` constructs unique strings, `U`, from input strings, `S`, by appending an underscore and a number to duplicate strings.

`U = matlab.lang.makeUniqueStrings(S,excludedStrings)` constructs strings that are unique within `U` and with respect to `excludedStrings`. The `makeUniqueStrings` function does not check `excludedStrings` for uniqueness.

`U = matlab.lang.makeUniqueStrings(S,whichStrings)` specifies the subset of `S` to make unique within the entire set. `makeUniqueStrings` makes the strings in `S(whichStrings)` unique among themselves and with respect to the remaining strings. `makeUniqueStrings` returns the remaining strings unmodified in `U`. Use this syntax when you have an array of strings, and need to check that only some elements of the array are unique.

`U = matlab.lang.makeUniqueStrings(S, ___ , maxStringLength)` specifies the maximum length, `maxStringLength`, of strings in `U`. If `makeUniqueStrings` cannot make elements in `S` unique without exceeding `maxStringLength`, it returns an error. You can use this syntax with any of the input arguments of the previous syntaxes.

`[U, modified] = matlab.lang.makeUniqueStrings( ___ )` returns a logical array, `modified`, indicating modified strings.

Examples

Construct Unique Strings

Create an array of strings and make each element unique.

```
S = {'John' 'Sue' 'Nick' 'John' 'Campion' 'John' 'Jason'};
U = matlab.lang.makeUniqueStrings(S)
```

```
U =

    'John'    'Sue'    'Nick'    'John_1'    'Campion'    'John_2'    'Jason'
```

The `makeUniqueStrings` function appends the duplicate strings in elements 3 and 5 with underscores and incrementing numbers.

Construct Unique Strings and Specify Exclusions

Without specifying excluded strings, make the strings in `U` unique.

```
S = {'John' 'Sue' 'Nick' 'John' 'Campion' 'John' 'Jason'};
U = matlab.lang.makeUniqueStrings(S)
```

```
U =

    'John'    'Sue'    'Nick'    'John_1'    'Campion'    'John_2'    'Jason'
```

Specify that the string, `'Nick'`, should be excluded from the output.

```
U = matlab.lang.makeUniqueStrings(S, 'Nick')
```

```
U =

    'John'    'Sue'    'Nick_1'    'John_1'    'Campion'    'John_2'    'Jason'
```

`makeUniqueStrings` excludes `'Nick'` from `U` and instead modifies the first duplicate string, found in element 3, to be `'Nick_1'`.

Exclude workspace variables from the unique string array.

```
Sue = 42;
U = matlab.lang.makeUniqueStrings(S, who)


U =

    'John'    'Sue_1'    'Nick'    'John_1'    'Campion'    'John_2'    'Jason'
```

Since `'Sue'` exists in the workspace, `makeUniqueStrings` makes this string unique by appending an underscore and number.

Construct Unique Strings for Specified Array Elements

Create an array of strings and make only the first four elements unique.

```
S = {'quiz' 'quiz' 'quiz' 'exam' 'quiz' 'exam'};
U = matlab.lang.makeUniqueStrings(S, 1:4)


U =

    'quiz_1'    'quiz_2'    'quiz_3'    'exam_1'    'quiz'    'exam'
```

The first four elements in `U` are unique among themselves, and among the remaining strings in elements 5 and 6 (`'quiz'` and `'exam'`). Alternatively, you can use a logical array instead of a range of linear indices to achieve the same results: `U = matlab.lang.makeUniqueStrings(S, [true true true true false false])` or `U = matlab.lang.makeUniqueStrings(S, logical([1 1 1 1 0 0]))`.

Append a duplicate `'quiz'` onto the end of `S` and make the first four elements unique.

```
S{end+1} = 'quiz'
U = matlab.lang.makeUniqueStrings(S, 1:4)


S =

    'quiz'    'quiz'    'quiz'    'exam'    'quiz'    'exam'    'quiz'


U =

    'quiz_1'    'quiz_2'    'quiz_3'    'exam_1'    'quiz'    'exam'    'quiz'
```

The strings that `makeUniqueStrings` checks are still unique among themselves and among the remaining strings. Since `makeUniqueStrings` does not check any elements after element 4, duplicate strings remain.

Construct Unique Strings with Maximum Length

Create an array from `S` where the first three elements are unique and the maximum length of each string is 5.

```
S = {'sampleData' 'sampleData' 'sampleData' 'sampleData'};
U = matlab.lang.makeUniqueStrings(S, 1:3, 5)
```

```
U =

    'sampl'    'sam_1'    'sam_2'    'sampleData'
```

The first element is truncated to 5 characters. The second and third elements are truncated to 3 characters to allow `makeUniqueStrings` to append an underscore and number, and still not exceed 5 characters.

Determine Modified Strings

```
S = {'a%name', 'name_1', '2_name'};
[N, modified] = matlab.lang.makeValidName(S)
```

```
N =

    'a_name'    'name_1'    'x2_name'


modified =

     1     0     1
```

`makeValidName` did not modify the second element.

Input Arguments

S — Input strings

string or cell array of strings

Input strings, specified as a string or cell array of strings.

`excludedStrings` — Strings to exclude

string or cell array of strings

Strings to exclude from `U`, specified as a string or cell array of strings.

Example: `'dontDuplicateThisString'`, `{'excludeS1','excludeS2'}`, `who`

`whichStrings` — Subset of strings to make unique

range of linear indices or logical array

Subset of strings, `S`, to make unique within the entire set, specified as a range of linear indices or as a logical array with the same size and shape as `S`. If there are duplicates in `S`, the `makeUniqueStrings` function only modifies those specified by `whichStrings`.

If `whichStrings` is a logical array, strings are checked for uniqueness when the array element in the same position has a value of `true`.

Example: `1:5`, `logical([1 0 1])`, `[true false true]`

`maxStringLength` — Maximum length of output strings

integer

Maximum length of output strings in `U`, specified as an integer. If `makeUniqueStrings` cannot make elements in `S` unique without exceeding `maxStringLength`, it returns an error.

Output Arguments

`U` — Unique strings

string | cell array of strings

Unique strings, returned as a string or cell array of strings. The output has the same dimension as the input, `S`.

`modified` — Indicator of modified strings

logical scalar | logical array

Indicator of modified strings, returned as a logical scalar or array and having the same dimension as the input, `S`. A value of `1` (`true`) indicates that `makeUniqeStrings`

modified the input string in the corresponding location. A value of `0` (`false`) indicates that `makeUniqeStrings` did not need to modify the input string in the corresponding location.

More About

Tips

- To ensure strings are valid and unique, use `matlab.lang.makeValidName` before `matlab.lang.makeUniqueStrings`.

```
S = {'my.Name','my_Name','my_Name'};
validStrings = matlab.lang.makeValidName(S)
validUniqueStrings = matlab.lang.makeUniqueStrings(validStrings,...
    {},namelengthmax)
```

```
validStrings =

    'my_Name'    'my_Name'    'my_Name'


validUniqueStrings =

    'my_Name'    'my_Name_1'    'my_Name_2'
```

See Also

`matlab.lang.makeValidName` | `namelengthmax` | `who`

Introduced in R2014a

matlab.lang.makeValidName

Construct valid MATLAB identifiers from input strings

Syntax

```
N = matlab.lang.makeValidName(S)
N = matlab.lang.makeValidName(S,Name,Value)
[N, modified] = matlab.lang.makeValidName( ___ )
```

Description

`N = matlab.lang.makeValidName(S)` constructs valid MATLAB identifiers, `N`, from input strings, `S`. The `makeValidName` function does not guarantee the strings in `N` are unique.

A valid MATLAB identifier is a string of alphanumerics (A–Z, a–z, 0–9) and underscores, such that the first character is a letter and the length of the string is less than or equal to `namelengthmax`.

`makeValidName` deletes any whitespace characters prior to replacing any characters that are not alphanumerics or underscores. If a whitespace character is followed by a lowercase letter, `makeValidName` converts the letter to the corresponding uppercase character.

`N = matlab.lang.makeValidName(S,Name,Value)` includes additional options specified by one or more Name,Value pair arguments.

`[N, modified] = matlab.lang.makeValidName( ___ )` returns a logical array, `modified`, indicating modified strings. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Construct Valid MATLAB Identifiers

```
S = {'Item_#','Price/Unit','1st order','Contact'};
N = matlab.lang.makeValidName(S)
```

```
N =

    'Item__'    'Price_Unit'    'x1stOrder'    'Contact'
```

In the first and second strings, `makeValidName` replaced the invalid characters (`#` and `/`), with underscores. In the third string, `makeValidName` appended a prefix because the string doesn't begin with a letter, deleted the empty space, and capitalized the character following the deleted space.

Construct Valid MATLAB Identifiers Using Specified Replacement Style

Replace invalid characters with the corresponding hexadecimal representation.

```
S = {'Item_#','Price/Unit','1st order','Contact'};
N = matlab.lang.makeValidName(S,'ReplacementStyle','hex')
```

```
N =

    'Item_0x23'    'Price0x2FUnit'    'x1stOrder'    'Contact'
```

In the first and second strings, `makeValidName` replaced the invalid characters (`#` and `/`), with their hexadecimal representation. In the third string, `makeValidName` appended a prefix because the string doesn't begin with a letter, deleted the empty space, and capitalized the character following the deleted space.

Delete invalid characters.

```
N = matlab.lang.makeValidName(S,'ReplacementStyle','delete')
```

```
N =

    'Item_'    'PriceUnit'    'x1stOrder'    'Contact'
```

`makeValidName` deleted the invalid characters (`#` and `/`). In the third string, `makeValidName` appended a prefix because the string doesn't begin with a letter, deleted the empty space, and capitalized the character following the deleted space.

Construct Valid MATLAB Identifiers Using Specified Prefix

```
S = {'1stMeasurement','2ndMeasurement','Control'};
```

```
N = matlab.lang.makeValidName(S,'Prefix','m_')

N =

    'm_1stMeasurement'    'm_2ndMeasurement'    'Control'
```

Only the strings that do not start with a letter are prepended with a prefix.

Determine Modified Strings

```
S = {'a%name', 'name_1', '2_name'};
[N, modified] = matlab.lang.makeValidName(S)

N =

    'a_name'    'name_1'    'x2_name'


modified =

     1     0     1
```

`makeValidName` did not modify the second element.

Input Arguments

S — Input strings

string or cell array of strings

Input strings, specified as a string or cell array of strings. If `S` is a cell array of strings, it must be a `1xN` or `Nx1` cell array.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ReplacementStyle','delete'` deletes invalid characters.

`'ReplacementStyle'` — Replacement style

`'underscore'` (default) | `'delete'` | `'hex'`

Replacement style, specified as a string. The value controls how MATLAB replaces nonalphanumeric characters.

`ReplacementStyle` Value	Description
`'underscore'` (default)	Replaces all characters that are not alphanumerics or underscores with underscores. `'underscore'` deletes whitespace characters and changes a lowercase letter following a whitespace to uppercase.
`'hex'`	Replaces each character that is not an alphanumeric or underscore with its corresponding hexadecimal representation. `'hex'` deletes whitespace characters and changes a lowercase letter following a whitespace to uppercase.
`'delete'`	Deletes all characters that are not alphanumerics or underscores. `'delete'` deletes whitespace characters and changes any lowercase letter following a whitespace to uppercase.

`'Prefix'` — Character to prefix

`'x'` (default) | string

Character to prefix to input strings that do not begin with a letter after `makeValidName` replaces nonalphanumeric characters, specified as a string. A valid prefix must start with a letter, contain only alphanumeric characters and underscores, and not be longer than the value of `namelengthmax`.

Output Arguments

`N` — Valid MATLAB identifiers

string | cell array of strings

Valid MATLAB identifiers, returned as a string or cell array of strings. The output has the same number of dimensions as the input, `S`.

`modified` — Indicator of modified strings

logical scalar | logical array

Indicator of modified strings, returned as a logical scalar or array and having the same number of dimensions as the input, `S`. A value of `1` (`true`) indicates that `makeValidName` modified the input string in the corresponding location. A value of `0` (`false`) indicates that `makeValidName` did not need to modify the input string in the corresponding location.

More About

Tips

- To ensure strings are valid and unique, use `matlab.lang.makeUniqueStrings` after `matlab.lang.makeValidName`.

```
S = {'my.Name','my_Name','my_Name'};
validStrings = matlab.lang.makeValidName(S)
validUniqueStrings = matlab.lang.makeUniqueStrings(validStrings,{},...
    namelengthmax)
```

```
validStrings =

    'my_Name'    'my_Name'    'my_Name'


validUniqueStrings =

    'my_Name'    'my_Name_1'    'my_Name_2'
```

- To customize an invalid character replacement, first use functions such as `strrep` or `regexprep` to convert to valid characters. For example, convert `'@'` characters in the string, `S`, to `'At'` using `strrep(S,'@','At')`. Then, use `matlab.lang.makeValidName` to ensure that all characters in the strings are valid.

See Also

`iskeyword` | `isletter` | `isvarname` | `matlab.lang.makeUniqueStrings` | `namelengthmax` | `regexp` | `regexprep` | `strrep` | `who`

Introduced in R2014a

matlabrc

Startup file for MATLAB program

Description

At startup time, MATLAB automatically executes the file `matlabrc.m`. This function establishes the MATLAB path, sets the default figure size, and sets some uicontrol defaults.

On multiuser or networked systems, system administrators can put messages, definitions, or other code that applies to all users in their `matlabrc.m` file.

The file `matlabrc.m` invokes the `startup.m` file, if it exists on the search path MATLAB uses.

Individual users should use the `startup.m` file to customize MATLAB startup. The `matlabrc.m` file, located in the *`matlabroot`*`/toolbox/local` folder, is reserved for system administrators.

Examples

Turning Off the Figure Window Toolbar

If you do not want the toolbar to appear in the figure window, remove the comment marks from the following line in the `matlabrc.m` file, or create a similar line in your own `startup.m` file.

```
%   set(0,'defaultfiguretoolbar','none')
```

More About

Tips

You can also start MATLAB using options you define at the Command Window prompt or in your Microsoft Windows shortcut for MATLAB.

Algorithms

MATLAB invokes `matlabrc` at startup. `matlabrc.m` contains the statements

```
if exist('startup') == 2
    startup
end
```

that invokes `startup.m`, if it exists. Extend this process to create additional startup files, if required.

- Startup Options

See Also

`matlabroot` | `quit` | `restoredefaultpath` | `startup`

matlabroot

Root folder

Syntax

```
matlabroot
mr = matlabroot
```

Description

`matlabroot` returns the name of the folder where the MATLAB software is installed. Use `matlabroot` to create a path to MATLAB and toolbox folders that does not depend on a specific platform, MATLAB version, or installation location.

`mr = matlabroot` returns the name of the folder in which the MATLAB software is installed and assigns it to `mr`.

Examples

Get the location where MATLAB is installed:

```
matlabroot
```

MATLAB returns:

```
C:\Program Files\MATLAB\R2009a
```

Produce a full path to the `toolbox/matlab/general` folder that is correct for the platform on which it is executed:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Change the current folder to the MATLAB root folder:

```
cd(matlabroot)
```

To add the folder `myfiles` to the MATLAB search path, run

```
addpath([matlabroot '/toolbox/local/myfiles'])
```

More About

Tips

matlabroot as Folder Name

The term *`matlabroot`* also refers to the folder where MATLAB files are installed. For example, “save to *`matlabroot`*`/toolbox/local`” means save to the `toolbox/local` folder in the MATLAB root folder.

Using $matlabroot as a Literal

In some files, `$matlabroot` is literal. In those files, MATLAB interprets `$matlabroot` as the full path to the MATLAB root folder. For example, including the line:

```
$matlabroot/toolbox/local/myfile.jar
```

in `javaclasspath.txt`, adds `myfile.jar`, which is located in the `toolbox/local` folder, to the static Java class path.

Sometimes, particularly in older code examples, the term `$matlabroot` or `$MATLABROOT` is not meant to be interpreted literally but is used to represent the value returned by the `matlabroot` function.

matlabroot on Macintosh Platforms

In R2008b (V7.7) and more recent versions, running `matlabroot` on Apple Macintosh platforms returns

```
/Applications/MATLAB_R2008b.app
```

In versions prior to R2008b (V7.7), such as R2008a (V7.6), running `matlabroot` on Macintosh platforms returns, for example

```
/Applications/MATLAB_R2008a
```

When you use GUIs on Macintosh platforms, you cannot directly view the contents of the MATLAB root folder. For more information, see “Navigating Within the MATLAB Root Folder on Macintosh Platforms”.

See Also

`fullfile` | `path` | `toolboxdir`

Introduced before R2006a

matlabshared.supportpkg.checkForUpdate

List of support packages that can be updated

Syntax

```
matlabshared.supportpkg.checkForUpdate
info = matlabshared.supportpkg.checkForUpdate
```

Description

`matlabshared.supportpkg.checkForUpdate` displays information about support package updates in the MATLAB Command Window. If an update is available, use `supportPackageInstaller` to install the updates.

`info = matlabshared.supportpkg.checkForUpdate` returns a structured array of information about installed support packages.

Examples

Check for support package updates

```
matlabshared.supportpkg.checkForUpdate
```

```
No support packages need updates.
```

If one or more updates are available, the command line displays that information in the response.

Get a structured array of support package updates

```
info = matlabshared.supportpkg.checkForUpdate
```

```
info =

                Name: 'Arduino'
    InstalledVersion: '3.0'
```

```
        BaseProduct: 'Simulink'
```

Output Arguments

`info` — Return argument from function
structure created using `matlabshared.supportpkg.checkForUpdate`

Information about support package updates, returned as a structured array.

See Also

`matlabshared.supportpkg.getInstalled` | `supportPackageInstaller` | `targetupdater`

matlabshared.supportpkg.getInstalled

List of installed support packages

Syntax

```
matlabshared.supportpkg.getInstalled
info = matlabshared.supportpkg.getInstalled
```

Description

`matlabshared.supportpkg.getInstalled` displays information about installed support packages in the MATLAB Command Window.

`info = matlabshared.supportpkg.getInstalled` returns a structured array of information about installed support packages.

Examples

Get a list of installed support packages

```
matlabshared.supportpkg.getInstalled
```

```
Name   Version Base Product
------- ------- ------------
Arduino 3.0     Simulink
```

Get a structured array of installed support packages

```
info = matlabshared.supportpkg.getInstalled
```

```
info =

               Name: 'Arduino'
    InstalledVersion: '3.0'
```

```
        BaseProduct: 'Simulink'
```

Output Arguments

`info` — Return argument from function
structure created using `matlabshared.supportpkg.getInstalled`

Information about installed support packages, returned as a structured array.

See Also

`matlabshared.supportpkg.checkForUpdate` | `supportPackageInstaller` | `targetupdater`

matlab (Linux)

Start MATLAB program from Linux system prompt

Syntax

```
matlab
matlab option1 ... optionN
```

Description

`matlab` is a Bourne shell script that starts the MATLAB program from a Linux system prompt. Here the term `matlab` refers to this script and MATLAB refers to the program.

`matlab option1 ... optionN` starts MATLAB with the specified startup options.

Alternatively, assign startup options in the MATLAB ".matlab7rc.sh Startup File" on page 1-4972. Modifying the `.matlab7rc.sh` file defines startup options every time you start MATLAB.

Input Arguments

`option1 ... optionN` — One or more startup options

strings

One or more startup options, specified as strings corresponding to valid startup options from the following tables.

Mode Options

Option	Result
`-desktop`	Start MATLAB without a controlling terminal. Use this option when you start MATLAB from a window manager menu or desktop icon.

Option	Result
`-nodesktop`	Run the JVM software without opening the MATLAB desktop. You can use development environment tools by calling them as functions. Use this option to run in batch processing mode. If you pipe to MATLAB using the > constructor, the `nodesktop` option is used automatically. Do not use `nodesktop` to provide a Command-Window-only interface. Instead, in the **Environment** section on the **Home** tab, click **Layout**. Then, under **Select Layout**, select **Command Window Only**.
`-nojvm`	Start MATLAB without the JVM software. Features that require Java software (such as the desktop tools and graphics) are not supported.

Display Options

Option	Result
`-noFigureWindows`	Disable the display of figure windows in MATLAB.
`-nosplash`	Do not display the splash screen during startup.
`-nodisplay`	Start the JVM software without starting the MATLAB desktop. This option does not display X commands. It overrides the `DISPLAY` environment variable.
`-display xDisp`	Send X commands to X Window Server display `xDisp`. This option overrides the `DISPLAY` environment variable.

Execute MATLAB Script or Function

Option	Result
`-r` *`statement`*	Execute the specified MATLAB *`statement`*, specified as a string or as the name of a MATLAB script or function. If

Option	Result
	statement is MATLAB code, enclose the string with double quotation marks. If *statement* is the name of a MATLAB function or script, do not specify the file extension and do not use quotation marks. Any required file must be on the MATLAB search path or in the startup folder. Example: `-r "disp(['Current folder: ' pwd])"` Example: `-r myscript`

Specify MATLAB Version

Option	Result
`v=`*variant*	Start the version of MATLAB in the `bin/`*variant* folder.

Debugging Options

Option	Result
`-logfile` *filename*	Copy Command Window output, including error reports, into *filename*, specified as a string. Example: `-logfile output.log`
`-n`	Display, without starting MATLAB, final values of the environment variables and arguments passed to the MATLAB executable. This option also displays other diagnostic information for use when working with a Technical Support Representative.
`-e`	Display, without starting MATLAB, all environment variables and their values to standard output. If the exit status is not `0` on return, then the variables and values might not be correct.
`-D`*debugger debugopts*	Start MATLAB in debug mode. This option uses the debugger program name, *debugger*, specified as a string, for example,

Option	Result
	`gdb`, `lldb`, or `dbx`. You can specify the full path to the debugger. This option must be the first option in the `matlab` command. Debugger program command-line options, *`debugopts`*, specified as a string of valid options for *`debugger`*. See your debugger documentation for details. Do not use any other `matlab` command options when using *`debugopts`*. Do not add a space between **D** and *`debugger`*. Example: `-Dgdb`
`-jdb` *`portnumber`*	Enable use of the Java debugger. The Java debugger uses the default *`portnumber`* value 4444 to communicate with MATLAB. The port number is optional. However, to use the Java debugger while running multiple MATLAB sessions, you must specify a port number. The *`portnumber`* value must be an integer in the range 0–65535. The integer cannot be reserved or currently in use by another application on your system.
`-debug`	Display information for debugging X-based problems. Use this option only when working with a Technical Support Representative from MathWorks, Inc.

Use Single Computational Thread

By default, MATLAB uses the multithreading capabilities of the computer on which it is running.

Option	Result
`-singleCompThread`	Limit MATLAB to a single computational thread.

Disable Searching Custom Java Class Path

Option	Result
`-nouserjavapath`	Disable use of `javaclasspath.txt` and `javalibrarypath.txt` files. For more information, see "java.opts Files".

OpenGL Library Options

These options control the use of software OpenGL libraries when MATLAB detects a graphics driver with known issues. For more information, see "Features with OpenGL Requirements".

Option	Result
`-softwareopengl`	Force MATLAB to start with software OpenGL libraries.
`-nosoftwareopengl`	Disable auto-selection of OpenGL software.

Specify License File

Option	Result
`-c` *`license`*	Use the specified license file, *`license`*, specified as a string, a colon-separated list of license file names, or a `port@host` entry. For more information, see Understanding License Files in "License Management".

Help Options

Option	Result
`-h`	Display startup options without starting MATLAB.
`-help`	Same as `-h` option.

Examples

Start MATLAB Without Desktop or JVM

```
matlab -nojvm -nodisplay -nosplash
```

Display Current Folder at Startup

```
matlab -r "disp(['Current folder: ' pwd])"
```

Definitions

.matlab7rc.sh Startup File

The `.matlab7rc.sh` shell script contains variable definitions used by the `matlab` script.

Use the `.matlab7rc.sh` file to redefine variables defined in the `matlab` script. `matlab` looks for the first occurrence of `.matlab7rc.sh` in the:

- Current folder
- Home folder (`$HOME`)
- *`matlabroot`*`/bin` folder

To edit the `.matlab7rc.sh` file, use the template located in the *`matlabroot`*`/bin` folder.

This table lists the variables. For more information, see the comments in the `.matlab7rc.sh` file.

Variable	Definition and Standard Assignment Behavior
ARCH	The machine architecture. MATLAB checks for a valid architecture in this order: • The value ARCH passed with the `-arch` or `-arch/ext` argument to the script. • The value of the environment variable MATLAB_ARCH.

Variable	Definition and Standard Assignment Behavior
DISPLAY	The host name of the X Window display MATLAB uses for output. The value of Xdisplay passed with the -display argument to the script is used; otherwise, the value in the environment is used. MATLAB ignores DISPLAY if the -nodisplay argument is passed.
LD_LIBRARY_PATH	Final Load library path. The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in .matlab7rc.sh as LDPATH_PREFIX. The second sublist is computed in the script and includes folders inside the MATLAB root folder and relevant Java folders. The third sublist contains any nonempty value of LD_LIBRARY_PATH from the environment possibly augmented in .matlab7rc.sh. The final sublist is defined in .matlab7rc.sh as LDPATH_SUFFIX.
MATLAB	The MATLAB root folder. MATLAB uses the default computed by the script unless MATLABdefault is reset in .matlab7rc.sh. Currently MATLABdefault is not reset in the shipping .matlab7rc.sh.
MATLABPATH	The MATLAB search path. The final value is a colon-separated list with the MATLABPATH from the environment prepended to a list of computed defaults. At startup, you can add subfolders of userpath to the MATLAB search path. See userpath for details.

Variable	Definition and Standard Assignment Behavior
SHELL	The shell to use when the "!" or unix command is issued in MATLAB. This value is taken from the environment, unless SHELL is reset in .matlab7rc.sh. Another environment variable called MATLAB_SHELL takes precedence over SHELL. MATLAB checks internally for MATLAB_SHELL first and, if empty or not defined, it checks SHELL. If SHELL is also empty or not defined, MATLAB uses /bin/sh. Use an absolute path for the value of MATLAB_SHELL, that is, /bin/sh, not simply sh. Currently, the shipping .matlab7rc.sh file does not reset SHELL. Also, this file does not reference or set MATLAB_SHELL.
TOOLBOX	Path of the toolbox folder. A nonempty value in the environment is used first. Otherwise, *matlabroot*/toolbox, computed by the script, is used unless TOOLBOX is reset in .matlab7rc.sh. Currently TOOLBOX is not reset in the shipping .matlab7rc.sh.

The matlab script determines the path of the MATLAB root folder by looking up the folder tree from the *matlabroot*/bin folder (where the matlab script is located). MATLAB use the MATLAB variable to locate all files within the MATLAB folder tree.

You can change the definition of MATLAB. For example, change the definition if you want to run a different version of MATLAB. Or change the definition if your system uses certain types of automounting schemes and the path determined by the matlab script is not correct.

See Also

matlab (Mac) | matlab (Windows)

Related Examples

- "Start MATLAB on Linux Platforms"

More About

- “Startup Options”

matlab (Mac)

Start MATLAB program from Mac Terminal

Syntax

```
matlab
matlab option1 ... optionN
```

Description

`matlab` is a Bourne shell script that starts the MATLAB program from a Mac system prompt. Here the term `matlab` refers to this script and MATLAB refers to the program.

The `matlab` script is located in the MATLAB application package, `/Applications/MATLAB_`*`release`*`.app/bin/matlab`, where *`release`* is the MATLAB release number.

The `matlab` script:

- Determines the MATLAB root folder, the value returned by the `matlabroot` function.
- Processes command-line options, if any.
- Reads the MATLAB startup file, `.matlab7rc.sh`.
- Sets MATLAB environment variables.

`matlab option1 ... optionN` starts MATLAB with the specified startup options.

Alternatively, assign startup options in the MATLAB ".matlab7rc.sh Startup File" on page 1-4981. Modifying the `.matlab7rc.sh` file defines startup options every time you start MATLAB.

Input Arguments

`option1 ... optionN` — One or more startup options
strings

One or more startup options, specified as strings corresponding to valid startup options from the following tables.

Mode Options

Option	Result
`-desktop`	Start MATLAB without a controlling terminal. Use this option when you start MATLAB from a window manager menu or desktop icon.
`-nodesktop`	Run the JVM software without opening the MATLAB desktop. You can use development environment tools by calling them as functions. Use this option to run in batch processing mode. If you pipe to MATLAB using the > constructor, the `nodesktop` option is used automatically. Do not use `nodesktop` to provide a Command-Window-only interface. Instead, in the **Environment** section, on the **Home** tab, click **Layout**. Then, under **Select Layout**, select **Command Window Only**.
`-nojvm`	Start MATLAB without the JVM software. Features that require Java software (such as the desktop tools and graphics) are not supported.

Display Options

Option	Result
`-noFigureWindows`	Disable the display of figure windows in MATLAB.
`-nosplash`	Do not display the splash screen during startup.
`-nodisplay`	Start the JVM in headless mode, and MATLAB from displaying all windows.

Execute MATLAB Script or Function

Option	Result
`-r statement`	Execute the specified MATLAB *`statement`*, specified as a string or as the name of a MATLAB script or function. If *`statement`* is MATLAB code, enclose the string with double quotation marks. If *`statement`* is the name of a MATLAB function or script, do not specify the file extension and do not use quotation marks. Any required file must be on the MATLAB search path or in the startup folder. Example: `-r "disp(['Current folder: ' pwd])"` Example: `-r myscript`

Specify MATLAB Version

Option	Result
`v=variant`	Start the version of MATLAB in the `bin/variant` folder.

Debugging Options

Option	Result
`-logfile filename`	Copy Command Window output, including error reports, into *`filename`*, specified as a string. Example: `-logfile output.log`
`-n`	Display, without starting MATLAB, final values of the environment variables and arguments passed to the MATLAB executable. This option also displays other diagnostic information for use when working with a Technical Support Representative.

Option	Result
`-e`	Display, without starting MATLAB, all environment variables and their values to standard output. If the exit status is not `0` on return, then the variables and values might not be correct.
`-D`*`debugger debugopts`*	Start MATLAB in debug mode. This option uses the debugger program name, *`debugger`*, specified as a string, for example, `gdb`, `lldb`, or `dbx`. You can specify the full path to the debugger. This option must be the first option in the `matlab` command. Debugger program command-line options, *`debugopts`*, specified as a string of valid options for *`debugger`*. See your debugger documentation for details. Do not use any other `matlab` command options when using *`debugopts`*. Do not add a space between **D** and *`debugger`*. Example: `-Dlldb`
`-jdb` *`portnumber`*	Enable use of the Java debugger. The Java debugger uses the default *`portnumber`* value 4444 to communicate with MATLAB. The port number is optional. However, to use the Java debugger while running multiple MATLAB sessions, you must specify a port number. The *`portnumber`* value must be an integer in the range 0–65535. The integer cannot be reserved or currently in use by another application on your system.

Use Single Computational Thread

By default, MATLAB uses the multithreading capabilities of the computer on which it is running.

Option	Result
`-singleCompThread`	Limit MATLAB to a single computational thread.

Disable Searching Custom Java Class Path

Option	Result
`-nouserjavapath`	Disable use of `javaclasspath.txt` and `javalibrarypath.txt` files. For more information, see "java.opts Files".

Specify License File

Option	Result
`-c` *`license`*	Use the specified license file, *`license`*, specified as a string, a colon-separated list of license file names, or a `port@host` entry. For more information, see Understanding License Files in "License Management".

Help Options

Option	Result
`-h`	Display startup options without starting MATLAB.
`-help`	Same as `-h` option.

Examples

Start MATLAB from Applications Folder

Move to the `bin` folder within the application package. For example:

```
cd /Applications/MATLAB_R2015a.app/bin
```

Preface the `matlab` command with `./` characters.

```
./matlab -nosplash
```

Start MATLAB from Any Folder

Start MATLAB from any Terminal folder by specifying the full path name.

```
/Applications/MATLAB_release.app/bin/matlab/matlab
```

where *`release`* is the MATLAB release number.

Startup MATLAB Without Desktop or JVM

```
matlab -nojvm -nodisplay -nosplash
```

Display Current Folder at Startup

```
matlab -r "disp(['Current folder: ' pwd])"
```

Definitions

.matlab7rc.sh Startup File

The `.matlab7rc.sh` shell script contains variable definitions used by the `matlab` script. Use the `.matlab7rc.sh` file to redefine variables defined in the `matlab` script.

`matlab` looks for the first occurrence of `.matlab7rc.sh` in the:

- Current folder
- Home folder (`$HOME`)
- *`matlabroot`*`/bin` folder

To edit the `.matlab7rc.sh` file, use the template located in the *`matlabroot`*`/bin` folder.

The following table lists the variables. For more information, see the comments in the `.matlab7rc.sh` file.

`.matlab7rc.sh` Variable	Definition and Standard Assignment Behavior
ARCH	The machine architecture. MATLAB checks for a valid architecture in this order: • The value ARCH passed with the `-arch` or `-arch/ext` argument to the script. • The value of the environment variable MATLAB_ARCH.

.matlab7rc.sh Variable	Definition and Standard Assignment Behavior
DYLD_LIBRARY_PATH	Final Load library path. The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in .matlab7rc.sh as LDPATH_PREFIX. The second sublist is computed in the script and includes folders inside the MATLAB root folder and relevant Java folders. The third sublist contains any nonempty value of DYLD_LIBRARY_PATH from the environment possibly augmented in .matlab7rc.sh. The final sublist is defined in .matlab7rc.sh as LDPATH_SUFFIX.
MATLAB	The MATLAB root folder. MATLAB uses the default computed by the script unless MATLABdefault is reset in .matlab7rc.sh. Currently MATLABdefault is not reset in the shipping .matlab7rc.sh.
MATLABPATH	The MATLAB search path. The final value is a colon-separated list with the MATLABPATH from the environment prepended to a list of computed defaults. At startup, you can add subfolders of userpath to the MATLAB search path. See userpath for details.

.matlab7rc.sh Variable	Definition and Standard Assignment Behavior
SHELL	The shell to use when the "!" or unix command is issued in MATLAB. This value is taken from the environment, unless SHELL is reset in .matlab7rc.sh. Another environment variable called MATLAB_SHELL takes precedence over SHELL. MATLAB checks internally for MATLAB_SHELL first and, if empty or not defined, it checks SHELL. If SHELL is also empty or not defined, MATLAB uses /bin/sh. Use an absolute path for the value of MATLAB_SHELL, that is, /bin/sh, not simply sh. Currently, the shipping .matlab7rc.sh file does not reset SHELL. Also, this file does not reference or set MATLAB_SHELL.
TOOLBOX	Path of the toolbox folder. A nonempty value in the environment is used first. Otherwise, *matlabroot*/toolbox, computed by the script, is used unless TOOLBOX is reset in .matlab7rc.sh. Currently TOOLBOX is not reset in the shipping .matlab7rc.sh.

The matlab script determines the path of the MATLAB root folder by looking up the folder tree from the *matlabroot*/bin folder (where the matlab script is located). MATLAB use the MATLAB variable to locate all files within the MATLAB folder tree.

You can change the definition of MATLAB. For example, change the definition if you want to run a different version of MATLAB. Or change the definition if your system uses certain types of automounting schemes and the path determined by the matlab script is not correct.

See Also

matlab (Windows) | matlab (Linux) | matlabroot

Related Examples

- "Start MATLAB on Mac Platforms"

More About

- "Startup Options"

matlab (Windows)

Start MATLAB program from Windows system prompt

Syntax

```
matlab
matlab option1 ... optionN
```

Description

`matlab` starts the MATLAB program from the Microsoft Windows system prompt. Here the term `matlab` refers to the command you type, and MATLAB refers to the program).

The `matlab` command:

- Determines the MATLAB root folder, the value returned by the `matlabroot` function.
- Determines the host machine architecture, 32-bit or 64-bit.
- Processes command-line options and passes other options to MATLAB.

`matlab option1 ... optionN` starts MATLAB with the specified startup options.

Input Arguments

`option1 ... optionN` — One or more startup options

strings

One or more startup options, specified as strings corresponding to valid startup options from the following tables.

Display Options

Option	Result
`-noFigureWindows`	Disable the display of figure windows in MATLAB.

Option	Result
`-nosplash`	Do not display the splash screen during startup.

Execute MATLAB Script or Function

Option	Result
`-r` *`statement`*	Execute the MATLAB *`statement`*, specified as a string or as the name of a MATLAB script or function. If *`statement`* is MATLAB code, enclose the string with double quotation marks. If *`statement`* is the name of a MATLAB function or script, do not specify the file extension and do not use quotation marks. Any required file must be on the MATLAB search path or in the startup folder. Example: `-r "disp(['Current folder: ' pwd])"` Example: `-r myscript`

Run 32-Bit MATLAB on 64-Bit Platforms

Option	Result
`-win32`	Run 32-bit MATLAB on 64-bit processors.

Debugging Options

Option	Result
`-logfile` *`filename`*	Copy Command Window output, including error log reports, in to *`filename`*, specified as a string. Example: `-logfile output.log`
`-jdb` *`portnumber`*	Enable use of the Java debugger. The Java debugger uses the default *`portnumber`* value 4444 to communicate with MATLAB.

Option	Result
	The port number is optional. However, to use the Java debugger while running multiple MATLAB sessions, you must provide a port number. The *`portnumber`* value must be an integer in the range 0–65535. The integer cannot be reserved or currently in use by another application on your system.

Use Single Computational Thread

By default, MATLAB uses the multithreading capabilities of the computer on which it is running.

Option	Result
`-singleCompThread`	Limit MATLAB to a single computational thread.

Disable Searching Custom Java Class Path

Option	Result
`-nouserjavapath`	Disable use of `javaclasspath.txt` and `javalibrarypath.txt` files. For more information, see "java.opts Files".

OpenGL Library Options

These options control the use of software OpenGL libraries when MATLAB detects a graphics driver with known issues. For more information, see "Features with OpenGL Requirements".

Option	Result
`-softwareopengl`	Force MATLAB to start with software OpenGL libraries.
`-nosoftwareopengl`	Disable auto-selection of OpenGL software.

COM Automation Server Options

Option	Result
`-automation`	Start MATLAB as a Component Object Model (COM) Automation server. MATLAB does not display the splash screen and minimizes the window. Use for a single call to MATLAB.
`-regserver`	Register MATLAB as a COM Automation server in the Windows registry. MATLAB displays a minimized command window; close this window. You must have administrator privileges to change the Windows registry. Based on your User Account Control (UAC) settings, you might need to right-click a Windows Command Processor and select **Run as administrator**. If that option is not available, contact your system administrator. MATLAB remains registered until you use the `-unregserver` option. Alternatively, you can register MATLAB from the MATLAB command prompt. Type: `!matlab -regserver` MATLAB displays a minimized command window. Open this window and exit MATLAB to continue working with MATLAB.
`-unregserver`	Remove MATLAB COM server entries from the registry.

Wait for MATLAB to Terminate

By default, when you call the `matlab` command from a script, the command starts MATLAB and then immediately executes the next statements in the script. The `-wait` option pauses the script until MATLAB terminates.

Option	Result
`-wait`	Use in a startup script to process the results from MATLAB. Calling MATLAB with this option blocks the script from continuing until the results are generated.

Shield Options

For 32-bit Windows platforms only.

Shield options provide the specified level of protection of the address space used by MATLAB during startup. These options attempt to provide the largest contiguous block of memory available after startup, which is useful for processing large data sets. These options load resources, such as DLLs, into locations without fragmenting the address space. For shield option values other than `none`, address space is protected up to or after the processing of `matlabrc`. Use higher levels of protection to secure larger initial blocks of contiguous memory. However, a higher level might not always provide a larger size block and might cause startup problems. Therefore, start with a lower level of protection, and if successful, try the next higher level. Use the MATLAB `memory` function after startup to see the size of the largest contiguous block of memory. This function helps you determine the actual effect of the setting you used. If your `matlabrc.m` or `startup.m` file requires significant memory, a higher level of protection might cause startup to fail. In that event try a lower level.

Enter one of the options shown in the following table.

Option	Description
`-shield minimum`	Default setting. Protect the range `0x50000000`–`0x70000000` until startup processes `matlabrc`. This option ensures that there is at least approximately 500 MB of contiguous memory up to this point. Start with this option. If MATLAB fails to start successfully, use the `-shield none` option instead. If MATLAB starts successfully and you want to try to ensure an even larger contiguous block after startup, try using the `-shield medium` option.

Option	Description
`-shield medium`	Protect the same range as `-shield minimum` until after startup processes `matlabrc`. This option ensures that there is at least approximately 500 MB of contiguous memory at this point. If MATLAB fails to start successfully with this option, use `-shield minimum` instead. If MATLAB starts successfully and you want to try to ensure an even larger contiguous block after startup, try using the `-shield maximum` option.
`-shield maximum`	Protect the maximum range, which can be up to approximately 1.5 GB, until startup processes `matlabrc`. If MATLAB fails to start successfully, use the `-shield medium` option instead.
`-shield none`	Disable address shielding. Use this option if MATLAB fails to start successfully with the `-shield minimum` option.

Specify License File

For more information, see Understanding License Files in "License Management".

Option	Result
`-c` *`license`*	Use the License File, *`license`*, specified as a string, a colon-separated list of license file names, or a `port@host` entry. If specifying multiple files, separate the names by semicolons and enclose the entire list in quotation marks. If the path to your license file contains a space, enclose the path name in quotation marks. Example: `-c "c:\TMW license \license_agreement.txt"`

Help Options

Option	Result
`-h`	Display startup options without starting MATLAB.
`-help`	Same as `-h` option.
`-?`	Same as `-h` option.

Examples

Startup Without Splash Screen

```
matlab -nosplash
```

Copy Command Window Output into output.log File

```
matlab -logfile output.log
```

See Also

`matlab (Linux)` | `matlab (Mac)` | `matlabroot` | `memory` | `userpath`

More About

- "Start MATLAB on Windows Platforms"
- "Startup Options"

max

Largest elements in array

Syntax

```
M = max(A)
M = max(A,[],dim)
[M,I] = max( ___ )

C = max(A,B)

___ = max( ___ ,nanflag)
```

Description

`M = max(A)` returns the largest elements of `A`.

- If `A` is a vector, then `max(A)` returns the largest element of `A`.
- If `A` is a matrix, then `max(A)` is a row vector containing the maximum value of each column.
- If `A` is a multidimensional array, then `max(A)` operates along the first array dimension whose size does not equal `1`, treating the elements as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same. If `A` is an empty array with first dimension `0`, then `max(A)` returns an empty array of the same size as `A`.

`M = max(A,[],dim)` returns the largest elements along dimension `dim`. For example, if `A` is a matrix, then `max(A,[],2)` is a column vector containing the maximum value of each row.

`[M,I] = max( ___ )` finds the indices of the maximum values of `A` and returns them in output vector `I`, using any of the input arguments in the previous syntaxes. If the maximum value occurs more than once, then `max` returns the index corresponding to the first occurrence.

`C = max(A,B)` returns an array the same size as `A` and `B` with the largest elements taken from `A` or `B`. Either the dimensions of `A` and `B` are the same, or one can be a scalar.

`___ = max(___,nanflag)` specifies whether to include or omit `NaN` values in the calculation for any of the previous syntaxes. For the single input case, to specify `nanflag` without specifying `dim`, use `max(A,[],nanflag)`. For example, `max(A,[],'includenan')` includes all `NaN` values in `A` while `max(A,[],'omitnan')` ignores them.

Examples

Largest Vector Element

Create a vector and compute its largest element.

```
A = [23 42 37 18 52];
M = max(A)
```

```
M =

    52
```

Largest Complex Element

Create a complex vector and compute its largest element, that is, the element with the largest magnitude.

```
A = [-2+2i 4+i -1-3i];
max(A)
```

```
ans =

   4.0000 + 1.0000i
```

Largest Element in Each Matrix Column

Create a matrix and compute the largest element in each column.

```
A = [2 8 4; 7 3 9]
```

```
A =
```

```
     2     8     4
     7     3     9

M = max(A)

M =

     7     8     9
```

Largest Element in Each Matrix Row

Create a matrix and compute the largest element in each row.

```
A = [1.7 1.2 1.5; 1.3 1.6 1.99]

A =

    1.7000    1.2000    1.5000
    1.3000    1.6000    1.9900

M = max(A,[],2)

M =

    1.7000
    1.9900
```

Largest Element Indices

Create a matrix A and compute the largest elements in each column, as well as the row indices of A in which they appear.

```
A = [1 9 -2; 8 4 -5]

A =

     1     9    -2
     8     4    -5
```

```
[M,I] = max(A)

M =

     8     9    -2

I =

     2     1     1
```

Largest Element Comparison

Create a matrix and return the largest value between each of its elements compared to a scalar.

```
A = [1 7 3; 6 2 9]

A =

     1     7     3
     6     2     9

B = 5;
C = max(A,B)

C =

     5     7     5
     6     5     9
```

Largest Element in Matrix

Create a matrix `A` and use its column representation, `A(:)`, to find the value and index of the largest element.

```
A = [8 2 4; 7 3 9]

A =
```

```
     8     2     4
     7     3     9

A(:)

ans =

     8
     7
     2
     3
     4
     9

[M,I] = max(A(:))

M =

     9

I =

     6
```

`I` is the index of `A(:)` containing the largest element.

Now, use the `ind2sub` function to extract the row and column indices of `A` corresponding to the largest element.

```
[I_row, I_col] = ind2sub(size(A),I)

I_row =

     2

I_col =
```

```
     3
```

If you need only the maximum value of `A` and not its index, then call the `max` function twice.

```
M = max(max(A))
```

```
M =

     9
```

Largest Element Involving `NaN`

Create a vector and compute its maximum, excluding `NaN` values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
M = max(A,[],'omitnan')
```

```
M =

    3.9800
```

`max(A)` will also produce this result since `'omitnan'` is the default option.

Use the `'includenan'` flag to return `NaN`.

```
M = max(A,[],'includenan')
```

```
M =

   NaN
```

Input Arguments

`A` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

- If `A` is complex, then `max(A)` returns the complex number with the largest magnitude. If magnitudes are equal, then `max(A)` returns the value with the largest magnitude and the largest phase angle.
- If `A` is a scalar, then `max(A)` returns `A`.
- If `A` is a 0-by-0 empty array, then `max(A)` is as well.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

`dim` — Dimension to operate along

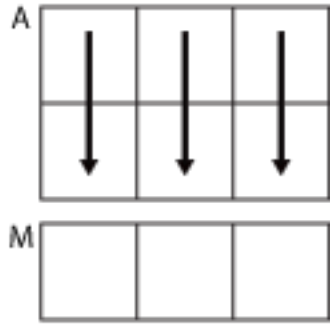
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

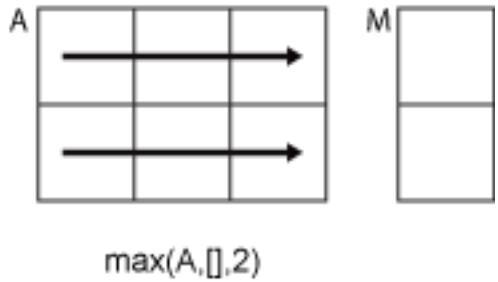
Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `max` returns an empty array with the same dimension sizes.

Consider a two-dimensional input array, `A`:

- If `dim = 1`, then `max(A,[],1)` returns a row vector containing the largest element in each column.



- If `dim = 2`, then `max(A,[],2)` returns a column vector containing the largest element in each row.



`max` returns `A` if `dim` is greater than `ndims(A)`.

`B` — Additional input array

scalar | vector | matrix | multidimensional array

Additional input array, specified as a scalar, vector, matrix, or multidimensional array.

- The dimensions of `A` and `B` must match, or one can be a scalar.
- `A` and `B` can be the same data type or one can be `double` with the other `single`, `duration`, or any integer data type.
- If `A` and `B` are ordinal categorical arrays, they must have the same sets of categories with the same order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

`nanflag` — `NaN` condition

'`omitnan`' (default) | '`includenan`'

`NaN` condition, specified as one of these values:

- '`omitnan`' — Ignore all `NaN` values in the input.
- '`includenan`' — Include the `NaN` values in the input for the calculation.

The `max` function does not support the `nanflag` option for `datetime`, `duration`, or `categorical` arrays.

Data Types: `char`

Output Arguments

`M` — Maximum values

scalar | vector | matrix | multidimensional array

Maximum values, returned as a scalar, vector, matrix, or multidimensional array. `size(M,dim)` is `1`, while the sizes of all other dimensions match the size of the corresponding dimension in `A`, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `M` is an empty array with the same dimension sizes as `A`.

`I` — Index to maximum values of `A`

scalar | vector | matrix | multidimensional array

Index to maximum values of `A`, returned as a scalar, vector, matrix, or multidimensional array. `I` is the same size as `M`. If the largest element occurs more than once, then `I` contains the index to the first occurrence of the value.

`C` — Maximum elements from `A` or `B`

scalar | vector | matrix | multidimensional array

Maximum elements from `A` or `B`, returned as a scalar, vector, matrix, or multidimensional array.

The size of `C` depends on the sizes of `A` and `B`:

- If `A` and `B` are arrays of the same size, then the size of `C` matches the size of `A` and `B`.
- If either `A` or `B` is a scalar, then the size of `C` matches the size of the nonscalar input array.
- If either `A` or `B` is an empty array with the other a scalar, then `C` is an empty array.

The data type of `C` depends on the data types of `A` and `B`:

- If `A` and `B` are the same data type, then `C` matches the data type of `A` and `B`.
- If either `A` or `B` is `single`, then `C` is `single`.
- If either `A` or `B` is an integer data type with the other a scalar `double`, then `C` assumes the integer data type.

More About

- “Matrix Indexing”

See Also

mean | median | min | sort

MaximizeCommandWindow

Open Automation server window

Syntax

IDL Method Signature

```
HRESULT MaximizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MaximizeCommandWindow
```

MATLAB Client

```
MaximizeCommandWindow(h)
```

Description

`MaximizeCommandWindow(h)` displays the window for the server attached to handle `h`, and makes it the currently active window on the desktop.

`MaximizeCommandWindow` restores the window to the size it had at the time it was minimized, not to the maximum size on the desktop. If the server window was not previously in a minimized state, `MaximizeCommandWindow` does nothing.

Examples

From a Visual Basic .NET client, modify the size of the command window in a MATLAB Automation server.

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
```

```
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

MinimizeCommandWindow

Introduced before R2006a

maxNumCompThreads

Control maximum number of computational threads

Note: `maxNumCompThreads` will be removed in a future version. You can set the `-singleCompThread` option when starting MATLAB to limit MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running.

Syntax

```
N = maxNumCompThreads
LASTN = maxNumCompThreads(N)
LASTN = maxNumCompThreads('automatic')
```

Description

`N = maxNumCompThreads` returns the current maximum number of computational threads `N`.

`LASTN = maxNumCompThreads(N)` sets the maximum number of computational threads to `N`, and returns the previous maximum number of computational threads, `LASTN`.

`LASTN = maxNumCompThreads('automatic')` sets the maximum number of computational threads using what the MATLAB software determines to be the most desirable. It additionally returns the previous maximum number of computational threads, `LASTN`.

Currently, the maximum number of computational threads is equal to the number of computational cores on your machine.

Note Setting the maximum number of computational threads using `maxNumCompThreads` does not propagate to your next MATLAB session.

mean

Average or mean value of array

Syntax

```
M = mean(A)
M = mean(A,dim)
M = mean( ___ ,outtype)
M = mean( ___ ,nanflag)
```

Description

`M = mean(A)` returns the mean of the elements of `A` along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `mean(A)` returns the mean of the elements.
- If `A` is a matrix, then `mean(A)` returns a row vector containing the mean of each column.
- If `A` is a multidimensional array, then `mean(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes `1` while the sizes of all other dimensions remain the same.

`M = mean(A,dim)` returns the mean along dimension `dim`. For example, if `A` is a matrix, then `mean(A,2)` is a column vector containing the mean of each row.

`M = mean( ___ ,outtype)` returns the mean with a specified data type, using any of the input arguments in the previous syntaxes. `outtype` can be `'default'`, `'double'`, or `'native'`.

`M = mean( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `mean(A,'includenan')` includes all `NaN` values in the calculation while `mean(A,'omitnan')` ignores them.

Examples

Mean of Matrix Columns

Create a matrix and compute the mean of each column.

```
A = [0 1 1; 2 3 2; 1 3 2; 4 2 2]
```

```
A =

     0     1     1
     2     3     2
     1     3     2
     4     2     2
```

```
M = mean(A)
```

```
M =

    1.7500    2.2500    1.7500
```

Mean of Matrix Rows

Create a matrix and compute the mean of each row.

```
A = [0 1 1; 2 3 2]
```

```
A =

     0     1     1
     2     3     2
```

```
M = mean(A,2)
```

```
M =

    0.6667
    2.3333
```

Mean of 3-D Array

Create a 4-by-2-by-3 array of integers between 1 and 10 and compute the mean values along the second dimension.

```
A = gallery('integerdata',10,[4,2,3],1);
M = mean(A,2)
```

```
M(:,:,1) =

    9.5000
    6.5000
    9.5000
    6.0000


M(:,:,2) =

    1.5000
    4.0000
    7.5000
    7.5000


M(:,:,3) =

    7.0000
    2.5000
    4.0000
    5.5000
```

Mean of Single-Precision Array

Create a single-precision vector of ones and compute its single-precision mean.

```
A = single(ones(1e8,1));
M = mean(A,'native')
```

```
M =

     1
```

The expected mean, 1, cannot be represented in single-precision due to saturation. Use the `double` output type to calculate the expected mean of `A`.

```
M = mean(A,'double')
```

```
M =

     1
```

Mean Excluding `NaN`

Create a vector and compute its mean, excluding `NaN` values.

```
A = [1 0 0 1 NaN 1 NaN 0];
M = mean(A,'omitnan')
```

```
M =

    0.5000
```

If you do not specify `'omitnan'`, then `mean(A)` returns `NaN`.

Input Arguments

`A` — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If `A` is a scalar, then `mean(A)` returns `A`.
- If `A` is an empty 0-by-0 matrix, then `mean(A)` returns `NaN`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `datetime` | `duration`

`dim` — Dimension to operate along

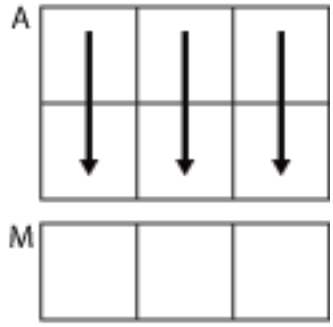
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `mean(A,1)` returns a row vector containing the mean of the elements in each column.



mean(A,1)

- If `dim = 2`, then `mean(A,2)` returns a column vector containing the mean of the elements in each row.

mean(A,2)

`mean` returns `A` when `dim` is greater than `ndims(A)` or when `size(A,dim)` is `1`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`outtype` — Output data type

`'default'` (default) | `'double'` | `'native'`

Output data type, specified as `'default'`, `'double'`, or `'native'`. These options also specify the data type in which the operation is performed.

`outtype`	Output data type
`'default'`	`double`, unless the input data type is `single`, `duration`, or `datetime`, in which case, the output is `'native'`
`'double'`	`double`, unless the data type is `duration` or `datetime`, in which case, `'double'` is not supported
`'native'`	same data type as the input, unless • Input data type is `logical`, in which case, the output is `double` • Input data type is `char`, in which case, `'native'` is not supported

Data Types: `char`

`nanflag` — `NaN` condition

'`includenan`' (default) | '`omitnan`'

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values when computing the mean, resulting in `NaN`.
- `'omitnan'` — Ignore all `NaN` values in the input.

The `mean` function does not support the `nanflag` option for `datetime` or `duration` arrays.

Data Types: `char`

More About

Mean

For a random variable vector A made up of N scalar observations, the mean is defined as

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i.$$

See Also

`mean` | `median` | `mode` | `std` | `sum` | `var`

median

Median value of array

Syntax

```
M = median(A)
M = median(A,dim)
M = median( ___ ,nanflag)
```

Description

`M = median(A)` returns the median value of `A`.

- If `A` is a vector, then `median(A)` returns the median value of `A`.
- If `A` is a nonempty matrix, then `median(A)` treats the columns of `A` as vectors and returns a row vector of median values.
- If `A` is an empty 0-by-0 matrix, `median(A)` returns `NaN`.
- If `A` is a multidimensional array, then `median(A)` treats the values along the first array dimension whose size does not equal `1` as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.

`median` computes natively in the numeric class of `A`, such that `class(M) = class(A)`.

`M = median(A,dim)` returns the median of elements along dimension `dim`. For example, if `A` is a matrix, then `median(A,2)` is a column vector containing the median value of each row.

`M = median( ___ ,nanflag)` optionally specifies whether to include or omit `NaN` values in the median calculation for any of the previous syntaxes. For example, `median(A,'omitnan')` ignores all `NaN` values in `A`.

Examples

Median of Matrix Columns

Define a 4-by-3 matrix.

```
A = [0 1 1; 2 3 2; 1 3 2; 4 2 2]
```

```
A =

     0     1     1
     2     3     2
     1     3     2
     4     2     2
```

Find the median value of each column.

```
M = median(A)
```

```
M =

    1.5000    2.5000    2.0000
```

For each column, the median value is the mean of the middle two numbers in sorted order.

Median of Matrix Rows

Define a 2-by-3 matrix.

```
A = [0 1 1; 2 3 2]
```

```
A =

     0     1     1
     2     3     2
```

Find the median value of each row.

```
M = median(A,2)
```

```
M =

     1
     2
```

For each row, the median value is the middle number in sorted order.

Median of 3-D Array

Create a 1-by-3-by-4 array of integers between 1 and 10.

```
A = gallery('integerdata',10,[1,3,4],1)
```

```
A(:,:,1) =

    10     8    10

A(:,:,2) =

     6     9     5

A(:,:,3) =

     9     6     1

A(:,:,4) =

     4     9     5
```

Find the median values of this 3-D array along the second dimension.

```
M = median(A)

M(:,:,1) =

    10

M(:,:,2) =

     6

M(:,:,3) =

     6

M(:,:,4) =

     5
```

This operation produces a 1-by-1-by-4 array by computing the median of the three values along the second dimension. The size of the second dimension is reduced to 1.

Compute the median along the first dimension of A.

```
M = median(A,1);
isequal(A,M)
```

```
ans =

     1
```

This command returns the same array as A because the size of the first dimension is 1.

Median of 8-Bit Integer Array

Define a 1-by-4 vector of 8-bit integers.

```
A = int8(1:4)
```

```
A =

    1    2    3    4
```

Compute the median value.

```
M = median(A),
class(M)
```

```
M =

    3


ans =

int8
```

M is the mean of the middle two numbers in sorted order returned as an 8-bit integer.

Median Excluding NaN

Create a vector and compute its median, excluding NaN values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
M = median(A,'omitnan')


M =
```

```
    0.2650
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. `A` can be a numeric array, ordinal categorical array, datetime array, or duration array.

dim — Dimension to operate along

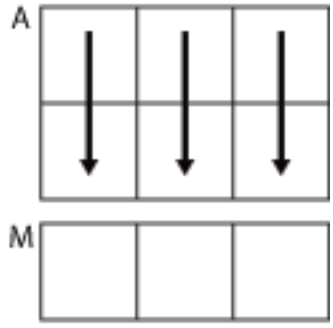
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

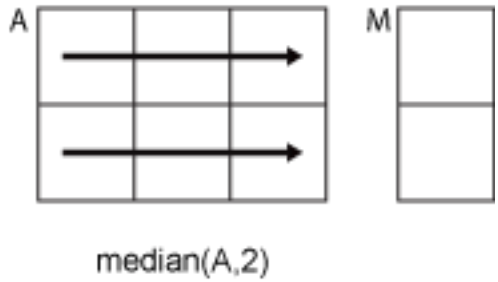
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `median(A,1)` returns a row vector containing the median of the elements in each column.



median(A,1)

- If `dim = 2`, then `median(A,2)` returns a column vector containing the median of the elements in each row.



`median` returns `A` when `dim` is greater than `ndims(A)`.

`nanflag` — `NaN` condition
'`includenan`' (default) | '`omitnan`'

`NaN` condition, specified as one of these values:

- '`includenan`' — the median of input containing `NaN` values is also `NaN`.
- '`omitnan`' — all `NaN` values appearing in the input are ignored. Note: the `NaN` flags are not set to `0`.

The `nanflag` option does not support datetime arrays, duration arrays, or categorical arrays.

Data Types: `char`

More About

Algorithms

For ordinal categorical arrays, MATLAB interprets the median of an even number of elements as follows:

If the number of categories between the middle two values is ...	Then the median is ...
zero (values are from consecutive categories)	larger of the two middle values

If the number of categories between the middle two values is ...	Then the median is ...
an odd number	value from category occurring midway between the two middle values
an even number	value from larger of the two categories occurring midway between the two middle values

See Also

`corrcoef` | `cov` | `max` | `mean` | `min` | `mode` | `std` | `var`

Introduced before R2006a

memmapfile

Create memory map to a file

Syntax

```
m = memmapfile(filename)
m = memmapfile(filename,Name,Value)
```

Description

`m = memmapfile(filename)` maps an existing file, `filename`, to memory and returns the memory map, `m`.

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of memory addresses within the MATLAB address space. Then, MATLAB can access files on disk in the same way it accesses dynamic memory, accelerating file reading and writing. Memory-mapping allows you to work with data in a file as if it were a MATLAB array.

`m = memmapfile(filename,Name,Value)` specifies the properties of `m` using one or more name-value pair arguments. For example, you can specify the format of the data in the file.

Examples

Map Entire File of `uint8` Data

At the command prompt, create a sample file in your current folder called `records.dat`, containing 10 `uint8` values.

```
myData = uint8(1:10)';

fileID = fopen('records.dat','w');
fwrite(fileID, myData,'uint8');
fclose(fileID);
```

Create a map for `records.dat`. When using `memmapfile`, the default data format is `uint8` so the file name is the only required input argument in this case.

```
m = memmapfile('records.dat')
```

```
m =

    Filename: 'd:\matlab\records.dat'
    Writable: false
      Offset: 0
      Format: 'uint8'
      Repeat: Inf
        Data: 10x1 uint8 array
```

MATLAB maps the entire `records.dat` file to memory, setting all properties of the memory map to their default values. The memory map is assigned to the variable, `m`. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

View the mapped data by accessing the `Data` property of `m`.

```
m.Data
```

```
ans =

    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
```

Map Entire File of Double-Precision Data

Create a memory map for double-precision data. The syntax is similar when specifying other data types.

At the command prompt, create a sample file in your current folder called `records.dat`, containing 10 `double` values.

```
myData = (1:10)';
```

```
fileID = fopen('records.dat','w');
fwrite(fileID,myData,'double');
fclose(fileID);
```

Create a memory map for `records.dat`, and set the `Format` property for the output to `'double'`.

```
m = memmapfile('records.dat','Format','double')
```

```
m =

    Filename: 'd:\matlab\records.dat'
    Writable: false
      Offset: 0
      Format: 'double'
      Repeat: Inf
        Data: 10x1 double array
```

The `Data` property contains the 10 double-precision values in `records.dat`.

Map and Change Part of a File

Create a memory map for a large array of `int32` data. Specify write access, and nondefault `Format` and `Offset` values.

At the command prompt, create a sample file in your current folder called `records.dat`, containing 10,000 `int32` values.

```
myData = int32([1:10000]);

fileID = fopen('records.dat','w');
fwrite(fileID,myData,'int32');
fclose(fileID);
```

Create a memory map for `records.dat`, and set the `Format` property for the output to `int32`. Also, set the `Offset` property to disregard the first 9000 bytes in the file, and the `Writable` property to permit write access.

```
m = memmapfile('records.dat',...
               'Offset',9000,...
               'Format','int32',...
               'Writable',true);
```

An `Offset` value of `9000` indicates that the first 9000 bytes of `records.dat` are not mapped.

Type the name of the memory map to see the current settings for all properties.

```
m
```

```
m =
    Filename: 'd:\matlab\records.dat'
    Writable: true
      Offset: 9000
      Format: 'int32'
      Repeat: Inf
        Data: 7750x1 int32 array
```

The `Format` property indicates that any read or write operation made via the memory map reads and writes the file contents as a sequence of signed 32-bit integers. The `Data` property contains only 7750 elements because the first 9000 bytes of `records.dat`, representing the first 2250 values in the file, are not mapped.

View the first five elements of the mapped data by accessing the `Data` property of `m`.

```
m.Data(1:5)
```

```
ans =

        2251
        2252
        2253
        2254
        2255
```

Map Region of File to Specific Array Shape

Create a memory map for a region of a file containing 100 double-precision values.

At the command prompt, create a sample file in your current folder called `mybinary.bin`, containing 100 double-precision values.

```
randData = gallery('uniformdata',[100,1],0,'double');

fileID = fopen('mybinary.bin','w');
fwrite(fileID,randData,'double');
fclose(fileID);
```

Map the first 75 values in `mybinary.bin` to a 5-by-5-by-3 array of double-precision values that can be referenced in the structure of the memory map using the field name `x`. Specify these parameters with the `Format` name-value pair argument.

```
m = memmapfile('mybinary.bin',...
               'Format',{'double',[5 5 3],'x'})
```

```
m =

    Filename: 'd:\matlab\mybinary.bin'
    Writable: false
      Offset: 0
      Format: {'double' [5 5 3] 'x'}
      Repeat: Inf
        Data: 1x1 struct array with fields:
                x
```

The `Data` property is a structure array that contains the mapped values in the field, `x`.

Assign the mapped data to a variable, `A`. Because the `Data` property is a structure array, you must index into the field, `x`, to access the data.

```
A = m.Data.x;
```

View information about `A`.

```
whos A
```

```
  Name      Size              Bytes  Class     Attributes

  A         5x5x3               600  double
```

Map Segments of File to Multiple Arrays

Map segments of a file with different array shapes and data types to memory.

At the command prompt, create a sample file in your current folder called `mybinary.bin`. Write `uint16` data and double-precision data representing sample pressure, temperature, and volume values into the file. In this case, each of the `uint16` arrays are 50-by-1 and the double-precision arrays are 5-by-10. `k` is a sample scaling factor.

```
k = 8.21;
[pres1,temp1] = gallery('integerdata',[1,300],[50,1],0,'uint16');
vol1 = double(reshape(k*temp1./pres1,5,10));
```

```
[pres2,temp2] = gallery('integerdata',[5,500],[50,1],5,'uint16');
vol2 = double(reshape(k*temp2./pres2,5,10));

fileID = fopen('mybinary.bin','w');
fwrite(fileID,pres1,'uint16');
fwrite(fileID,temp1,'uint16');
fwrite(fileID,vol1,'double');
fwrite(fileID,pres2,'uint16');
fwrite(fileID,temp2,'uint16');
fwrite(fileID,vol2,'double');
fclose(fileID);
```

Map the file to arrays accessible by unique names. Define a field, `pressure`, containing a 50-by-1 array of `uint16` values, followed by a field, `temperature`, containing 50-by-1 `uint16` values. Define a field, `volume`, containing a 5-by-10 array of double-precision values. Use a cell array to define the format of the mapped region and repeat the pattern twice.

```
m = memmapfile('mybinary.bin',...
'Format',{'uint16',[50 1],'pressure';...
'uint16',[50,1],'temperature';...
'double',[5,10],'volume'},'Repeat',2)

m =

    Filename: 'd:\matlab\mybinary.bin'
    Writable: false
      Offset: 0
      Format: {'uint16' [50 1] 'pressure'
               'uint16' [50 1] 'temperature'
               'double' [5 10] 'volume'}
      Repeat: 2
        Data: 2x1 struct array with fields:
         pressure
      temperature
           volume
```

The `Data` property of the memory map, `m`, is a 2-by-1 structure array because the `Format` is applied twice.

Copy the `Data` property to a variable, `A`. Then, view the last block of `double` data, which you can access using the field name, `volume`.

```
A = m.Data;
```

```
myVolume = A(2).volume

myVolume =

     2    13    32     5     5    16     4    22     3     8
     2     9    53    38    13    19    23    85     2   120
    29    10     6     1     2     5     6    58    20    11
     7    15     4     1     5    18     1     4    14     8
     9     8     4     2     0     9     8     6     3     3
```

- “Map File to Memory”

Input Arguments

`filename` — Name of file to map

string

Name of the file to map including the file extension, specified as a string. The `filename` argument cannot include any wildcard characters (for example, `*` or `?`).

Example: `'myFile.dat'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `m = memmapfile('myFile.dat','Format','int32','Offset',255)` maps `int32` data in the file, `myFile.dat`, to memory starting from the 256th byte.

`'Writable'` — Type of access allowed to mapped region

`false` (default) | `true`

Type of access allowed to the mapped region, specified as the comma-separated pair consisting of `'Writable'` and either `true` or `false`. If the `Writable` property is set to `false`, the mapped region is read-only. If `true`, then write access is allowed.

Example: `'Writeable',true`

Data Types: `logical`

`'Offset'` — Number of bytes from start of file to start of mapped region

0 (default) | nonnegative integer

Number of bytes from the start of the file to the start of the mapped region, specified as the comma-separated pair consisting of `'Offset'` and a nonnegative integer. This value is zero-based. That is, an `Offset` value of 0 represents the start of the file.

Example: `'Offset',1024`

Data Types: `double`

`'Format'` — Format of mapped region

`'uint8'` (default) | string | `n`-by-3 cell array

Format of the mapped region contents, specified as the comma-separated pair consisting of `'Format'` and a single string or an `n`-by-3 cell array.

- If the file region you are mapping contains data of only one type, specify the `Format` value as a string identifying that type.
  Example: `'int16'`
- To specify an array shape to apply to the data read or written to the mapped file, and a field name to reference this array, specify the `Format` value as a 1-by-3 cell array. The first cell contains a string identifying the data type to apply to the mapped region. The second cell contains the array dimensions to apply to the region. The third cell contains a string specifying the field name to use in the `Data` structure array of the memory map.
  Example: `{'uint64',[30 4 10],'x'}`
- If the region you are mapping is composed of segments of varying data types or array shapes, you can specify an individual format for each segment using an `n`-by-3 cell array, where `n` is the number of segments.
  Example: `{'uint64',[30 4 10],'x'; 'uint32',[30 4 6],'y'}`

You can use any of the following data types when you specify a `Format` value:

- `'int8'`
- `'int16'`
- `'int32'`
- `'int64'`

- `'uint8'`
- `'uint16'`
- `'uint32'`
- `'uint64'`
- `'single'`
- `'double'`

Data Types: `char` | `cell`

`'Repeat'` — Number of times to apply `Format` parameter

`Inf` (default) | positive integer

Number of times to apply the `Format` parameter to the mapped region of the file, specified as the comma-separated pair consisting of `'Repeat'` and a positive integer. If the value of `Repeat` is `Inf`, then `memmapfile` applies the `Format` parameter until the end of the file.

Example: `'Repeat',2000`

Data Types: `double`

Output Arguments

`m` — Memory map

`memmapfile` object

Memory map, returned as a `memmapfile` object with the following properties.

Property	Description
`Filename`	Path and name of the mapped file
`Writable`	Type of access allowed to the mapped region
`Offset`	Number of bytes from the start of the file to the start of the mapped region
`Format`	Format of the contents of the mapped region, including data type, array size, and field name by which to access the data

Property	Description
`Repeat`	Number of times to apply the pattern specified by the `Format` property to the mapped region of the file
`Data`	Memory-mapped data from the file. `Data` can be a numeric array or a structure array with field names specified in the `Format` property

The values for any property (except for `Data`) are set at the time you call `memmapfile`, using name-value pair arguments.

Access any property of `m` with dot notation similar to accessing fields of a structure array. For example, to access the memory-mapped data in the `Data` property, do one of the following:

- If `Data` is a numeric array, call `m.Data`.
- If `Data` is a scalar (1-by-1) structure array, call `m.Data.`*`fieldname`*, where *`fieldname`* is the name of a field.
- If `Data` is a nonscalar structure array, call `m.Data(`*`index`*`).`*`fieldname`* where *`index`* is the index for the element in the structure array, and *`fieldname`* is the name of a field. For example, to access the file data in the `temperature` field of the first element of `Data`, call `m.Data(1).temperature`.

After you create a memory map, `m`, you can change the value of any of its properties, except for `Data`. To assign a new value, use dot notation. For example, to set a new `Offset` value for `m`, type:

```
m.Offset = 2048;
```

More About

Tips

- You can map only an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.
- After `memmapfile` locates the file, MATLAB stores the file's absolute pathname internally, and then uses this stored path to locate the file from that point on. As

a result, you can work in other directories outside your current work directory and retain access to the mapped file.

- `memmapfile` does not expand or append to a mapped file. Use instead standard file I/O functions like `fopen` and `fwrite`.

Algorithms

The actual mapping of a file to the MATLAB address space does not take place when you construct a `memmapfile` object. A memory map, based on the information currently stored in the mapped object, is generated the first time you reference or modify the `Data` property for that object.

- "Overview of Memory-Mapping"

Introduced before R2006a

memory

Display memory information

Syntax

```
memory
userview = memory
[userview systemview] = memory
```

Limitations

- The `memory` function is available only on Microsoft Windows systems. Results are dependant on your computer hardware and the load on your computer.

Description

`memory` displays information showing how much memory is available and how much the MATLAB software is currently using. The information displayed at your computer screen includes the following items, each of which is described in a section below:

- “Maximum Possible Array” on page 1-5031
- “Memory Available for All Arrays” on page 1-5032
- “Memory Used By MATLAB” on page 1-5033
- “Physical Memory (RAM)” on page 1-5033

`userview = memory` returns user-focused information on memory use in structure `userview`. The information returned in `userview` includes the following items, each of which is described in a section below:

- “Maximum Possible Array” on page 1-5031
- “Memory Available for All Arrays” on page 1-5032
- “Memory Used By MATLAB” on page 1-5033

`[userview systemview] = memory` returns both user- and system-focused information on memory use in structures `userview` and `systemview`, respectively.

The `userview` structure is described in the command syntax above. The information returned in `systemview` includes the following items, each of which is described in a section below:

- "Virtual Address Space" on page 1-5034
- "System Memory" on page 1-5034
- "Physical Memory (RAM)" on page 1-5033

Output Arguments

Each of the sections below describes a value that is displayed or returned by the `memory` function.

Maximum Possible Array

Maximum Possible Array is the size of the largest contiguous free memory block. As such, it is an upper bound on the largest single array MATLAB can create at this time.

MATLAB derives this number from the smaller of the following two values:

- The largest contiguous memory block found in the MATLAB virtual address space
- The total available system memory

To see how many array elements this number represents, divide by the number of bytes in the array class. For example, for a `double` array, divide by 8. The actual number of elements MATLAB can create is always fewer than this number.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Maximum possible array:`
`user = memory`	Structure field `user.MaxPossibleArrayBytes`

All values are double-precision and in units of bytes.

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems show either the first or second footnote; 64-bit systems show only the second footnote:

`Limited by contiguous virtual address space available.`

There is sufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the /3GB switch is in effect or not.

`Limited by System Memory (physical + swap file) available.`

There is insufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process.

Memory Available for All Arrays

Memory Available for All Arrays is the total amount of memory available to hold data. The amount of memory available is guaranteed to be at least as large as this field.

MATLAB derives this number from the smaller of the following two values:

- The total available MATLAB virtual address space
- The total available system memory

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Memory available for all arrays:`
`user = memory`	Structure field `user.MemAvailableAllArrays`

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems show either the first or second footnote; 64-bit systems show only the latter footnote:

`Limited by virtual address space available.`

There is sufficient system memory to allow mapping of all available virtual addresses in the MATLAB process virtual address space to system memory. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the /3GB switch is in effect or not.

`Limited by System Memory (physical + swap file) available.`

There is insufficient system memory to allow mapping of all available virtual addresses in the MATLAB process.

Memory Used By MATLAB

Memory Used By MATLAB is the total amount of system memory reserved for the MATLAB process. It is the sum of the physical memory and potential swap file usage.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Memory used by MATLAB:`
`user = memory`	Structure field `user.MemUsedMATLAB`

Physical Memory (RAM)

Physical Memory is the available and total amounts of physical memory (RAM) on the computer running MATLAB.

When you enter the `memory` command without assigning its output, MATLAB displays the total memory as a string. When you do assign the output, MATLAB returns both the available and total memory in a structure field. See the table below.

Command	Value	Returned in
`memory`	Total memory	String labelled `Physical Memory (RAM):`
`[user,sys] = memory`	Available memory	Structure field `sys.PhysicalMemory.Available`
	Total memory	Structure field `sys.PhysicalMemory.Total`

Available physical memory is the same as `Available` found in the Windows Task Manager: Performance/Physical Memory, and the total physical memory is the same as `Total`.

You can use the amount of available physical memory as a measure of how much data you can access quickly.

Virtual Address Space

Virtual Address Space is the amount of available and total virtual memory for the MATLAB process. MATLAB returns the information in two fields of the return structure: `Available` and `Total`.

Command	Return Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.VirtualAddressSpace.Available`
	Total memory	`sys.VirtualAddressSpace.Total`

You can monitor the difference:

```
VirtualAddressSpace.Total - VirtualAddressSpace.Available
```

as the Virtual Bytes counter in the Windows Performance program. (e.g., Windows XP Control Panel/Administrative Tool/Performance program).

System Memory

System Memory is the amount of available system memory on your computer system. This number includes the amount of available physical memory and the amount of available swap file space on the computer running MATLAB. MATLAB returns the information in the `SystemMemory` field of the return structure.

Command	Return Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.SystemMemory`

This is the same as the difference:

```
limit - total (in bytes)
```

found in the Windows Task Manager: Performance/Commit Charge.

Examples

Display memory statistics on a 64-bit Windows system:

```
memory
```

```
Maximum possible array:            14253 MB (1.495e+10 bytes) *
Memory available for all arrays:           14253 MB (1.495e+10 bytes) *
Memory used by MATLAB:              747 MB (7.833e+08 bytes)
Physical Memory (RAM):            12279 MB (1.288e+10 bytes)

*  Limited by System Memory (physical + swap file) available.
```

Return in the structure `userview`, information on the largest array MATLAB can create at this time, how much memory is available to hold data, and the amount of memory currently being used by your MATLAB process:

```
userview = memory
```

```
userview =

    MaxPossibleArrayBytes: 1.4957e+10
    MemAvailableAllArrays: 1.4957e+10
            MemUsedMATLAB: 784044032
```

Assign the output to two structures, `user` and `sys`, to obtain the information shown here:

```
[user sys] = memory;

%  --- Largest array MATLAB can create ---
user.MaxPossibleArrayBytes
```

```
ans =
   1.4956e+10
```

```
%  --- Memory available for data ---
user.MemAvailableAllArrays
```

```
ans =
   1.4956e+10
```

```
%  --- Memory used by MATLAB process ---
user.MemUsedMATLAB
```

```
ans =
   784039936
```

```
%  --- Virtual memory for MATLAB process ---
sys.VirtualAddressSpace

ans =
    Available: 8.7910e+12
        Total: 8.7961e+12

%  --- Physical memory and paging file ---
sys.SystemMemory

ans =
    Available: 1.4956e+10

%   --- Computer's physical memory ---
sys.PhysicalMemory

ans =
    Available: 2.7093e+09
        Total: 1.2876e+10
```

More About

Tips

Details on Memory Used By MATLAB

MATLAB computes the value for Memory Used By MATLAB by walking the MATLAB process memory structures and summing all the sections that have physical storage allocated in memory or in the paging file on disk.

Using the Windows Task Manager, you have for the MATLAB.exe image:

```
Mem Usage < MemUsedMATLAB < Mem Usage + VM Size (in bytes)
```

where both of the following are true:

- `Mem Usage` is the working set size in kilobytes.
- `VM Size` is the page file usage, or private bytes, in kilobytes.

The working set size is the portion of the MATLAB virtual address space that is *currently* resident in RAM and can be referenced without a memory page fault. The page file usage gives the portion of the MATLAB virtual address space that requires a backup that

doesn't already exist. Another name for page file usage is *private bytes*. It includes all MATLAB variables and workspaces. Since some of the pages in the page file may also be part of the working set, this sum is an overestimate of `MemUseMATLAB`. Note that there are virtual pages in the MATLAB process space that already have a backup. For example, code loaded from EXEs and DLLs and memory-mapped files. If any part of those files is in memory when the memory builtin is called, that memory will be counted as part of `MemUsedMATLAB`.

Reserved Addresses

Reserved addresses are addresses sets aside in the process virtual address space for some specific future use. These reserved addresses reduce the size of `MemAvailableAllArrays` and can reduce the size of the current or future value of `MaxPossibleArrayBytes`.

Example 1 — Java Virtual Machine (JVM)

At MATLAB startup, part of the MATLAB virtual address space is reserved by the Java Virtual Machine (JVM) and cannot be used for storing MATLAB arrays.

Example 2 — Standard Windows Heap Manager

MATLAB, by default, uses the standard Windows heap manager except for a set of small preselected allocation sizes. One characteristic of this heap manager is that its behavior depends upon whether the requested allocation is less than or greater than the fixed number of 524,280 bytes. For, example, if you create a sequence of MATLAB arrays, each less than 524,280 bytes, and then clear them all, the `MemUsedMATLAB` value before and after shows little change, and the `MemAvailableAllArrays` value is now smaller by the total space allocated.

The result is that, instead of globally freeing the extra memory, the memory becomes reserved. It can *only* be reused for arrays less than 524,280 bytes. You cannot reclaim this memory for a larger array except by restarting MATLAB.

See Also

`clear` | `pack` | `whos` | `inmem` | `save` | `load` | `mlock` | `munlock`

menu

Create multiple-choice dialog box

Syntax

```
choice = menu('mtitle','opt1','opt2',...,'optn')
choice = menu('mtitle',options)
```

Description

`choice = menu('mtitle','opt1','opt2',...,'optn')` displays the menu whose title is in the string variable `'mtitle'` and whose choices are string variables `'opt1'`, `'opt2'`, and so on. The menu opens in a modal dialog box. `menu` returns the number of the selected menu item, or 0 if the user clicks the close button on the window.

`choice = menu('mtitle',options)` , where options is a 1-by-N cell array of strings containing the menu choices.

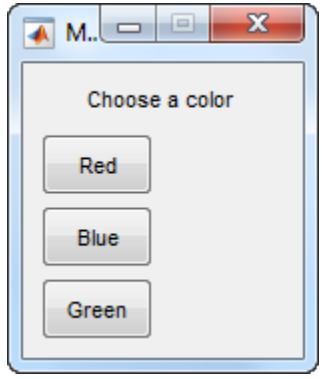
If the user's terminal provides a graphics capability, `menu` displays the menu items as push buttons in a figure window (Example 1). Otherwise. they will be given as a numbered list in the Command Window (Example 2).

Examples

Example 1

On a system with a display, `menu` displays choices as buttons in a dialog box:

```
choice = menu('Choose a color','Red','Blue','Green')
```
displays the following dialog box.



The number entered by the user in response to the prompt is returned as `choice` (i.e., `choice = 2` implies that the user selected `Blue`).

After input is accepted, the dialog box closes, returning the output in `choice`. You can use `choice` to control the color of a graph:

```
t = 0:.1:60;
s = sin(t);
color = ['r','b','g']
plot(t,s,color(choice))
```

Example 2

On a system without a display, `menu` displays choices in the Command Window:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the following text.

```
----- Choose a color -----
1) Red
2) Blue
3) Green
Select a menu number:
```

More About

Tips

To call `menu` from a uicontrol or other ui object, set that object's Interruptible property to `'on'`. For more information, see Uicontrol Properties.

See Also

`guide` | `input` | `uicontrol` | `uimenu`

Introduced before R2006a

mergecats

Merge categories in categorical array

Syntax

```
B = mergecats(A,oldcats)
B = mergecats(A,oldcats,newcat)
```

Description

`B = mergecats(A,oldcats)` merges two or more categories in `A` into the first category, `oldcats(1)`. Any values in `A` from `oldcats` become `oldcats(1)` in `B`.

`B = mergecats(A,oldcats,newcat)` merges `oldcats` into a single new category, `newcat`. Any values in `A` from `oldcats` become `newcat` in `B`.

Examples

Merge Two Categories into One

Create a categorical array containing various colors.

```
A = categorical({'red';'blue';'pink';'red';'blue';'red'})
```

```
A =

     red
     blue
     pink
     red
     blue
     red
```

`A` is a 6-by-1 categorical array.

Display the categories of `A`.

```
categories(A)
```

```
ans =

    'blue'
    'pink'
    'red'
```

The three categories are in alphabetical order.

Merge the categories `red` and `pink` into the category `red`. Specify `red` first in `oldcats` to use it as the merged category.

```
oldcats = {'red','pink'};
B = mergecats(A,oldcats)
```

```
B =

     red
     blue
     red
     red
     blue
     red
```

`mergecats` replaces the value `pink` from `A(3)` with `red`.

Display the categories of `B`.

```
categories(B)
```

```
ans =

    'blue'
    'red'
```

`B` has two categories instead of three.

Merge Alphabetically Listed Categories

Create a categorical array containing various items.

```
A = categorical({'shirt' 'pants'; 'shoes' 'shirt'; 'dress' 'belt'})
```

```
A =

     shirt      pants
     shoes      shirt
```

```
     dress      belt
```

Display the categories of A.

```
categories(A)
```

```
ans =

    'belt'
    'dress'
    'pants'
    'shirt'
    'shoes'
```

The five categories are in alphabetical order.

Merge the categories `belt` and `shoes` into a new category called `other`.

```
B = mergecats(A,{'belt' 'shoes'},'other')
```

```
B =

     shirt      pants
     other      shirt
     dress      other
```

The value `other` replaces all instances of `belt` and `shoes`.

Display the categories of B.

```
categories(B)
```

```
ans =

    'other'
    'dress'
    'pants'
    'shirt'
```

B has four categories and the order is no longer alphabetical. `other` appears in place of `belt`.

Merge Categories of Ordinal Categorical Array

Create an ordinal categorical array.

```
A = categorical([1 2 3 2 1],1:3,{'poor','fair','good'},'Ordinal',true)
```

```
A =

     poor      fair      good      fair      poor
```

Display the categories of A.

```
categories(A)
```

```
ans =

    'poor'
    'fair'
    'good'
```

Since A is ordinal, the categories have the mathematical ordering `poor < fair < good`.

Consider all `fair` or `poor` values to be `bad`. Since A is ordinal, the categories to merge must be consecutive.

```
B = mergecats(A,{'fair' 'poor'},'bad')
```

```
B =

     bad       bad       good      bad       bad
```

The value `bad` replaces all instances of `fair` and `poor`.

Display the categories of B.

```
categories(B)
```

```
ans =

    'bad'
    'good'
```

B has two categories with the mathematical ordering: `bad < good`.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`oldcats` — Categories to merge

cell array of strings

Categories to merge, specified as a cell array of strings. If `A` is ordinal, the categories to merge must be consecutive.

`newcat` — New category

`oldcats(1)` (default) | string

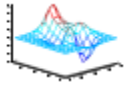
New category, specified as a string.

See Also

`addcats` | `categories` | `iscategory` | `removecats` | `renamecats` | `reordercats` | `setcats`

mesh

Mesh plot



Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
mesh(...,'PropertyName',PropertyValue,...)
mesh(axes_handles,...)
h = mesh(...)
```

Description

`mesh(X,Y,Z)` draws a wireframe mesh with color determined by `Z`, so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, $(X(j), Y(i), Z(i,j))$ are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, $(X(i,j), Y(i,j), Z(i,j))$ are the intersections of the wireframe grid lines.

`mesh(Z)` draws a wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`mesh(...,C)` draws a wireframe mesh with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`mesh(...,'PropertyName',PropertyValue,...)` sets the value of the specified surface property. Multiple property values can be set with a single statement.

`mesh(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

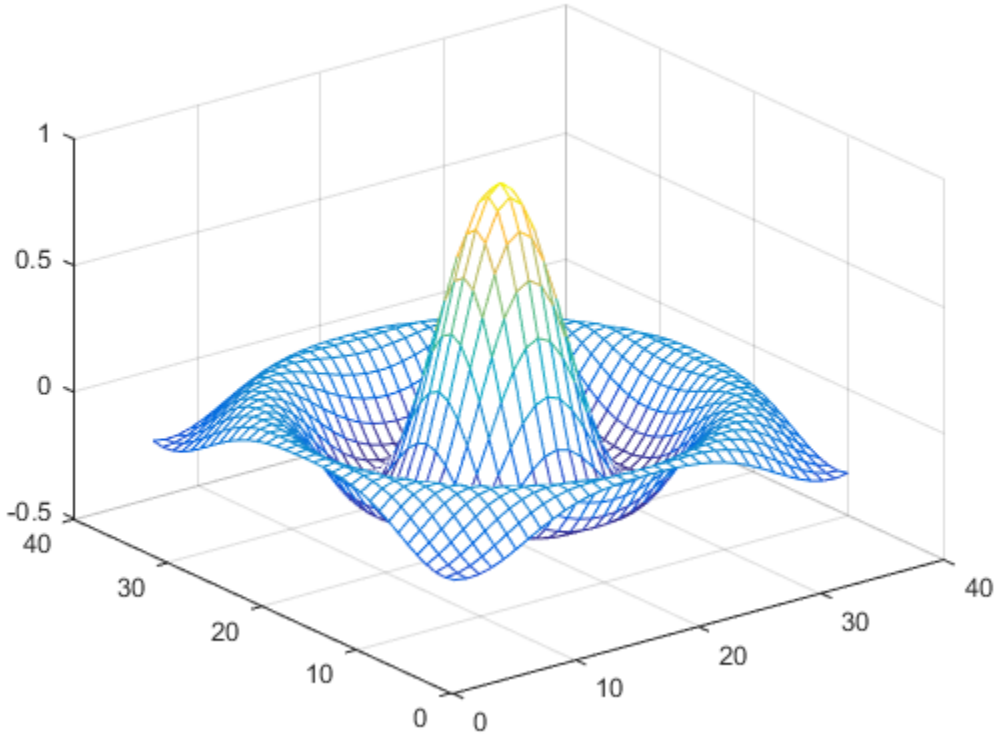
`h = mesh(...)` returns a handle to a Chart Surface Properties graphics object.

Examples

Create Mesh Plot of Sinc Function

Create a mesh plot of the sinc function, $z = \sin(r)/r$.

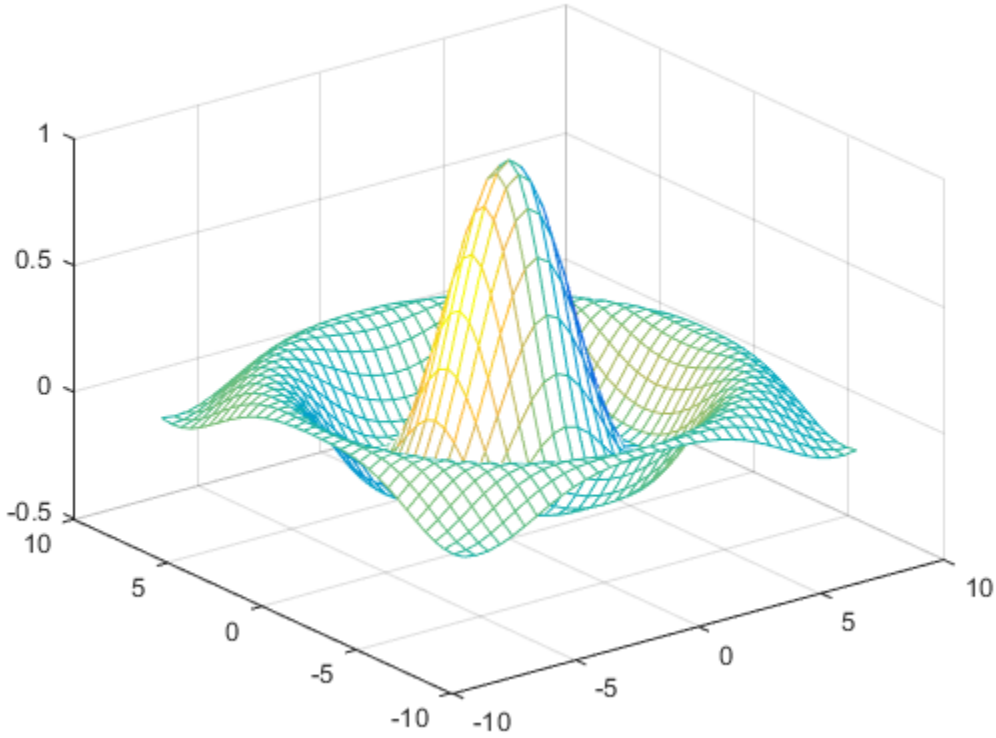
```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;

figure
mesh(Z)
```



Specify Color for Mesh Plot

Specify a color matrix for a mesh plot.

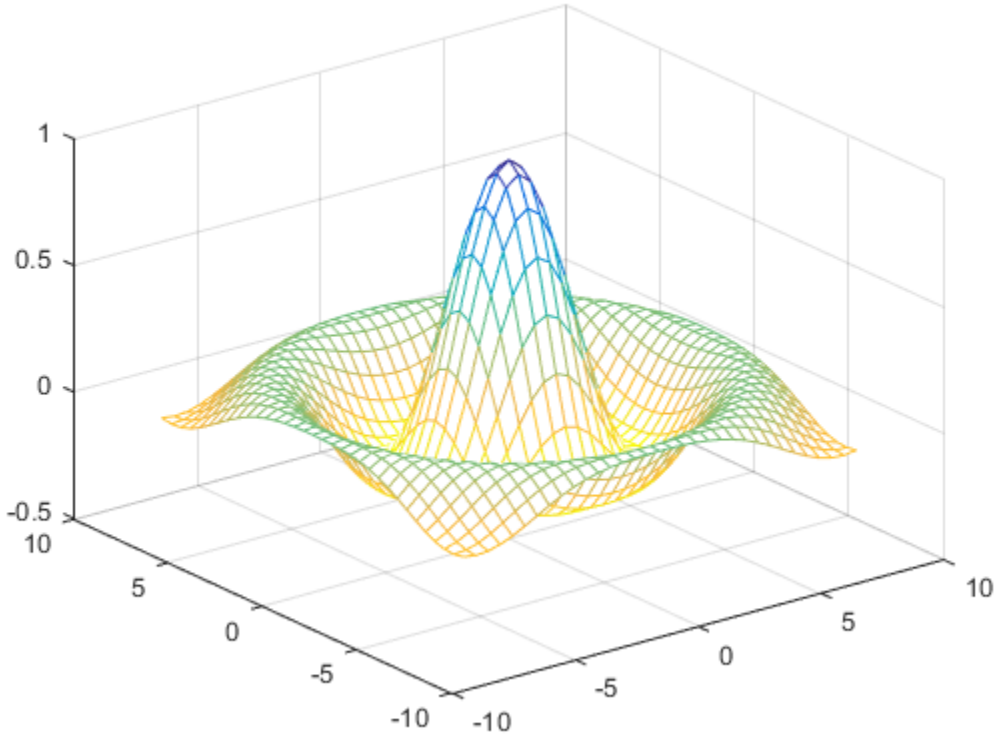
```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
C = gradient(Z);

figure
mesh(X,Y,Z,C)
```



Change Lighting and Line Width for Mesh Plot

Change the lighting and the line width for a mesh plot using `Name,Value` pair arguments.

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
C = del2(Z);

figure
mesh(X,Y,Z,C,'FaceLighting','gouraud','LineWidth',0.3)
```



More About

Tips

`mesh` does not accept complex inputs.

A mesh is drawn as a `Surfaceplot` graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or `none` when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the simulation of hidden-surface elimination in the mesh, and the `shading` command controls the shading model.

- “Representing Data as a Surface”

See Also

Functions

`scatteredInterpolant` | `axis` | `colormap` | `griddata` | `hidden` | `hold` | `meshc` | `meshgrid` | `meshz` | `shading` | `surf` | `surface` | `view` | `waterfall`

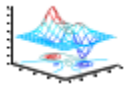
Properties

Chart Surface Properties

Introduced before R2006a

meshc

Plot a contour graph under mesh graph



Syntax

```
meshc(X,Y,Z)
meshc(Z)
meshc(...,C)
meshc(axes_handles,...)
h = meshc(...)
```

Description

`meshc(X,Y,Z)` draws a wireframe mesh and a contour plot under it with color determined by `Z`, so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, `(`X`(j)`, Y`(i)`, Z`(i,j))` are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, `(`X`(i,j)`, Y`(i,j)`, Z`(i,j))` are the intersections of the wireframe grid lines.

`meshc(Z)` draws a contour plot under wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`meshc(...,C)` draws a `meshc` graph with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`meshc(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

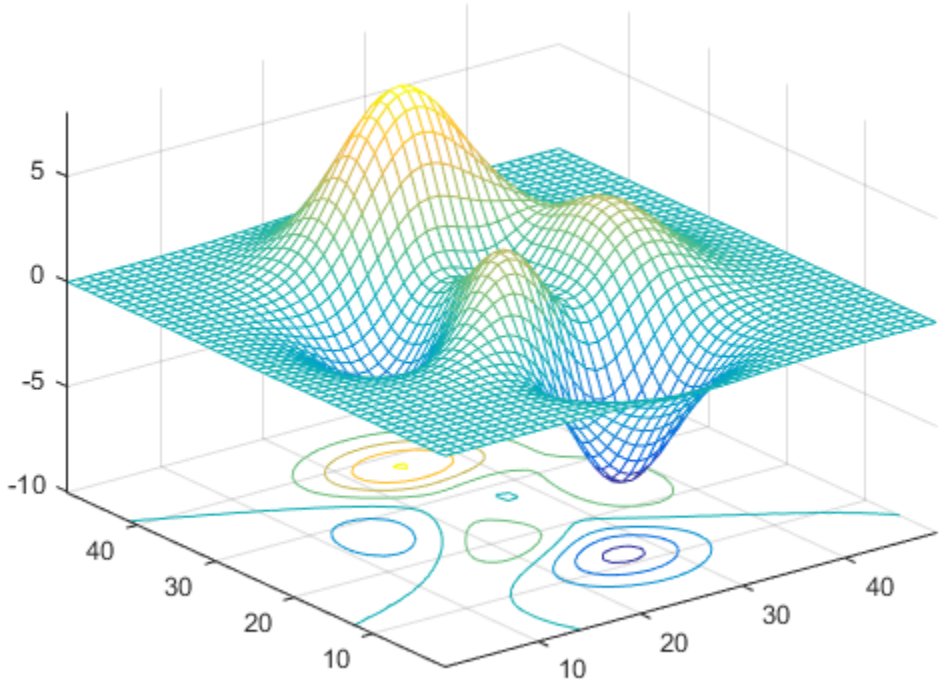
`h = meshc(...)` returns handles to the Chart Surface Properties and Contour Properties graphics object.

Examples

Display Contour Plot Under Mesh Plot

Use `meshc` to display a combination of a mesh plot and a contour plot of the `peaks` function.

```
figure
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(Z)
```



More About

Tips

`meshc` does not accept complex inputs.

A mesh is drawn as a `Surfaceplot` graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or `none` when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the simulation of hidden-surface elimination in the mesh, and the `shading` command controls the shading model.

Algorithms

The range of `X`, `Y`, and `Z`, or the current settings of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties, determine the axis limits. `axis` sets these properties.

The range of `C`, or the current settings of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function), determine the color scaling. Use the scaled color values are used as indices into the current colormap.

The mesh rendering functions produce color values by mapping the *z* data values (or an explicit color array) onto the current colormap. The MATLAB default behavior is to compute the color limits automatically using the minimum and maximum data values (also set using `caxis auto`). The minimum data value maps to the first color value in the colormap and the maximum data value maps to the last color value in the colormap. MATLAB performs a linear transformation on the intermediate values to map them to the current colormap.

`meshc` calls `mesh`, turns `hold` on, and then calls `contour` and positions the contour on the *x*-*y* plane. For additional control over the appearance of the contours, issue these commands directly. You can combine other types of graphs in this manner, for example `surf` and `pcolor` plots.

`meshc` assumes that `X` and `Y` are monotonically increasing. If `X` or `Y` is irregularly spaced, `contour3` calculates contours using a regularly spaced contour grid, and then it transforms the data to `X` or `Y`.

See Also

Functions

`axis` | `caxis` | `colormap` | `contour` | `hidden` | `hold` | `mesh` | `meshgrid` | `meshz` | `shading` | `surf` | `surface` | `surfc` | `surfl` | `view` | `waterfall`

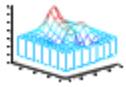
Properties

Chart Surface Properties | Contour Properties

Introduced before R2006a

meshz

Plot a curtain around mesh plot



Syntax

```
meshz(X,Y,Z)
meshz(Z)
meshz(...,C)
meshz(axes_handles,...)
h = meshz(...)
```

Description

`meshz(X,Y,Z)` draws a curtain around the wireframe mesh with color determined by `Z`, so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, `(`X`(j)`, Y`(i)`, Z`(i,j))` are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, `(`X`(i,j)`, Y`(i,j)`, Z`(i,j))` are the intersections of the wireframe grid lines.

`meshz(Z)` draws a curtain around the wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`meshz(...,C)` draws a `meshz` graph with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`meshz(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

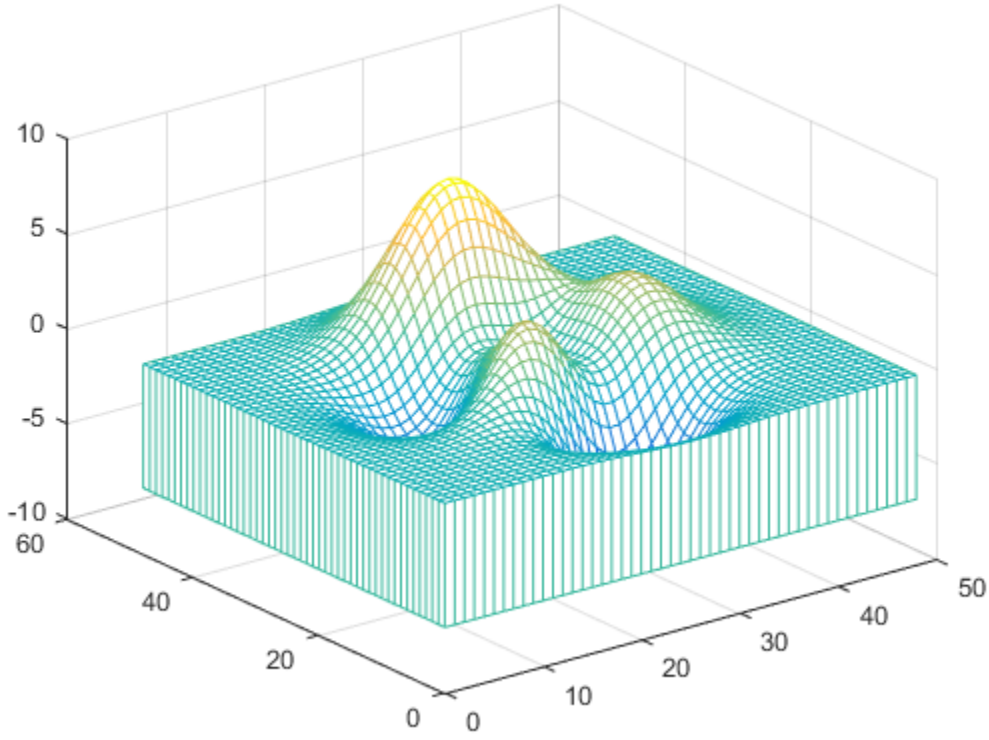
`h = meshz(...)` returns a handle to a Chart Surface Properties graphics object.

Examples

Curtain Plot of Peaks Function

Generate a curtain plot of the `peaks` function using `meshz`.

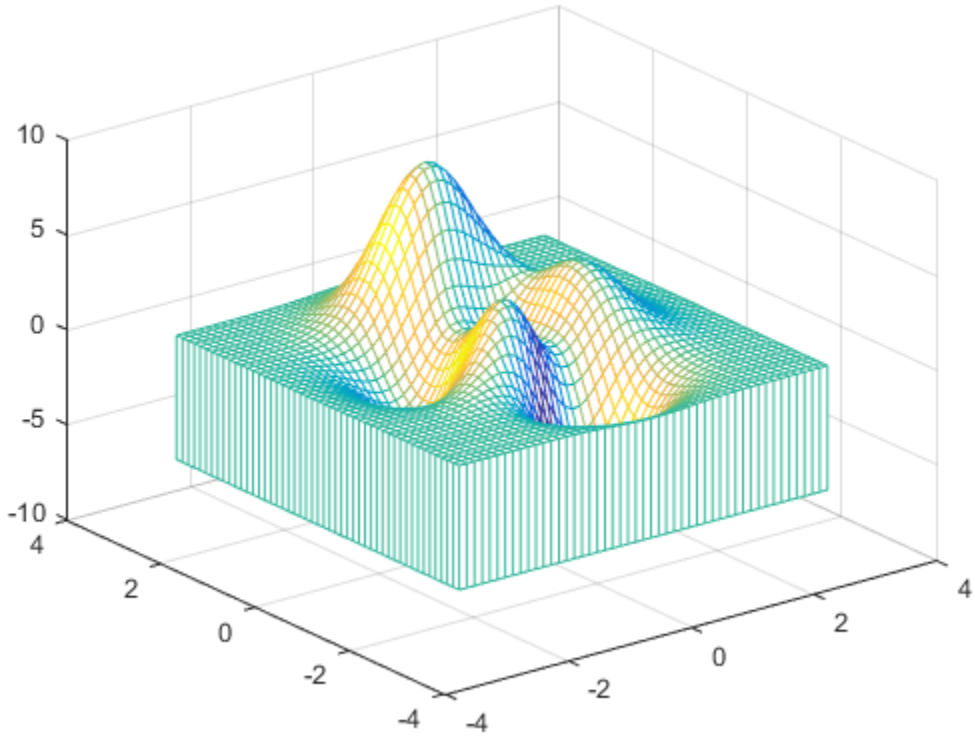
```
figure
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshz(Z)
```



Specify Color for Curtain Plot

Specify a color matrix for a curtain plot.

```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
C = gradient(Z);

figure
meshz(X,Y,Z,C)
```



More About

Tips

`meshz` does not accept complex inputs.

A mesh is drawn as a chart surface graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or `none` when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the simulation of hidden-surface elimination in the mesh, and the `shading` command controls the shading model.

See Also

Functions

`axis` | `caxis` | `colormap` | `contour` | `hidden` | `hold` | `mesh` | `meshc` | `meshgrid` | `shading` | `surf` | `surface` | `surfc` | `surfl` | `view` | `waterfall`

Properties

Chart Surface Properties

Introduced before R2006a

meshgrid

Rectangular grid in 2-D and 3-D space

Syntax

```
[X,Y] = meshgrid(xgv,ygv)
[X,Y,Z] = meshgrid(xgv,ygv,zgv)
[X,Y] = meshgrid(gv)
[X,Y,Z] = meshgrid(gv)
```

Description

`[X,Y] = meshgrid(xgv,ygv)` replicates the grid vectors `xgv` and `ygv` to produce a full grid. This grid is represented by the output coordinate arrays `X` and `Y`. The output coordinate arrays `X` and `Y` contain copies of the grid vectors `xgv` and `ygv` respectively. The sizes of the output arrays are determined by the length of the grid vectors. For grid vectors `xgv` and `ygv` of length `M` and `N` respectively, `X` and `Y` will have `N` rows and `M` columns.

`[X,Y,Z] = meshgrid(xgv,ygv,zgv)` produces three-dimensional coordinate arrays. The output coordinate arrays `X`, `Y`, and `Z` contain copies of the grid vectors `xgv`, `ygv`, and `zgv` respectively. The sizes of the output arrays are determined by the length of the grid vectors. For grid vectors `xgv`, `ygv`, and `zgv` of length `M`, `N`, and `P` respectively, `X`, `Y`, and `Z` will have `N` rows, `M` columns, and `P` pages.

`[X,Y] = meshgrid(gv)` is the same as `[X,Y] = meshgrid(gv,gv)`. In other words, you can reuse the same grid vector in each respective dimension. The dimensionality of the output arrays is determined by the number of output arguments.

`[X,Y,Z] = meshgrid(gv)` is the same as `[X,Y,Z] = meshgrid(gv,gv,gv)`. Again, the dimensionality of the output arrays is determined by the number of output arguments.

The output coordinate arrays are typically used to evaluate functions of two or three variables. They are also frequently used to create surface and volumetric plots.

Input Arguments

`xgv,ygv,zgv`

Grid vectors specifying a series of grid point coordinates in the `x`, `y` and `z` directions, respectively.

`gv`

Generic grid vector specifying a series of point coordinates.

Output Arguments

`X,Y,Z`

Output arrays that specify the full grid.

Examples

2-D Grid From Vectors

Create a full grid from two monotonically increasing grid vectors:

```
[X,Y] = meshgrid(1:3,10:14)
X =
     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3
Y =
    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14
```

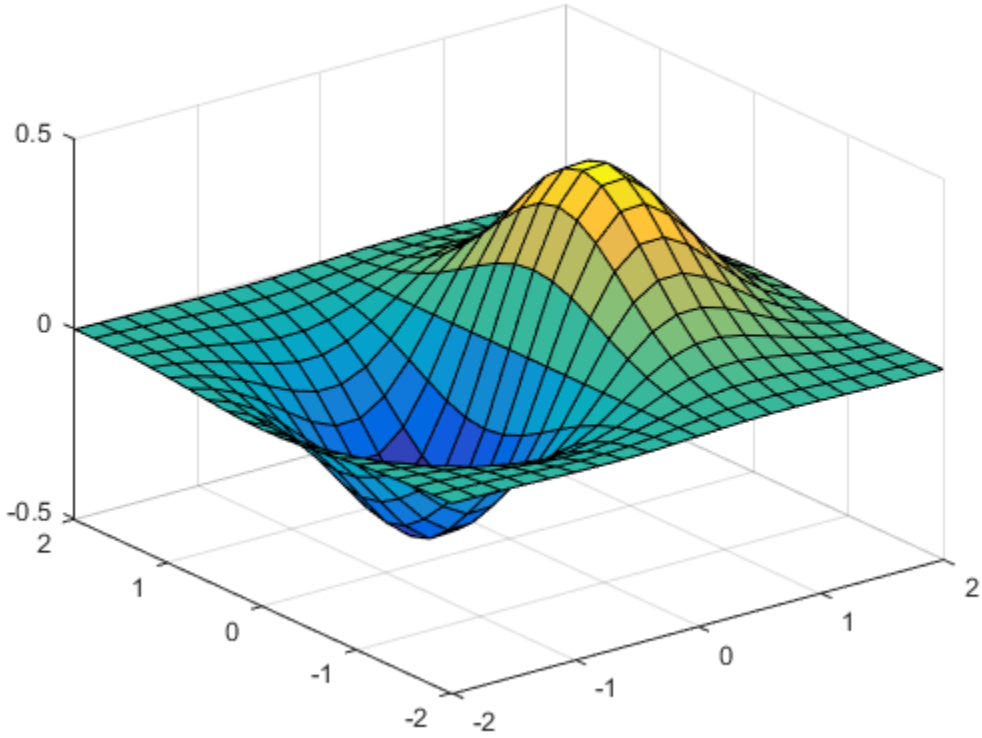
Plot 3-D Functional Surface

Use `meshgrid` to create a gridded *(X,Y)* domain.

```
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
```

Evaluate the function $z(x,y) = xe^{-x^2-y^2}$ over this domain and generate a surface plot of the results.

```
Z = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
```



More About

Tips

The `meshgrid` function is similar to `ndgrid`, however `meshgrid` is restricted to 2-D and 3-D while `ndgrid` supports 1-D to N-D. The coordinates output by each function are the same, but the shape of the output arrays in the first two dimensions are different. For grid vectors `x1gv`, `x2gv` and `x3gv` of length `M`, `N` and `P` respectively, `meshgrid(x1gv, x2gv)` will output arrays of size `N`-by-`M` while `ndgrid(x1gv, x2gv)` outputs arrays of size `M`-by-`N`. Similarly, `meshgrid(x1gv, x2gv, x3gv)` will output arrays of size `N`-by-`M`-by-`P` while `ndgrid(x1gv, x2gv, x3gv)` outputs arrays of size `M`-by-`N`-by-`P`.

See “Grid Representation” in the MATLAB Mathematics documentation for more information.

- “Interpolating Gridded Data”

See Also

`griddedInterpolant` | `mesh` | `ndgrid` | `surf`

Introduced before R2006a

meta.class

Describe MATLAB class

Description

Instances of the `meta.class` class contains information about MATLAB classes. The read/write properties of the `meta.class` class correspond to class attributes and are set only from within class definitions on the `classdef` line. You can query the read–only properties of the `meta.class` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class).

You cannot instantiate a `meta.class` object directly. You can construct a `meta.class` object from an instance of a class or using the class name:

- `metaclass` — returns a `meta.class` object representing the object passed as an argument.
- `?`*`ClassName`* — returns a `meta.class` object representing the named class.
- `fromName` — static method returns a `meta.class` object representing the named class.

For example, the `metaclass` function returns the `meta.class` object representing `MyClass`.

```
ob = MyClass;
obmeta = metaclass(ob);
obmeta.Name
```

```
ans =
MyClass
```

You can use the class name to obtain the `meta.class` object:

```
obmeta = ?MyClass;
```

You can also use the `fromName` static method:

```
obmeta = meta.class.fromName('MyClass');
```

Properties

Property	Purpose
`Abstract` attribute, default = `false`	If `true`, this class is an abstract class (cannot be instantiated). See “Abstract Classes” for more information.
`AllowedSubclasses` attribute, write only from `classdef` statement	List classes that can subclass this class. Specify subclasses as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects Specify `meta.class` objects using the `?ClassName` syntax only.
`ConstructOnLoad` attribute, default = `false`	If `true`, MATLAB calls the class constructor automatically when loading an object from a MAT-file. Therefore, the construction must be implemented so that calling it with no arguments does not produce an error. See “Save and Load Process”
`ContainingPackage` read only	A `meta.package` object describing the package within which this class is contained, or an empty object if this class is not in a package. See “Packages Create Namespaces”.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`Enumeration` attribute, default = `false`	If `true`, this class is an enumeration class. See “Working with Enumerations”.
`EventList` read only	An array of `meta.event` objects describing each event defined by this class, including all inherited events. See “Events”.
`Events` read only	A cell array of `meta.event` objects describing each event defined by this class, including all inherited events.

Property	Purpose
Use `EventList` instead	
`EnumerationMemberList`	An array of `meta.EnumeratedValue` objects describing the member names defined by an enumeration class. See “Enumerations” for more information on enumeration classes.
`EnumeratedValues` read only Use `EnumeratedMemberList` instead	A cell array of `meta.EnumeratedValue` objects describing the member names defined by an enumeration class. See “Enumerations” for more information on enumeration classes.
`Hidden` attribute, default = `false`	If set to `true`, the class does not appear in the output of MATLAB commands or tools that display class names.
`InferiorClasses` attribute, default = `{}`	A cell array of `meta.class` objects defining the precedence of classes represented by the list as inferior to this class. See “Class Precedence”
`MethodList` read only	An array of `meta.method` objects describing each method defined by this class, including all inherited public and protected methods. See “How to Use Methods”.
`Methods` read only Use `MethodList` instead	A cell array of `meta.method` objects describing each method defined by this class, including all inherited public and protected methods.
`Name` read only	Name of the class associated with this `meta.class` object (`char` array)
`PropertyList` read only	An array of `meta.property` objects describing each property defined by this class, including all inherited public and protected properties. See “Properties”.

Property	Purpose
`Properties` read only Use `PropertyList` instead	A cell array of `meta.property` objects describing each property defined by this class, including all inherited public and protected properties. See “Properties”.
`Sealed` attribute, default = `false`	If `true`, the class cannot be subclassed.
`SuperClassList` read only	An array of `meta.class` objects describing each direct superclass from which this class is derived. See “Creating Subclasses — Syntax and Techniques”.
`SuperClasses` read only Use `SuperClassList` instead	A cell array of `meta.class` objects describing each direct superclass from which this class is derived.

Methods

Method	Purpose
`fromName`	Returns the `meta.class` object associated with the specified class name.
`tf = eq(Cls)`	Equality function (`a == b`). Use to test if two variables refer to equal classes (classes that contain exactly the same list of elements).
`tf = ne(Cls)`	Not equal function (`a ~= b`). Use to test if two variables refer to different meta–classes.
`tf = lt(ClsA,ClsB)`	Less than function (`ClsA < ClsB`). Use to determine if `ClsA` is a strict subclass of `ClsB` (i.e., a strict subclass means `ClsX < ClsX` is `false`).
`tf = le(ClsA,ClsB)`	Less than or equal to function (`ClsA <= ClsB`). Use to determine if `ClsA` is a subclass of `ClsB`.
`tf = gt(ClsA,ClsB)`	Greater than function (`ClsA > ClsB`). Use to determine if `ClsA` is a strict superclass of `ClsB` (i.e., a strict superclass means `ClsX > ClsX` is `false`).

Method	Purpose
`tf = ge(ClsA,ClsB)`	Greater than or equal to function (`ClsA >= ClsB`). Use to determine if `ClsA` is a superclass of `ClsB`.

Events

Event	Purpose
`InstanceCreated`	If the class is a handle class, this event occurs every time a new instance of this handle class is created, including new instances of any subclasses. The event occurs immediately after all constructor functions finish executing.
`InstanceDestroyed`	If the class is a handle class, this event occurs every time an instance of this handle class is destroyed, including all subclasses. The event occurs immediately before any destructor functions execute.

Examples

Find property attributes using the `handle` class `findobj` method and the `audioplayer` `meta.class` object. Determine if a class defines the property named `SampleRate` and does it have public set access.

```
mc = ?audioplayer;
mp = findobj(mc.PropertyList,'Name','SampleRate');
strcmp(mp.SetAccess,'public')
    ...
```

More About

- "Getting Information About Classes and Objects"

See Also

`fromName` | `meta.property` | `meta.method` | `meta.event` | `meta.package`

meta.class.fromName

Return `meta.class` object associated with named class

Syntax

```
mcls = meta.class.fromName('ClassName')
```

Description

`mcls = meta.class.fromName('ClassName')` is a static method that returns the `meta.class` object associated with the class *`ClassName`*. Note that you can also use the `?` operator to obtain the `meta.class` object for a class name:

```
mcls = ?ClassName;
```

The equivalent call to `meta.class.fromName` is:

```
mcls = meta.class.fromName('ClassName');
```

Use `meta.class.fromName` when using a `char` variable for the class name:

```
function mcls = getMetaClass(clname)
   % Do error checking
   mcls = meta.class.fromName(clname);
   ...
end
```

See Also

`meta.class`

meta.DynamicProperty

Describe dynamic property of MATLAB object

Description

The `meta.DynamicProperty` class contains descriptive information about dynamic properties that you have added to an instance of a MATLAB classes. The MATLAB class must be a subclass of `dynamicprops`. The properties of the `meta.DynamicProperty` class correspond to property attributes that you specify from within class definitions. Dynamic properties are not defined in `classdef` blocks, but you can set their attributes by setting the `meta.DynamicProperty` object properties.

You add a dynamic property to an object using the `addprop` method of the `dynamicprops` class. The `addprop` method returns a `meta.DynamicProperty` instance representing the new dynamic property. You can modify the properties of the `meta.DynamicProperty` object to set the attributes of the dynamic property or to add set and get access methods, which would be defined in the `classdef` for regular properties.

You cannot instantiate the `meta.DynamicProperty` class. You must use `addprop` to obtain a `meta.DynamicProperty` object.

To remove the dynamic property, call the `delete` handle class method on the `meta.DynamicProperty` object.

Obtain a `meta.DynamicProperty` object from the `addprops` method, which returns an array of `meta.DynamicProperty` objects, one for each dynamic property.

See “Dynamic Properties — Adding Properties to an Instance” for more information.

Properties

Property	Purpose
`Name`	Name of the property.
`Description`	Currently not used
`DetailedDescription`	Currently not used

Property	Purpose
`AbortSet`	If `true`, and this property belongs to a handle class, then MATLAB does not set the property value if the new value is the same as the current value. This approach prevents the triggering of property `PreSet` and `PostSet` events.
`Abstract` attribute, default = `false`	If `true`, the property has no implementation, but a concrete subclass must redefine this property without `Abstract` being set to `true`. • Abstract properties cannot define set or get access methods. See "Property Access Methods" • Abstract properties cannot define initial values. "Assigning a Default Value" • All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. • `Abstract=true` should be used with the class attribute `Sealed=false` (the default).
`Access`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only (not subclasses) List of classes that have get and set access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. Use `Access` to set both `SetAccess` and `GetAccess` to the same value. Query the values of `SetAccess` and `GetAccess` directly (not `Access`).

Property	Purpose
`Constant` attribute, default = `false`	Set to `true` if you want only one value for this property in all instances of the class. • Subclasses inherit constant properties, but cannot change them. • `Constant` properties cannot be `Dependent` • `SetAccess` is ignored. See “Properties with Constant Values”
`DefaultValue`	Querying this property returns an error because dynamic properties cannot define default values.
`DefiningClass`	The `meta.class` object representing the class that defines this property.
`GetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only
`Dependent` attribute, default = `false`	If `false`, property value is stored in object. If `true`, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See “Property Get Methods”
`Transient` attribute, default = `false`	If `true`, property value is not saved when object is saved to a file. See “Save and Load Process” for more about saving objects.
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).

Property	Purpose
`GetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See "Property-Set and Query Events"
`SetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See "Property-Set and Query Events"
`GetMethod`	Function handle of the get method associated with this property. Empty if there is no get method specified. See "Get Method Syntax"
`SetMethod`	Function handle of the set method associated with this property. Empty if there is no set method specified. See "Property Set Methods"
`HasDefault`	Always `false`. Dynamic properties cannot define default values.

Events

See "Listen for Changes to Property Values" for information on using property events.

Event Name	Purpose
`PreGet`	Event occurs just before property is queried.
`PostGet`	Event occurs just after property has been queried
`PreSet`	Event occurs just before this property is modified
`PostSet`	Event occurs just after this property has been modified
`ObjectBeingDestroyed`	Inherited from `handle`

See Also

`addprop` | `handle`

meta.EnumeratedValue

Describe enumeration member of MATLAB class

Description

The `meta.EnumeratedValue` class contains information about enumeration members defined by MATLAB classes. The properties of a `meta.EnumeratedValue` object correspond the to attributes of the enumeration member being described.

All `meta.EnumeratedValue` properties are read-only. Query the `meta.EnumeratedValue` object to obtain information about the enumeration member it describes.

Obtain a `meta.EnumeratedValue` object from the `EnumerationMemberList` property of the `meta.class` object. `EnumerationMemberList` is an array of `Meta.EnumeratedValue` instances, one per enumeration member.

The `meta.EnumeratedValue` class is a subclass of the `handle` class.

Example

To access the `meta.EnumeratedValue` objects for a class, first create a `meta.class` object for that class. For example, give the following `OnOff` class definition:

```
classdef OnOff < logical
   enumeration
      On (true)
      Off (false)
   end
end
```

Obtain a `meta.EnumeratedValue` object from the `EnumerationMemberList` property of the `meta.class` object:

```
% Obtain the meta.class instance for the OnOff class
mc = ?OnOff;
% Get the array of EnumerateValue objects
enumList = mc.EnumerationMemberList;
% Access the Name property of the first object in the array
```

```
enumList(1).Name =
ans =
On
```

Properties

Property	Purpose
`Name` read only	Name of the enumeration member associated with this `meta.EnumeratedValue` object
`Description` read only	This property is not used
`DetailedDescription` read only	This property is not used

Methods

See the `handle` superclass for inherited methods.

Events

See the `handle` superclass for inherited events.

More About

- "Working with Enumerations"
- "Getting Information About Classes and Objects"

See Also

`meta.property` | `meta.event` | `meta.class` | `meta.method`

meta.event

Describe event of MATLAB class

Description

The `meta.event` class provides information about MATLAB class events. The read/write properties of the `meta.event` class correspond to event attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.event` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining the event).

You cannot instantiate a `meta.event` object directly. Obtain a `meta.event` object from the `meta.class` `EventList` property, which contains an array of `meta.event` objects, one for each event defined by the class. For example, replace *`ClassName`* with the name of the class whose events you want to query:

```
mco = ?ClassName;
elist = mco.EventList;
elist(1).Name; % name of first event in list
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Name` read only	Name of the event.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`Hidden`	If `true`, the event does not appear in the list of events returned by the `events` function (or other event listing functions or viewers)
`ListenAccess`	Determines where you can create listeners for the event.

Property	Purpose

	• `public` — unrestricted access
	• `protected` — access from methods in class or subclasses
	• `private` — access by class methods only (not from subclasses)
	• List classes that have listen access to this event. Specify classes as `meta.class` objects in the form:
	  • A single `meta.class` object
	  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.
	See "Control Access to Class Members"

`NotifyAccess`	Determines where code can trigger the event.
	• `public` — any code can trigger event
	• `protected` — can trigger event from methods in class or subclasses
	• `private` — can trigger event by class methods only (not from subclasses)
	• List classes that have notify access to this event. Specify classes as `meta.class` objects in the form:
	  • A single `meta.class` object
	  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.
	See "Control Access to Class Members"

`DefiningClass`	The `meta.class` object representing the class that defines this event.

More About

- "Events"
- "Getting Information About Classes and Objects"

See Also

meta.property | meta.class | meta.method | metaclass

meta.MetaData class

Package: meta
Superclasses: matlab.mixin.Heterogeneous

Superclass for MATLAB object metadata

Description

The `meta.MetaData` class of objects represent MATLAB class definitions and the constituent parts of those definitions, such as properties and methods. Metadata enable a program to get information about a class definition.

The `meta.MetaData` class forms the root of the metadata class hierarchy, which enables the formation of arrays of metadata objects belonging to different specific classes.

MATLAB uses instances of the `meta.MetaData` class as the default object to fill in missing array elements.

`findobj` and `findprop`, can search the metadata hierarchy and return an array of different metadata objects. These function require the ability to form heterogeneous arrays containing various metaclass objects.

See the `matlab.mixin.Heterogeneous` class for more information on heterogeneous hierarchies.

Construction

You cannot create an instance of the `meta.MetaData` class directly. MATLAB constructs instances of this class as required.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

This example shows how the `meta.MetaData` class facilitates working with metaclasses.

Create a `meta.class` instance representing the MATLAB `timeseries` class:

```
>> mc = ?timeseries;
```

MATLAB uses `meta.MetaData` objects to fill empty array elements:

```
>> m(2) = mc
>> class(m(1))

ans =

meta.MetaData
>> class(m(2))

ans =

meta.class
```

Use `findobj` to find all properties and methods that have protected access:

```
>> protectedMembers = findobj(mc,{'Access','protected'},...
'-or',{'SetAccess','protected'},...
'-or',{'GetAccess','protected'});
```

The `timeseries` class defines both properties and methods that have protected access. Therefore, `findobj` returns a heterogeneous array of class `meta.MetaData`. This array contains both `meta.property` and `meta.method` objects.

```
>> protectedMembers

protectedMembers =

  11x1 heterogeneous meta.MetaData (meta.property, meta.method)
  handle with no properties.
  Package: meta
>> class(protectedMembers(1))

ans =

meta.property

>> protectedMembers(1).Name
```

```
ans =

Length
>> protectedMembers(1).SetAccess

ans =

protected
>> protectedMembers(1).GetAccess

ans =

public
```

See Also

`handle` | `matlab.mixin.Heterogeneous`

How To

- Class Attributes
- Property Attributes
- “Getting Information About Classes and Objects”

meta.method

Describe method of MATLAB class

Description

The `meta.method` class provides information about the methods of MATLAB classes. The read/write properties of the `meta.method` class correspond to method attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.method` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining a method).

You cannot instantiate a `meta.method` object directly. Obtain a `meta.method` object from the `meta.class` `MethodList` property, which contains an array of `meta.method` objects, one for each class method. For example, replace *`ClassName`* with the name of the class whose methods you want to query:

```
mco = ?ClassName;
mlist = mco.MethodList;
mlist(1).Name; % name of first method in the list
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Abstract`	If `true`, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method. • Subclasses are not required to define the same number of input and output arguments. • The method can have comments after the `function` line

Property	Purpose
	• Does not contain `function` or `end` keywords, only the function syntax (e.g., `[a,b] = myMethod(x,y)`)
`Access` attribute, default = `public`	Determines what code can call this method. • `public` — unrestricted access • `protected` — access from methods in class or subclasses • `private` — access by class methods only (not from subclasses)
`DefiningClass`	The `meta.class` object representing the class that defines this method.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`Hidden` attribute, default = `false`	When `false`, the method name shows in the list of methods displayed using the `methods` or `methodsview` commands. If set to `true`, the method name is not included in these listings.
`Name` read only	Name of the method.
`Sealed` attribute, default = `false`	If `true`, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.

`Static` attribute, default = `false`	Set to `true` to define a method that does not depend on an object of the class and does not require an object argument. Call static methods using the class name in place of the object:
	`classname.methodname()`
	Or with an instance of the class, like any method:
	`o.methodname()`
	See “Static Methods”

More About

- "Methods"
- "Getting Information About Classes and Objects"

See Also

`meta.property` | `meta.class` | `meta.event` | `metaclass`

meta.package

Describe MATLAB package

Description

The `meta.package` class contains information about MATLAB packages.

You cannot instantiate a `meta.package` object directly. Obtain a `meta.package` object from the `meta.class` `ContainingPackage` property, which contains a `meta.package` object, or an empty object, if the class is not in a package.

Properties

Property	Purpose
`Name` read only	Name of the package associated with this `meta.package` object
`ClassList` read only	List of classes that are scoped to this package. An object array of `meta.class` objects.
`Classes` read only Use `ClassList` instead	List of classes that are scoped to this package. A cell array of `meta.class` objects.
`FunctionList` read only	List of functions that are scoped to this package. An object array of function handles.
`Functions` read only Use `FunctionList` instead	List of functions that are scoped to this package. A cell array of function handles.
`PackageList` read only	List of packages that are scoped to this package. An object array of `meta.package` objects.
`Packages` read only Use `PackageList` instead	List of packages that are scoped to this package. A cell array of `meta.package` objects.

Property	Purpose
`ContainingPackage` read only	A `meta.package` object describing the package within which this package is contained, or an empty object if this package is not nested.

Methods

Method	Purpose
`fromName`	Static method returns a `meta.package` object for a specified package name.
`getAllPackages`	Static method returns a cell array of `meta.package` objects representing all top-level packages.

More About

- "Getting Information About Classes and Objects"

See Also

`meta.property` | `meta.event` | `meta.class` | `meta.method`

meta.abstractDetails

Package: meta

Find abstract methods and properties

Syntax

```
meta.abstractDetails(ClassName)
meta.abstractDetails(mc)
absMembers = meta.abstractDetails( ___ )
```

Description

`meta.abstractDetails(ClassName)` displays a list of abstract methods and properties for the class with name `ClassName`. Use the fully specified name for classes in packages. MATLAB displays all public and protected abstract methods and properties, including those declared `Hidden`.

`meta.abstractDetails(mc)` displays a list of abstract methods and properties for the class represented by the `meta.class` object `mc`.

`absMembers = meta.abstractDetails( ___ )` returns an array of the metaclass objects corresponding to the abstract members of the class, and can include any of the input arguments in previous syntaxes. If the class has both abstract methods and abstract properties, `absMembers` is a heterogeneous array of class `meta.MetaData` containing `meta.method` and `meta.property` objects.

A class can be abstract without defining any abstract methods or properties if it declares the `Abstract` class attribute. In this case, `meta.abstractDetails` returns no abstract members for that class, but the class is abstract. See “Determine If a Class Is Abstract” for more information.

Input Arguments

`ClassName`

Name of the class specified as a character string (for example, `'MyClass'`)

`mc`

`meta.class` object representing the class (for example, `?MyClass`)

Output Arguments

`absMembers`

Array of metaclass objects representing abstract class members

Examples

Display Abstract Member Names

Define the class, `AbsBase`, with an abstract property:

```
classdef AbsBase
   properties (Abstract)
      Prop1
   end
   methods(Abstract)
      result = methodOne(obj)
      output = methodTwo(obj)
   end
end
```

Pass the class name (`AbsBase`) as a string:

```
meta.abstractDetails('AbsBase')
```

`meta.abstractDetails` displays the names of the abstract properties and methods defined in the class `AbsBase`.

```
Abstract methods for class AbsBase:
    methodTwo   % defined in AbsBase
    methodOne   % defined in AbsBase

Abstract properties for class AbsBase:
```

```
    Prop1   % defined in AbsBase
```

Return Abstract Member Metaclass Objects

Pass a `meta.class` object representing the `AbsBase` class and return the metaclass objects for the abstract members. Use the definition of the `AbsBase` class from the previous example.

```
mc = ?AbsBase;
absMembers = meta.abstractDetails(mc);
```

`absMembers` is a heterogeneous array containing a `meta.property` object for the `Prop1` abstract property and `meta.method` objects for the `methodOne` and `methodTwo` abstract methods.

List the names of the metaclass objects.

```
for k=1:length(absMembers)
   disp(absMembers(k).Name)
end
```

```
methodTwo
methodOne
Prop1
```

Find Inherited Abstract Members

Derive the `SubAbsBase` class from `AbsBase`, which is defined in a previous example.

```
classdef SubAbsBase < AbsBase
   properties
      SubProp = 1;
   end
   methods
      function result = methodOne(obj)
         result = obj.SubProp + 1;
      end
   end
end
```

Display the names of the abstract members inherited by `SubAbsBase`.

```
meta.abstractDetails('SubAbsBase')
```

```
Abstract methods for class SubAbsBase:
```

```
    methodTwo   % defined in AbsBase

Abstract properties for class SubAbsBase:
    Prop1   % defined in AbsBase
```

To make `SubAbsBase` a concrete class, you need to implement concrete versions of `methodTwo` and `Prop1` in the subclass.

More About

- "Abstract Classes"
- "Getting Information About Classes and Objects"

See Also

`meta.class` | `meta.class.fromName`

meta.package.fromName

Return `meta.package` object for specified package

Syntax

```
mpkg = meta.package.fromName('pkgname')
```

Description

`mpkg = meta.package.fromName('pkgname')` is a static method that returns the `meta.package` object associated with the named package. If `pkgname` is a nested package, then you must provide the fully qualified name (e.g., `'pkgname1.pkgname2'`).

Examples

List the classes in the `event` package:

```
mev = meta.package.fromName('event');
for k=1:length(mev.Classes)
   disp(mev.Classes{k}.Name)
end
event.EventData
event.PropertyEvent
event.listener
event.proplistener
```

See Also

`meta.package` | `meta.package.getAllPackages`

meta.package.getAllPackages

Get all top-level packages

Syntax

```
P = meta.package.getAllPackages
```

Description

`P = meta.package.getAllPackages` is a static method that returns a cell array of `meta.package` objects representing all the top-level packages that are visible on the MATLAB path or defined as top-level built-in packages. You can access subpackages using the `Packages` property of each `meta.package` object.

Note that the time required to find all the packages on the path might be excessively long in some cases. You should therefore avoid using this method in any code where execution time is a consideration. `getAllPackages` is generally intended for interactive use only.

See Also

`meta.package` | `meta.package.fromName`

meta.property

Describe property of MATLAB class

Description

The `meta.property` class provides information about the properties of MATLAB classes. The read/write properties of the `meta.property` class correspond to property attributes and are specified only from within your class definitions.

You can query the read-only properties of the `meta.property` object to obtain information that is specified syntactically by the class (for example, to obtain the function handle of a property's set access method).

You cannot instantiate a `meta.property` object directly. Obtain a `meta.property` object from the `meta.class` `PropertyList` property, which contains an array of `meta.property` objects, one for each class property. For example, replace *`ClassName`* with the name of the class whose properties you want to query:

```
mco = ?ClassName;
plist = mco.PropertyList;
plist(1).Name; % name of first property
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Name` read only	Name of the property.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`AbortSet` attribute, default = `false`	If `true`, and this property belongs to a handle class, then MATLAB does not set the property value if the new

Property	Purpose
	value is the same as the current value. This prevents the triggering of property `PreSet` and `PostSet` events. See “Listen for Changes to Property Values”
`Abstract` attribute, default = `false`	If `true`, the property has no implementation, but a concrete subclass must redefine this property without `Abstract` being set to `true`. • Abstract properties cannot define set or get access methods. See “Property Access Methods” • Abstract properties cannot define initial values. “Assigning a Default Value” • All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. • `Abstract=true` should be used with the class attribute `Sealed=false` (the default).
`Access`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only (not subclasses) List of classes that have get and set access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. Use `Access` to set both `SetAccess` and `GetAccess` to the same value. Query the values of `SetAccess` and `GetAccess` directly (not `Access`).

Property	Purpose
`Constant` attribute, default = `false`	Set to `true` if you want only one value for this property in all instances of the class. • Subclasses inherit constant properties, but cannot change them. • `Constant` properties cannot be `Dependent` • `SetAccess` is ignored. See “Properties with Constant Values”
`DefaultValue`	Property default value (if specified in class definition). See also `HasDefault` property. Abstract, dependent and dynamic properties cannot specify default values.
`DefiningClass`	The `meta.class` object representing the class that defines this property.
`GetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only List classes that have get access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. See “Control Access to Class Members”
`Dependent` attribute, default = `false`	If `false`, property value is stored in object. If `true`, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See “Property Get Methods”

Property	Purpose
`Transient` attribute, default = `false`	If `true`, property value is not saved when object is saved to a file. See "Save and Load Process" for more about saving objects.
`GetMethod` read only	Function handle of the get method associated with this property. Empty if there is no get method specified. See "Property Get Methods"
`GetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See "Property-Set and Query Events"
`HasDefault`	Property contains a boolean value indicating if the property defines a default value. Test `HasDefault` before querying the `DefaultValue` property to avoid a `MATLAB:class:NoDefaultDefined` error.
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only `immutable` — property can be set only in the constructor. See "Mutable and Immutable Properties" List classes that have set access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. See "Control Access to Class Members"

Property	Purpose
`SetMethod` read only	Function handle of the set method associated with this property. Empty if there is no set method specified. See "Property Set Methods"
`SetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See "Property-Set and Query Events"

Events

See "Listen for Changes to Property Values" for information on using property events.

Event Name	Purpose
`PreGet`	Event occurs just before property is queried.
`PostGet`	Event occurs just after property has been queried
`PreSet`	Event occurs just before this property is modified
`PostSet`	Event occurs just after this property has been modified

More About

- "Properties"
- "Getting Information About Classes and Objects"

See Also

`meta.method` | `meta.class` | `meta.event` | `metaclass`

metaclass

Obtain `meta.class` object

Syntax

```
mc = metaclass(object)
mc = ?ClassName
```

Description

`mc = metaclass(object)` returns the `meta.class` object for the class of `object`. The `object` input argument can be a scalar or an array of objects. However, `metaclass` always returns a scalar `meta.class` object.

`mc = ?ClassName` returns the `meta.class` object for the class with name, *`ClassName`*. The `?` operator works only with a class name, not an object.

If you pass a class name as a string to the `metaclass` function, it returns the `meta.class` object for the `char` class. Use the `?` operator or the `meta.class.fromName` method to obtain the `meta.class` object from a class name. Use this method if you want to pass the class name in a string variable.

Examples

Return the `meta.class` object for an instance of the `MException` class:

```
obj = MException('Msg:ID','MsgTxt');
mc = metaclass(obj);
```

Use the `?` operator to get the `meta.class` object for the `matlab.mixin.SetGet` class:

```
mc = ?matlab.mixin.SetGet;
```

See Also

`meta.class` | `meta.class.fromName`

methods

Class method names

Syntax

```
methods('classname')
methods(...,'-full')
m = methods(...)
```

Description

`methods('classname')` displays the names of the methods for the class *`classname`*. If *`classname`* is a MATLAB or Java class, then `methods` displays only public methods, including those methods inherited from superclasses.

`methods(...,'-full')` displays a full description of the methods, including inheritance information and, for MATLAB and Java methods, method attributes and signatures. `methods` does not remove duplicate method names with different signatures. Do not use this option with classes defined before MATLAB 7.6.

`m = methods(...)` returns the method names in a cell array of strings.

`methods` is also a MATLAB class-definition keyword. See `classdef` for more information on class-definition keywords.

This function does not show generic methods from classes based on the Microsoft .NET Framework. Use your product documentation to get information on generic methods.

Examples

Retrieve the names of the static methods in class `MException`:

```
methods('MException')
```

```
Methods for class MException:
```

```
addCause     getReport      ne           throw
eq           isequal        rethrow      throwAsCaller

Static methods:

last
```

See Also

`methodsview` | `properties` | `events` | `what` | `which`

Introduced before R2006a

methodsview

View class methods

Syntax

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

Description

`methodsview packagename.classname` displays information about the methods in the class, `classname`. If the class is in a package, include `packagename`. If `classname` is a MATLAB or Java class, `methodsview` lists only public methods, including those methods inherited from superclasses.

`methodsview classname` displays information describing the class `classname`.

`methodsview(object)` displays information about the methods of the class of `object`.

`methodsview` creates a window that displays the methods defined in the specified class. `methodsview` provides additional information like arguments, returned values, and superclasses. It also includes method qualifiers (for example, `abstract` or `synchronized`) and possible exceptions thrown.

Examples

List information on all methods in the `java.awt.MenuItem` class:

```
methodsview java.awt.MenuItem
```

MATLAB displays this information in a new window.

See Also

`methods` | `import` | `class` | `javaArray`

Introduced before R2006a

mex

Build MEX-function from C/C++ or Fortran source code

Syntax

```
mex filenames
mex option1 ... optionN filenames

mex -setup lang
```

Description

`mex filenames` compiles and links one or more C, C++, or Fortran source files into a binary MEX-file, callable from MATLAB. filenames specify the source files. Also builds executable files for standalone MATLAB engine and MAT-file applications.

MATLAB automatically selects a compiler, if installed, based on the language of the filenames arguments.

`mex option1 ... optionN filenames` builds with the specified build options. The option1 ... optionN arguments supplement or override the default `mex` build configuration.

`mex -setup lang` selects a compiler for the given lang. Use this option when you want to change the default compiler for the given language.

Examples

Build C MEX-File

Build a single C program, `yprime.c`, into a MEX-file.

Each example is based on MEX examples in the *`matlabroot`*`/extern/examples` subfolders. To build the example code, copy the source file to a writable folder on your path, such as `c:\work`. Set the current folder to `c:\work`.

```
[s,msg,msgid] = mkdir('c:\work');
if (isempty(msgid))
    mkdir('c:\work')
end
cd c:\work
```

Copy the source code, `yprime.c`.

```
copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'),'.','f');
```

Build the MEX-file.

```
mex yprime.c
```

```
Building with 'Microsoft Visual C++ 2010 (C)'.
MEX completed successfully.
```

The output displays information specific to your compiler.

Test.

```
T=1;
Y=1:4;
yprime(T,Y)
```

```
ans =
     2.0000    8.9685    4.0000    -1.0947
```

Display Detailed Build and Troubleshooting Information

To display the compile and link commands and other information useful for troubleshooting, use verbose mode.

```
mex -v -compatibleArrayDims yprime.c
```

The output displays information specific to your platform and compiler.

Override Default Compiler Switch Option

Build the `yprime.c` MEX-file by appending the value `-Wall` to the existing compiler flag. Because the value includes a space character, you must delineate the string; the character you use depends on the platform.

At the MATLAB prompt, use MATLAB single quotes (').

```
mex -v COMPFLAGS='$COMPFLAGS -Wall' yprime.c
```

At the Windows Command Prompt, use double quotes (").

```
mex -v COMPFLAGS="$COMPFLAGS -Wall" yprime.c
```

At the shell command line on Mac and Linux, use single quotes (').

```
mex -v CFLAGS='$CFLAGS -Wall' yprime.c
```

Build MEX-File from Multiple Source Files

The MEX-file example, `fulltosparse`, consists of two Fortran source files, `loadsparse.F` and `fulltosparse.F`.

To run this example, you need a supported Fortran compiler installed on your system.

Copy the source files to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook',...
  'loadsparse.F'),'.','f');
copyfile(fullfile(matlabroot,'extern','examples','refbook',...
  'fulltosparse.F'),'.','f');
```

Build the `fulltosparse` MEX-file.

```
mex -largeArrayDims fulltosparse.F loadsparse.F
```

The MEX-file name is `fulltosparse` because `fulltosparse.F` is the first file on the command line.

Test.

```
full = eye(5);
spar = fulltosparse(full)
```

```
spar =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Preview Build Commands

To preview the build command details without executing the commands, use the `-n` option.

```
mex -n yprime.c
```

The output displays information specific to your platform and compiler.

Create and Link to Separate Object Files

You can link to object files that you compile separately from your source MEX-files.

The MEX-file example, `fulltosparse`, consists of two Fortran source files, `loadsparse.F` and `fulltosparse.F`. The `fulltosparse` file is the gateway routine (contains the `mexFunction` subroutine) and `loadsparse` contains the computational routine.

To run this example, you need a supported Fortran compiler installed on your system.

Copy the computational subroutine to your current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook',...
  'loadsparse.F'),'.','f');
```

Compile the subroutine and place the object file in a separate folder, `c:\objfiles`.

```
mkdir c:\objfiles
mex -largeArrayDims -c -outdir c:\objfiles loadsparse.F
```

Copy the gateway subroutine to your current folder. Compile and link with the `loadsparse` object file.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook',...
  'fulltosparse.F'),'.','f');
mex -largeArrayDims fulltosparse.F c:\objfiles\loadsparse.obj
```

Specify Path to Include File

Use the `-I` option to specify the path to include the MATLAB LAPACK library subroutines for handling complex number routines. To use these subroutines, your MEX-file must access the header file, `fort.h`.

Copy the `matrixDivideComplex.c` example to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook',...
  'matrixDivideComplex.c'),'.','f');
```

Create the `-I` argument by concatenating `'-I'` with the path to `fort.h` file.

```
ipath = ['-I' fullfile(matlabroot,'extern','examples','refbook')];
```

Create variables for the names and paths to the LAPACK library file and the file, `fort.c`, containing the complex number handling routines.

```
lapacklib = fullfile(matlabroot, ...
    'extern','lib',computer('arch'),'microsoft','libmwlapack.lib');
fortfile = fullfile(matlabroot,'extern','examples',...
    'refbook','fort.c');
```

Build the MEX-file.

```
mex('-v','-largeArrayDims',ipath, ...
    'matrixDivideComplex.c',fortfile,lapacklib)
```

Specify Path to Library File

Build the `matrixDivide.c` example on a Windows platform.

Use the `-L` and `-l` options to specify the `libmwlapack.lib` library, located in the folder, *`matlabroot`*`\extern\lib\`*`arch`*`\microsoft`.

Copy the `matrixDivide.c` example to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','matrixDivide.c'),...
    '.','f');
```

Capture the value of *`matlabroot`*.

```
matlabroot
```

```
ans =

C:\Program Files\MATLAB\R2014a
```

Capture the value of *`arch`*.

```
arch
```

```
ans =

win64
```

To build the MEX-file, copy the values of *`matlabroot`* and *`arch`* into the `mex` command, as shown in the following statement.

```
mex -largeArrayDims '-LC:\Program Files\MATLAB\R2014a\extern\lib\win64\microsoft'...
   -llibmwlapack matrixDivide.c
```

You must use the ' characters because \Program Files in the path includes a space.

Define Directive

Define the character to use between strings in a matrix.

The MATLAB example, mxcreatecharmatrixfromstr.c, uses a #define symbol, SPACE_PADDING, to determine what character to use between strings in a matrix. To set the value, build the MEX-file with the -D option.

Copy the mxcreatecharmatrixfromstr.c example to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','mx',...
  'mxcreatecharmatrixfromstr.c'),'.','f');
```

Set the SPACE_PADDING directive to add a space between strings.

```
mex mxcreatecharmatrixfromstr.c -DSPACE_PADDING
```

Build Engine Application

Copy the engwindemo.c engine example to the current folder.

```
copyfile(fullfile(matlabroot,...
  'extern','examples','eng_mat','engwindemo.c'),'.','f');
```

```
mex -client engine engwindemo.c
```

Run the example.

```
!engwindemo
```

Select C Compiler

```
mex -setup
```

MATLAB displays the options for your version and system based on the list of Supported and Compatible Compilers.

- "Table of MEX-File Source Code Files"

Input Arguments

`filenames` — One or more file names

string

One or more file names, including name and file extension, specified as a string. If the file is not in the current folder, specify the full path to the file. File names can be any combination of:

- C, C++, or Fortran language source files
- Simulink S-function files
- object files
- library files

The first source code file listed in filenames is the name of the binary MEX-file. To override this naming convention, use the `'-output'` option.

Data Types: `char`

`option1 ... optionN` — One or more build options

strings corresponding to valid option flags

One or more build options, specified as one of these values. Options can appear in any order on any platform, except where indicated.

Option	Description
`@rspfile`	Uses Windows RSP file. An RSP file is a text file containing command-line options.
`-c`	Compiles an object file only. Does not build a binary MEX-file.
`-client engine`	Build engine application.
`-compatibleArrayDims` (default) `-largeArrayDims`	Links with the specified MATLAB array-handling API. • `-compatibleArrayDims` — Uses the MATLAB Version 7.2 array-handling API, which limits arrays to 2^{31}-1 elements. Default option. • `-largeArrayDims` — Uses the MATLAB large-array-handling API. This API handles arrays with more than 2^{31}-1 elements. Must use this option when calling LAPACK or BLAS functions. In verbose mode (`-v` option), if you do not specify either the `-compatibleArrayDims` or the `-largeArrayDims`

Option	Description
	option, MATLAB displays a message showing the default option.

`-Dsymbolname`
`-Dsymbolname=symbolvalu`
`-Usymbolname`
	The `-D` options define C preprocessor macros. Equivalent to the following in the source file:
	• `#define symbolname`
	• `#define symbolname symbolvalue`
	
	The `-U` option removes any initial definition of the C preprocessor macro, `symbolname`. Inverse of the `-D` option.
	
	Do not add a space between **D** or **U** and *`symbolname`*. Do not add spaces around the **=** sign.
	
	Example: "Define Directive" on page 1-5109

`-f filepath`	To build engine applications, use the `-client engine` option. Specifies name and location of the `mex` configuration file. Overrides the default compiler selection. For information about using a non-default compiler, see "Changing Default Compiler". *`filepath`* is the name and full path of the configuration file, specified as a string.
`-g`	Adds symbolic information and disables optimizing built object code. Use for debugging. To debug with optimization, add the `-O` option.
`-h[elp]`	Displays help for `mex`. Use from an operating system prompt.
`-Ipathname`	Adds *`pathname`* to the list of folders to search for `#include` files. Do not add a space between **I** and *`pathname`*. Example: "Specify Path to Include File" on page 1-5107

Option	Description
`-l`*`libname`* `-L`*`libfolder`* `-l`*`libname`*	Links with object library *`libname`* in (optional) *`libfolder`*. MATLAB expands *`libname`* to: • *`libname`*`.lib` or `lib`*`libname`*`.lib` — Windows systems • `lib`*`libname`*`.dylib` — Mac systems • `lib`*`libname`*`.so` — Linux systems If used, the `-L` option must precede the `-l` option. When using the `-L` option on Linux or Mac systems, you also must set the runtime library path, as explained in "Setting Run-Time Library Path". Do not add a space between **l** and *`libname`* or between **L** and *`libfolder`*. Specify the `-l` option with the lowercase letter **L**. Example: "Specify Path to Library File" on page 1-5108

`-largeArrayDims`
`-compatibleArrayDims`
(default)
	Links with the specified MATLAB array-handling API.
	
	• `-compatibleArrayDims` — Uses the MATLAB Version 7.2 array-handling API, which limits arrays to 2^{31}-1 elements. Default option.
	• `-largeArrayDims` — Uses the MATLAB large-array-handling API. This API handles arrays with more than 2^{31}-1 elements. Must use this option when calling LAPACK or BLAS functions.
	
	In verbose mode (`-v` option), if you do not specify either the `-compatibleArrayDims` or the `-largeArrayDims` option, MATLAB displays a message showing the default option.

Option	Description
`-n`	Displays, but does not execute, commands that `mex` would execute. Example: "Preview Build Commands" on page 1-5106
`-O`	Optimizes the object code. Use this option to compile with optimization. Optimization is enabled by default. Optimization is disabled when the `-g` option appears without the `-O` option. Specify this option with the capital letter **O**.
`-outdir` *`dirname`*	Places all output files in folder *`dirname`*. Example: "Create and Link to Separate Object Files" on page 1-5107
`-output` *`mexname`*	Overrides the default MEX-file naming mechanism. Creates binary MEX-file named *`mexname`* with the appropriate MEX-file extension.
`-setup` *`lang`*	Change the default compiler to build *`lang`* language MEX-files. When you use this option, all other command-line options are ignored.
`-silent`	Suppresses informational messages. The `mex` function still reports errors and warnings, even when you specify `-silent`.
`-U`*`symbolname`*	Removes any initial definition of the C preprocessor macro *`symbolname`*. (Inverse of the `-D` option.) Do not add a space between **U** and *`symbolname`*.
`-v`	Builds in verbose mode. Displays values for internal variables after all command-line arguments are considered. Displays each compile and link step fully evaluated. Use for troubleshooting compiler setup problems. Example: "Display Detailed Build and Troubleshooting Information" on page 1-5105

Option	Description
varname=varvalue	Overrides default setting for variable *varname*. This option is processed after all command-line arguments are considered. Example: "Override Default Compiler Switch Option" on page 1-5105.

`lang` — Language
`C` (default) | `C++` | `CPP` | `Fortran`

Language, specified as one of these values.

`C`	C compilers, including C++.
`C++` or `CPP`	C++ compilers.
`Fortran`	Fortran compilers.

More About

Tips

- You can run `mex` from:
  - MATLAB Command Window
  - Windows system prompt
  - Mac Terminal
  - Linux shell

  For command-line usage outside of MATLAB, the `mex` program is located in the folder specified by `[matlabroot '/bin']`.
- The MEX-file has a platform-dependent extension. You can place binary MEX-files for different platforms in the same folder.

MEX-File Platform-Dependent Extension

Platform	Binary MEX-File Extension
Linux (64-bit)	`mexa64`

Platform	Binary MEX-File Extension
Apple Mac (64-bit)	`mexmaci64`
Microsoft Windows (32-bit)	`mexw32`
Windows (64-bit)	`mexw64`

To identify the MEX-file extension, use the `mexext` function.

- To use `mex` to build executable files for standalone MATLAB engine and MAT-file applications, use the `-client engine` option.

- "Build MEX-File"
- "Changing Default Compiler"
- Supported and Compatible Compilers

See Also

"C/C++ Matrix Library API" | "Fortran Matrix Library API" | `clear` | `computer` | `dbmex` | `inmem` | `loadlibrary` | `mex.getCompilerConfigurations` | `mexext` | `pcode` | `prefdir` | `system`

Introduced before R2006a

mex.getCompilerConfigurations

Get compiler configuration information for building MEX-files

Syntax

```
cc = mex.getCompilerConfigurations
cc = mex.getCompilerConfigurations(lang)
cc = mex.getCompilerConfigurations(lang,list)
```

Description

`cc = mex.getCompilerConfigurations` returns an object cc containing information about the default compiler configurations used by the `mex` command. There is one configuration for each supported language.

`cc = mex.getCompilerConfigurations(lang)` returns an array of objects for the given language, lang.

`cc = mex.getCompilerConfigurations(lang,list)` returns information about the set of configurations, list.

Examples

Display Information for C Compiler

```
myCCompiler = mex.getCompilerConfigurations('C','Selected')
```

```
myCCompiler =

  CompilerConfiguration with properties:

            Name: 'Microsoft Visual C++ 2010 (C)'
    Manufacturer: 'Microsoft'
        Language: 'C'
         Version: '10.0'
        Location: 'c:\Program Files (x86)\Microsoft Visual Studio 10.0'
       ShortName: 'MSVC100'
```

```
         Priority: 'A'
          Details: [1x1 mex.CompilerConfigurationDetails]
       LinkerName: 'link'
    LinkerVersion: ''
           MexOpt: 'C:\Users\auser\AppData\Roaming\MathWorks\MATLAB\R2014a\mex_C_win64
```

MATLAB displays information depending on your architecture and your version of MATLAB.

Display Number of Supported C Compilers

```
cLanguageCC = mex.getCompilerConfigurations('C','Supported');
length(cLanguageCC)
```

```
ans =
    10
```

The number of compilers for your version of MATLAB might be different.

Input Arguments

`lang` — Language

`'Any'` (default) | `'C'` | `'C++'` | `'CPP'` | `'Fortran'`

Language, specified as one of these values.

`'Any'`	All supported languages. This is the default value.
`'C'`	All C compiler configurations, including C++ configurations.
`'C++'` or `'CPP'`	All C++ compiler configurations.
`'Fortran'`	All Fortran compiler configurations.

`list` — Set of configurations

`'Selected'` (default) | `'Installed'` | `'Supported'`

Set of configurations, specified as one of these values.

`'Selected'`	The default compiler for each language.
`'Installed'`	All supported compilers `mex` finds installed on your system.

`'Supported'`	All compilers supported in the current release.

Output Arguments

`cc` — Compiler information

mex.CompilerConfiguration object or array of objects

Compiler information, specified as a `mex.CompilerConfiguration` object or array of `mex.CompilerConfiguration` objects. The `mex.CompilerConfiguration` class contains the following read-only properties.

Property	Purpose
`Name`	Compiler name.
`ShortName`	Character string used to identify options file for the compiler.
`Manufacturer`	Name of the manufacturer of the compiler.
`Language`	Compiler language.
`Version`	(Windows platforms only) Version of the compiler.
`Location`	(Windows platforms only) Folder where compiler is installed.
`Details`	More read-only properties about the compiler configuration. These properties might differ across compilers, platforms, and releases of MATLAB.
`LinkerName`	Linker name.
`LinkerVersion`	(Windows platforms only) Version of the linker.
`MexOpt`	Name and full path to options file.
`Priority`	The priority of this compiler.

More About

- Supported and Compatible Compilers

See Also

`mex`

MException class

Capture error information

Description

Any MATLAB code that detects an error and throws an exception must construct an `MException` object. This class contains retrievable information about errors. MATLAB can throw either predefined exceptions or exceptions that you construct.

Construction

`ME = MException(msgID,msgtext)` captures information about a specific error and stores it in the `MException` object, ME. The `MException` object is constructed with a message identifier, `msgID`, and an error message string, `msgtext`.

`ME = MException(msgID,msgtext,A1,...,An)` allows formatting of the error message string using text or numeric values, `A1,...,An`, to replace conversion specifiers in msgtext at run time.

Input Arguments

`msgID` — Identifier for error

string

Identifier for the error, specified as a string. Use the message identifier with exception handling to better identify the source of the error or to control a selected subset of the exceptions in your program.

The message identifier includes a `component` and `mneumonic`. The identifier must always contain a colon and follows a simple format: `component:mnemonic`. The `component` and `mneumonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No white space characters can appear anywhere in msgID. For more information, see “Message Identifiers”.

Example: `'MyComponent:noSuchVariable'`

`msgtext` — Information about cause of error

string

Information about the cause of the error and how you might correct it, specified as a string. To format the string, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see "Formatting Strings".

Example: `'Error opening file.'`

Example: `'Error on line %d.'`

`A1,...,An` — Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array. This input argument provides the values that correspond to and replace the conversion specifiers in `msgtext`.

Properties

`identifier` — Unique identifier of error

string

String that uniquely identifies the error, specified as a string by the `msgID` input argument. This property is read only. For more information, see "Message Identifiers".

`message` — Error message

string

String that contains the error message that is displayed when MATLAB throws the exception, specified by the `msgtext` and `A1,...,An` input arguments. This property is read only. For more information, see "Text of the Error Message".

`stack` — Stack trace information

array of structures

Structure array that contains stack trace information including the file name (`file`), function name (`name`), and line number (`line`) where MATLAB throws the exception. If the error occurs in a called function, the `stack` property also contains the file name,

function name, and line number for each of the called functions. MATLAB generates the stack only when it throws the exception.

`stack` is an N-by-1 `struct` array, where N represents the depth of the call stack. This property is read only. For more information, see "The Call Stack".

`cause` — Cause of the exception

cell array of `MException` objects

Cell array of `MException` objects that caused MATLAB to create this exception. Use the `addCause` method to add an exception to the `cause` field of the another exception. For more information, see "The Cause Array".

Methods

addCause	Record additional causes of exception
getReport	Get error message for exception
last	Return last uncaught exception
rethrow	Rethrow previously caught exception
throw	Throw exception
throwAsCaller	Throw exception as if occurs within calling function

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create MException Object

```
msgID = 'myComponent:inputError';
msgtext = 'Input does not have the expected format.';

ME = MException(msgID,msgtext)
```

```
ME =

  MException with properties:

    identifier: 'myComponent:inputError'
       message: 'Input does not have the expected format.'
         cause: {}
         stack: [0x1 struct]
```

Create MException with Formatted Error Message

```
msgID = 'MATLAB:test';
msgtext = 'There are %d errors on this page';
A1 = 10;

ME = MException(msgID,msgtext,A1)
```

```
ME =

  MException with properties:

    identifier: 'MATLAB:test'
       message: 'There are 10 errors on this page'
         cause: {}
         stack: [0x1 struct]
```

Create and Throw MException Object

Throw an exception if an input variable name does not exist in the workspace.

```
str = input('Type a variable name: ','s');
if ~exist(str,'var')
```

```
    ME = MException('MyComponent:noSuchVariable', ...
        'Variable %s not found',str);
    throw(ME)
end
```

At the input prompt, enter any variable that does not exist in your workspace. For example, enter `notaVariable`.

```
Variable notaVariable not found
```

Since `notVariable` doesn't exist in your workspace, MATLAB creates an `MException` object, and then throws it.

Use try/catch to Capture Exception

Catch the exception generated by calling a nonexistent function, `notaFunction`. If the function is not defined, issue a warning and assign the output a value of 0.

```
try
    a = notaFunction(5,6);
catch ME
    if strcmp(ME.identifier,'MATLAB:UndefinedFunction')
        warning('Function is undefined.  Assigning a value of 0.');
    else
        rethrow(ME)
    end
end
```

```
Warning: Function is undefined.  Assigning a value of 0.
```

By itself, the call to `notaFunction` results in an error. Using `try` and `catch`, this code catches the undefined function exception and repackages it as a warning, allowing MATLAB to continue executing subsequent commands. If the caught exception has a different error identifier, MATLAB rethrows the exception.

- " Capture Information About Exceptions"

See Also

`MException.throw` | `MException.rethrow` | `MException.throwAsCaller` | `MException.addCause` | `MException.getReport` | `MException.last` | `assert` | `dbstack` | `error` | `try, catch`

addCause

Class: MException

Record additional causes of exception

Syntax

```
baseException = addCause(baseException,causeException)
```

Description

`baseException = addCause(baseException,causeException)` modifies the existing `MException` object baseException by appending causeException to its `cause` property. Catching the resulting exception in a `try/catch` statement makes the base exception, along with all of the appended cause records, available to help diagnose the error.

Input Arguments

`baseException` — Primary exception

`MException` object

Primary exception containing the primary cause and location of an error, specified as an `MException` object.

`causeException` — Related exception

`MException` object

Related exception containing the cause and location of an error related to `baseException`, specified as an `MException` object.

Examples

Add Causes to Exception

Create an array, and an index into it with a logical array.

```
A = [13 42; 7 20];
idx = [1 0 1; 0 1 0];
```

Create an exception that provides general information about an error. Test the index array and add exceptions with more detailed information about the source of the failure.

```
try
    A(idx);
catch
    msgID = 'MYFUN:BadIndex';
    msg = 'Unable to index into array.';
    baseException = MException(msgID,msg);

    try
        assert(islogical(idx),'MYFUN:notLogical',...
            'Indexing array is not logical.')
    catch causeException
        baseException = addCause(baseException,causeException);
    end

    if any(size(idx) > size(A))
        msgID = 'MYFUN:incorrectSize';
        msg = 'Indexing array is too large.';
        causeException2 = MException(msgID,msg);
        baseException = addCause(baseException,causeException2);
    end
    throw(baseException)
end
```

```
Unable to index into array.

Caused by:
    Indexing array is not logical.
    Indexing array is too large.
```

Examine the `baseException` object.

```
baseException
```

```
baseException =

  MException with properties:

    identifier: 'MYFUN:BadIndex'
       message: 'Unable to index into array.'
```

```
        cause: {2x1 cell}
        stack: [0x1 struct]
```

The value of the `cause` property is a `2x1` cell array.

Examine the first cause of the exception.

```
baseException.cause{1}
```

```
ans =

  MException with properties:

    identifier: 'MYFUN:notLogical'
       message: 'Indexing array is not logical.'
         cause: {0x1 cell}
         stack: [0x1 struct]
```

Examine the second cause of the exception.

```
baseException.cause{2}
```

```
ans =

  MException with properties:

    identifier: 'MYFUN:incorrectSize'
       message: 'Indexing array is too large.'
         cause: {}
         stack: [0x1 struct]
```

See Also

`MException` | `MException.throw` | `MException.rethrow` | `MException.throwAsCaller` | `MException.getReport` | `MException.last` | `assert` | `error` | `try, catch`

getReport

Class: MException

Get error message for exception

Syntax

```
msgString = getReport(exception)
msgString = getReport(exception,type)
msgString = getReport(exception,type,'hyperlinks',hlink)
```

Description

`msgString = getReport(exception)` gets the error message for an exception and returns it as a formatted string, `msgString`. The message string is the value of the `message` property of the `MException` object, `exception`. It is the same string that MATLAB displays when it throws the exception.

`msgString = getReport(exception,type)` returns the error message using the indicated level of detail, specified by `type`.

`msgString = getReport(exception,type,'hyperlinks',hlink)` uses the value of `hlink` to determine whether to include active hyperlinks to the failing lines of code within the error message.

Input Arguments

`exception` — Exception object that provides error message

`MException` object

Exception object that provides the error message, specified as a scalar `MException` object.

`type` — Detail indicator of message string

`'extended'` (default) | `'basic'`

Detail indicator of the message string returned, specified as `'extended'` or `'basic'`.

`type` Value	`msgString` Detail Level
`'extended'` (default)	`msgString` includes the line number, error message, cause, and stack summary. To display the proper stack, MATLAB first must throw an exception.
`'basic'`	`msgString` includes the error message.

`hlink` — Hyperlink indicator of message string

`'on'` (default) | `'off'` | `'default'`

Hyperlink indicator of the message string that includes active hyperlinks to the failing lines of code, specified as `'on'`, `'off'`, or `'default'`.

`hlink` Value	Action
`'on'`	Display hyperlinks to failing lines of code.
`'off'`	Do not display hyperlinks to failing lines of code.
`'default'`	Use the default for the Command Window to determine whether or not to use hyperlinks in the error message.

Examples

Get Error Message from Exception

Cause MATLAB to throw an exception.

```
plus
```

```
Error using +
Not enough input arguments.
```

Get the error message from the exception.

```
exception = MException.last;
msgString = getReport(exception)
```

```
msgString =

Error using +
```

```
Not enough input arguments.
```

Specify Detail Level in Error Message

In a file in your current working folder, create the following function in `testFunc.m`.

```
function a = testFunc
try
    a = notaFunction(5,6);
catch a

end
```

Since the function, `notaFunction`, does not exist, `testFunc` returns an `mException` object.

At the command prompt, call `testFunc` and get the error message.

```
m = testFunc;
msgString = getReport(m)
```

```
msgString =

Undefined function 'notaFunction' for input arguments of type 'double'.

Error in testFunc (line 3)
    a = notaFunction(5,6);
```

Specify that the error message only contains the error message and not the stack information.

```
msgString = getReport(m,'basic')
```

```
msgString =

Undefined function 'notaFunction' for input arguments of type 'double'.
```

Turn Off Hyperlinks in Error Message

Cause MATLAB to throw an exception.

```
try
    surf
catch exception
end
```

Get the error message from the exception.

```
msgString = getReport(exception)
```

```
msgString =

Error using surf (line 49)
Not enough input arguments.
```

Get the error message without active hyperlinks to `surf.m`.

```
msgString = getReport(exception,'extended','hyperlinks','off')
```

```
msgString =

Error using surf (line 49)
Not enough input arguments.
```

See Also

`MException.addCause` | `MException.rethrow` | `MException.last` | `MException.throw` | `MException.throwAsCaller` | `MException` | `assert` | `error` | `try, catch`

MException.last

Class: MException

Return last uncaught exception

Syntax

```
exception = MException.last
MException.last('reset')
```

Description

`exception = MException.last` returns the contents of the most recently thrown, uncaught `MException` object. `MException.last` is not set if the last exception is caught by a `try/catch` statement. `MException.last` is a static method of the `MException` class.

`MException.last('reset')` clears the properties of the exception returned from `MException.last`. It sets the `MException` `identifier` and `message` properties to an empty string, the `stack` property to a 0-by-1 structure, and the `cause` property to an empty cell array.

Tips

- Use `MException.last` only from the Command Window, not within a function.

Examples

Get Last Uncaught Exception

Cause MATLAB to throw, but not catch, an exception.

```
A = 25;
A(2)
```

```
Index exceeds matrix dimensions.
```

Get the uncaught exception.

```
exception = MException.last
```

```
exception =

  MException with properties:

    identifier: 'MATLAB:badsubscript'
       message: 'Index exceeds matrix dimensions.'
         cause: {}
         stack: [0x1 struct]
```

Reset Last Uncaught Exception

Call the `surf` function with no input arguments.

```
surf
```

```
Error using surf (line 49)
Not enough input arguments.
```

Get the uncaught exception.

```
exception = MException.last
```

```
exception =

  MException with properties:

    identifier: 'MATLAB:nargchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         cause: {0x1 cell}
         stack: [1x1 struct]
```

Get the last, uncaught exception.

```
MException.last('reset')
exception = MException.last
```

```
exception =

  MException with properties:
```

```
    identifier: ''
       message: ''
         cause: {0x1 cell}
         stack: [0x1 struct]
```

See Also

MException.addCause | MException.getReport | MException.rethrow | MException.throw | MException.throwAsCaller | MException | assert | error | try, catch

rethrow

Class: MException

Rethrow previously caught exception

Syntax

```
rethrow(exception)
```

Description

`rethrow(exception)` rethrows a previously caught exception, `exception`. MATLAB typically responds to errors by terminating the currently running program. However, you can use a `try/catch` block to catch the exception. This interrupts the program termination so you can execute your own error handling procedures. End the `catch` block with a `rethrow` statement to terminate the program and redisplay the exception.

`rethrow` handles the stack trace differently from `error`, `assert`, and `throw`. Instead of creating the stack from where MATLAB executes the method, `rethrow` preserves the original exception information and enables you to retrace the source of the original error.

Input Arguments

`exception` — Exception containing cause and location of error

`MException` object

Exception containing the cause and location of an error, specified as a scalar `MException` object.

Examples

Catch and Rethrow Exception

Cause MATLAB to throw an error by calling `surf` with no inputs. Catch the exception, display the error identifier, and rethrow the exception.

```
try
    surf
catch ME
    disp(['ID: ' ME.identifier])
    rethrow(ME)
end
```

```
ID: MATLAB:nargchk:notEnoughInputs
Error using surf (line 49)
Not enough input arguments.
```

Compare Behavior of throw and rethrow

Create a function, `combineArrays`, in your working folder.

```
function C = combineArrays(A,B)
try
    C = catAlongDim1(A,B);       % Line 3
catch exception
    throw(exception)             % Line 5
end
end

function V = catAlongDim1(V1,V2)
V = cat(1,V1,V2);                % Line 10
end
```

Call the `combineArrays` function with arrays of different sizes.

```
A = 1:5;
B = 1:4;

combineArrays(A,B)
```

```
Error using combineArrays (line 5)
Dimensions of matrices being concatenated are not consistent.
```

The stack refers to line 5 where MATLAB throws the exception.

Replace `throw(exception)` with `rethrow(exception)` on line 5 of the `combineArrays` function, and call the function again.

```
combineArrays(A,B)
```

```
Error using cat
```

```
Dimensions of matrices being concatenated are not consistent.

Error in combineArrays>catAlongDim1 (line 10)
V = cat(1,V1,V2);               % Line 10

Error in combineArrays (line 3)
    C = catAlongDim1(A,B);      % Line 3
```

The `rethrow` method maintains the original stack and indicates the error is on line 3.

See Also

`MException.addCause` | `MException.getReport` | `MException.last` | `MException.throw` | `MException.throwAsCaller` | `MException` | `assert` | `error` | `try, catch`

throw

Class: MException

Throw exception

Syntax

```
throw(exception)
```

Description

`throw(exception)` throws an exception based on the information contained in the `MException` object, exception. The exception terminates the currently running function and returns control either to the keyboard or to an enclosing `catch` block. When you throw an exception from outside a `try/catch` statement, MATLAB displays the error message in the Command Window.

The `throw` method, unlike the `throwAsCaller` and `rethrow` methods, creates the stack trace from the location where MATLAB executes the method.

You can access the `MException` object via a `try/catch` statement or the `MException.last` method.

Input Arguments

`exception` — Exception containing cause and location of error

`MException` object

Exception containing the cause and location of an error, specified as a scalar `MException` object.

Examples

Create and Throw MException Object

Throw an exception if an input variable name does not exist in the workspace.

```
str = input('Type a variable name: ','s');
if ~exist(str,'var')
    ME = MException('MyComponent:noSuchVariable', ...
        'Variable %s not found',str);
    throw(ME)
end
```

At the input prompt, enter any variable that does not exist in your workspace. For example, enter `notaVariable`.

```
Variable notaVariable not found
```

Since `notVariable` doesn't exist in your workspace, MATLAB creates an `MException` object, and then throws it.

Compare Behavior of throw and rethrow

Create a function, `combineArrays`, in your working folder.

```
function C = combineArrays(A,B)
try
    C = catAlongDim1(A,B);       % Line 3
catch exception
    throw(exception)             % Line 5
end
end

function V = catAlongDim1(V1,V2)
V = cat(1,V1,V2);                % Line 10
end
```

Call the `combineArrays` function with arrays of different sizes.

```
A = 1:5;
B = 1:4;

combineArrays(A,B)
```

```
Error using combineArrays (line 5)
Dimensions of matrices being concatenated are not consistent.
```

The stack refers to line 5 where MATLAB throws the exception.

Replace `throw(exception)` with `rethrow(exception)` on line 5 of the `combineArrays` function, and call the function again.

```
combineArrays(A,B)

Error using cat
Dimensions of matrices being concatenated are not consistent.

Error in combineArrays>catAlongDim1 (line 10)
V = cat(1,V1,V2);               % Line 10

Error in combineArrays (line 3)
    C = catAlongDim1(A,B);      % Line 3
```

The `rethrow` method maintains the original stack and indicates the error is on line 3.

See Also

`MException` | `MException.rethrow` | `MException.throwAsCaller` | `MException.addCause` | `MException.getReport` | `MException.last` | | `error` | `try, catch`

throwAsCaller

Class: MException

Throw exception as if occurs within calling function

Syntax

```
throwAsCaller(exception)
```

Description

`throwAsCaller(exception)` throws an exception as if it occurs within the calling function. The exception terminates the currently running function and returns control to the keyboard or an enclosing `catch` block. When you throw an exception from outside a `try/catch` statement, MATLAB displays the error message in the Command Window.

You can access the `MException` object via a `try/catch` statement or the `MException.last` method.

In some cases, it is more informative for the error to point to the location in the calling function that results in the exception rather than pointing to the function that actually throws the exception. You can use `throwAsCaller` to simplify the error display.

Input Arguments

`exception` — Exception containing cause and location of error

`MException` object

Exception containing the cause and location of an error, specified as a scalar `MException` object.

Examples

Compare Behavior of throw and throwAsCaller

Create a function, `sayHello`, in your working folder.

```
function sayHello(N)
checkInput(N)
str = ['Hello, ' N '!'];
disp(str)

function checkInput(N)
if ~ischar(N)
    ME = MException('sayHello:inputError','Input must be char.');
    throw(ME)
end
```

At the command prompt, call the function with a numeric input.

```
sayHello(42)
```

```
Error using sayHello>checkInput (line 9)
Input must be char.

Error in sayHello (line 2)
checkInput(N)
```

The top of the stack refers to line 9 because this is where MATLAB throws the exception. After the initial stack frame, MATLAB displays information from the calling function.

Replace `throw(ME)` with `throwAsCaller(ME)` in line 9 of `sayHello.m` and call the function again.

```
sayHello(42)
```

```
Error using sayHello (line 2)
Input must be char.
```

The top of the stack refers to line 2 because that is the location of the error in the calling function.

See Also

`MException` | `MException.rethrow` | `MException.throwAsCaller` | `MException.addCause` | `MException.getReport` | `MException.last` | | `assert` | `error` | `try, catch`

mexext

Binary MEX-file-name extension

Syntax

```
ext = mexext
extlist = mexext('all')
```

Description

`ext = mexext` returns the file-name extension for the current platform.

`extlist = mexext('all')` returns the extensions for all platforms.

Examples

Display File Extension for Your Computer

Find the MEX-file extension for the system you are currently working on.

```
ext = mexext
```

```
ext =
   mexw32
```

Your results reflect your system.

Find File Extension for Specific Platform

Find the MEX-file extension for the Apple Macintosh systems.

Get the list for supported platforms.

```
extlist = mexext('all');
```

The `mex` function identifies a platform by its `arch` value, which is the output of the `computer('arch')` command. For Macintosh platforms, the value is `maci64`.

Search the `arch` field in the results, `extlist`, for `'maci64'`, and display the corresponding `ext` field.

```
for k=1:length(extlist)
   if strcmp(extlist(k).arch, 'maci64')
   disp(sprintf('Arch: %s      Ext: %s', ...
            extlist(k).arch, extlist(k).ext))
end, end
```

```
Arch: maci64       Ext: mexmaci64
```

The file extension is `mexmaci64`.

Output Arguments

`ext` — File-name extension

`mexa64 | mexmaci64 | mexw32 | mexw64`

File-name extension for MEX-file, returned as one of these values.

MEX-File Platform-Dependent Extension

Platform	Binary MEX-File Extension
Linux (64-bit)	`mexa64`
Apple Mac (64-bit)	`mexmaci64`
Microsoft Windows (32-bit)	`mexw32`
Windows (64-bit)	`mexw64`

`extlist` — All file-name extensions

structure

All file-name extensions, returned as a structure with these fields:

`arch` — Platform

string

Platform, specified as a string. The name of the platform is the output of the `computer('arch')` command.

`ext` — File extension

string

File extension, specified as a string.

More About

Tips

- To use the MEX-file-name extension in makefiles or scripts outside MATLAB, type one of the following from the system command prompt. The script is located in the *`matlabroot`*`\bin` folder.
  - `mexext.bat`—Windows platform.
  - `mexext.sh`—UNIX platform.

  For example, the following commands are in a GNU® makefile.

  ```
  ext = $(shell mexext)
  yprime.$(ext) : yprime.c
    mex yprime.c
  ```
- MATLAB continues to execute a MEX-file with a `.dll` extension, but future versions of MATLAB will not support this extension.

See Also

`computer` | `mex`

Introduced before R2006a

mfilename

File name of currently running function

Syntax

```
mfilename
p = mfilename('fullpath')
c = mfilename('class')
```

Description

`mfilename` returns a string containing the file name of the most recently invoked function. When called from within the file, it returns the name of that file. This allows a function to determine its name, even if the file name has been changed.

`p = mfilename('fullpath')` returns the full path and name of the file in which the call occurs, not including the filename extension.

`c = mfilename('class')` in a method, returns the class of the method, not including the leading `@` sign. If called from a nonmethod, it yields the empty string.

More About

Tips

If `mfilename` is called with any argument other than the above two, it behaves as if it were called with no argument.

When called from the command line, `mfilename` returns an empty string.

To get the names of the callers of a MATLAB function file, use `dbstack` with an output argument.

See Also

`dbstack` | `nargin` | `function` | `nargout` | `inputname`

Introduced before R2006a

mget

Class: FTP

Download files from FTP server

Syntax

```
mget(ftpobj,contents)
mget(ftpobj,contents,target)
```

Description

`mget(ftpobj,contents)` retrieves the file or folder specified by contents from an FTP server into the MATLAB current folder.

`mget(ftpobj,contents,target)` retrieves the file or folder into the local folder specified by target, which includes an absolute or relative path.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`contents`

String enclosed in single quotation marks that specifies either a file name or a folder name. Can include a wildcard character (**`*`**).

`target`

String enclosed in single quotation marks that specifies the absolute or relative path of the local folder to contain the downloaded contents.

Examples

Connect to an FTP server and retrieve the file `README` into the current MATLAB folder:

```
mw = ftp('ftp.mathworks.com');
mget(mw, 'README');
close(mw);
```

See Also

`mput` | `cd` | `ftp`

Introduced before R2006a

milliseconds

Duration in milliseconds

Syntax

```
MS = milliseconds(X)
```

Description

`MS = milliseconds(X)` converts `duration` array `X` to a `double` array `MS`, where each element of `MS` is equivalent to the number of milliseconds in the corresponding element of `X`.

Examples

Convert Durations to Numeric Array of Milliseconds

Create a `duration` array.

```
X = minutes(2) + seconds(1:3)
```

```
X =
   2.0167 mins   2.0333 mins     2.05 mins
```

Convert each duration in `X` to a number of milliseconds.

```
MS = milliseconds(X)
```

```
MS =
      121000       122000       123000
```

`MS` is a `double` array.

Input Arguments

X — Input array

duration array

Input array, specified as a duration array.

See Also

`duration`

Introduced in R2015a

min

Smallest elements in array

Syntax

```
M = min(A)
M = min(A,[],dim)
[M,I] = min( ___ )

C = min(A,B)

___ = min( ___ ,nanflag)
```

Description

`M = min(A)` returns the smallest elements of `A`.

- If `A` is a vector, then `min(A)` returns the smallest element of `A`.
- If `A` is a matrix, then `min(A)` is a row vector containing the minimum value of each column.
- If `A` is a multidimensional array, then `min(A)` operates along the first array dimension whose size does not equal `1`, treating the elements as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same. If `A` is an empty array with first dimension `0`, then `min(A)` returns an empty array of the same size as `A`.

`M = min(A,[],dim)` returns the smallest elements along dimension `dim`. For example, if `A` is a matrix, then `min(A,[],2)` is a column vector containing the minimum value of each row.

`[M,I] = min( ___ )` finds the indices of the minimum values of `A` and returns them in output vector `I`, using any of the input arguments in the previous syntaxes. If the minimum value occurs more than once, then `min` returns the index corresponding to the first occurrence.

`C = min(A,B)` returns an array the same size as `A` and `B` with the smallest elements taken from `A` or `B`.

Either the dimensions of A and B are the same, or one can be a scalar.

`___ = min( ___ ,nanflag)` specifies whether to include or omit `NaN` values in the calculation for any of the previous syntaxes. For the single input case, to specify `nanflag` without specifying `dim`, use `min(A,[],nanflag)`. For example, `min(A,[],'includenan')` includes all `NaN` values in `A` while `min(A,[],'omitnan')` ignores them.

Examples

Smallest Vector Element

Create a vector and compute its smallest element.

```
A = [23 42 37 15 52];
M = min(A)
```

```
M =

    15
```

Smallest Complex Element

Create a complex vector and compute its smallest element, that is, the element with the smallest magnitude.

```
A = [-2+2i 4+i -1-3i];
min(A)
```

```
ans =

  -2.0000 + 2.0000i
```

Smallest Element in Each Matrix Column

Create a matrix and compute the smallest element in each column.

```
A = [2 8 4; 7 3 9]
```

```
A =
```

```
     2     8     4
     7     3     9

M = min(A)

M =

     2     3     4
```

Smallest Element in Each Matrix Row

Create a matrix and compute the smallest element in each row.

```
A = [1.7 1.2 1.5; 1.3 1.6 1.99]

A =

    1.7000    1.2000    1.5000
    1.3000    1.6000    1.9900

M = min(A,[],2)

M =

    1.2000
    1.3000
```

Smallest Element Indices

Create a matrix A and compute the smallest elements in each column as well as the row indices of A in which they appear.

```
A = [1 9 -2; 8 4 -5]

A =

     1     9    -2
     8     4    -5
```

```
[M,I] = min(A)

M =

     1     4    -5


I =

     1     2     2
```

Smallest Element Comparison

Create a matrix and return the smallest value between each of its elements compared to a scalar.

```
A = [1 7 3; 6 2 9]

A =

     1     7     3
     6     2     9


B = 5;
C = min(A,B)

C =

     1     5     3
     5     2     5
```

Smallest Element in Matrix

Create a matrix `A` and use its column representation, `A(:)`, to find the value and index of the smallest element.

```
A = [8 2 4; 7 3 9]
```

```
A =

     8     2     4
     7     3     9
```

```
A(:)

ans =

     8
     7
     2
     3
     4
     9
```

```
[M,I] = min(A(:))

M =

     2

I =

     3
```

`I` is the index of `A(:)` containing the smallest element.

Now, use the `ind2sub` function to extract the row and column indices of `A` corresponding to the smallest element.

```
[I_row, I_col] = ind2sub(size(A),I)

I_row =

     1

I_col =
```

```
     2
```

If you need only the minimum value of `A` and not its index, call the `min` function twice.

```
M = min(min(A))
```

```
M =

     2
```

Smallest Element Involving `NaN`

Create a vector and compute its minimum, excluding `NaN` values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
M = min(A,[],'omitnan')
```

```
M =

   -2.9500
```

`min(A)` will also produce this result since `'omitnan'` is the default option.

Use the `'includenan'` flag to return `NaN`.

```
M = min(A,[],'includenan')
```

```
M =

   NaN
```

Input Arguments

`A` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

- If `A` is complex, then `min(A)` returns the complex number with the smallest magnitude. If magnitudes are equal, then `min(A)` returns the value with the smallest magnitude and the smallest phase angle.
- If `A` is a scalar, then `min(A)` returns `A`.
- If `A` is a 0-by-0 empty array, then `min(A)` is as well.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

`dim` — Dimension to operate along

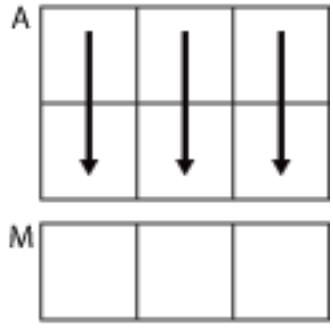
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `min` returns an empty array with the same dimension sizes.

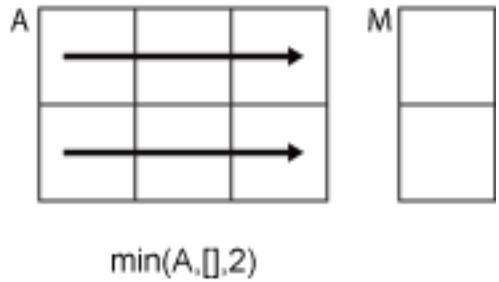
Consider a two-dimensional input array, `A`:

- If `dim = 1`, then `min(A,[],1)` returns a row vector containing the smallest element in each column.



min(A,[],1)

- If `dim = 2`, then `min(A,[],2)` returns a column vector containing the smallest element in each row.



`min` returns `A` if `dim` is greater than `ndims(A)`.

`B` — Additional input array

scalar | vector | matrix | multidimensional array

Additional input array, specified as a scalar, vector, matrix, or multidimensional array.

- The dimensions of `A` and `B` must match, or one can be a scalar.
- `A` and `B` can be the same data type or one can be `double` with the other `single`, `duration`, or any integer data type.
- If `A` and `B` are ordinal categorical arrays, they must have the same sets of categories with the same order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

`nanflag` — `NaN` condition

'`omitnan`' (default) | '`includenan`'

`NaN` condition, specified as one of these values:

- '`omitnan`' — Ignore all `NaN` values in the input.
- '`includenan`' — Include the `NaN` values in the input for the calculation.

The `min` function does not support the `nanflag` option for `datetime`, `duration`, or `categorical` arrays.

Data Types: `char`

Output Arguments

`M` — Minimum values

scalar | vector | matrix | multidimensional array

Minimum values, returned as a scalar, vector, matrix, or multidimensional array. `size(M,dim)` is `1`, while the sizes of all other dimensions match the size of the corresponding dimension in `A`, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `M` is an empty array with the same dimensions as `A`.

`I` — Index to minimum values of `A`

scalar | vector | matrix | multidimensional array

Index to minimum values of `A`, returned as a scalar, vector, matrix, or multidimensional array. `I` is the same size as `M`. If the smallest element occurs more than once, then `I` contains the index to the first occurrence of the value.

`C` — Minimum elements from `A` or `B`

scalar | vector | matrix | multidimensional array

Minimum elements from `A` or `B`, returned as a scalar, vector, matrix, or multidimensional array.

The size of `C` depends on the sizes of `A` and `B`:

- If `A` and `B` are arrays of the same size, then the size of `C` matches the size of `A` and `B`.
- If either `A` or `B` is a scalar, then the size of `C` matches the size of the nonscalar input array.
- If either `A` or `B` is an empty array with the other a scalar, then `C` is an empty array.

The data type of `C` depends on the data types of `A` and `B`:

- If `A` and `B` are the same data type, then `C` matches the data type of `A` and `B`.
- If either `A` or `B` is `single`, then `C` is `single`.
- If either `A` or `B` is an integer data type with the other a scalar `double`, then `C` assumes the integer data type.

More About

- “Matrix Indexing”

See Also

`max` | `mean` | `median` | `sort`

MinimizeCommandWindow

Minimize size of Automation server window

Syntax

IDL Method Signature

```
HRESULT MinimizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MinimizeCommandWindow
```

MATLAB Client

```
MinimizeCommandWindow(h)
```

Description

`MinimizeCommandWindow(h)` minimizes the window for the server attached to handle `h`, and makes it inactive.

If the server window was already in a minimized state, `MinimizeCommandWindow` does nothing.

Examples

From a Visual Basic .NET client, modify the size of the command window in a MATLAB Automation server.

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow
```

```
'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

MaximizeCommandWindow

Introduced before R2006a

minres

Minimum residual method

Syntax

```
x = minres(A,b)
minres(A,b,tol)
minres(A,b,tol,maxit)
minres(A,b,tol,maxit,M)
minres(A,b,tol,maxit,M1,M2)
minres(A,b,tol,maxit,M1,M2,x0)
[x,flag] = minres(A,b,...)
[x,flag,relres] = minres(A,b,...)
[x,flag,relres,iter] = minres(A,b,...)
[x,flag,relres,iter,resvec] = minres(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)
```

Description

`x = minres(A,b)` attempts to find a minimum norm residual solution `x` to the system of linear equations `A*x=b`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `minres` converges, a message to that effect is displayed. If `minres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`minres(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `minres` uses the default, `1e-6`.

`minres(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `minres` uses the default, `min(n,20)`.

`minres(A,b,tol,maxit,M)` and `minres(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = inv(sqrt(M))*y`. If `M` is `[]` then `minres` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x)` returns `M\x`.

`minres(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `minres` uses the default, an all-zero vector.

`[x,flag] = minres(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`minres` converged to the desired tolerance `tol` within `maxit` iterations.
1	`minres` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`minres` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `minres` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = minres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = minres(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = minres(A,b,...)` also returns a vector of estimates of the `minres` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)` also returns a vector of estimates of the Conjugate Gradients residual norms at each iteration.

Examples

Using minres with a Matrix Input

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M1 = spdiags(4*on,0,n,n);

x = minres(A,b,tol,maxit,M1);
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Using minres with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_minres` that

- Calls `minres` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_minres` are available to `afun`.

The following shows the code for `run_minres`:

```
function x1 = run_minres
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M = spdiags(4*on,0,n,n);
x1 = minres(@afun,b,tol,maxit,M);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
       end
end
```

When you enter

```
x1=run_minres;
```

MATLAB software displays the message

```
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Using minres instead of pcg

Use a symmetric indefinite matrix that fails with `pcg`.

```
A = diag([20:-1:1, -1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
```

displays the following message:

```
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, `minres` can handle the indefinite matrix `A`.

```
x = minres(A,b,1e-6,40);
minres converged at iteration 39 to a solution with relative
residual 1.3e-007
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

See Also

`bicg` | `bicgstab` | `cgs` | `function_handle` | `gmres` | `ichol` | `lsqr` | `mldivide` | `pcg` | `qmr` | `symmlq`

Introduced before R2006a

minus, -

Subtraction

Syntax

```
C = A - B
C = minus(A,B)
```

Description

`C = A - B` subtracts array `B` from array `A` and returns the result in `C`.

`C = minus(A,B)` is an alternate way to execute `A - B`, but is rarely used. It enables operator overloading for classes.

Examples

Subtract Scalar from Array

Create an array, `A`, and subtract a scalar value from it.

```
A = [2 1; 3 5];
C = A - 2
```

```
C =

     0    -1
     1     3
```

The scalar is subtracted from each entry of `A`.

Subtract Two Arrays

Create two arrays, `A` and `B`, and subtract the second, `B`, from the first, `A`.

```
A = [1 0; 2 4];
B = [5 9; 2 1];
```

```
C = A - B

C =

    -4    -9
     0     3
```

The elements of `B` are subtracted from the corresponding elements of `A`.

Use the syntax `-C` to negate the elements of `C`.

```
-C

ans =

     4     9
     0    -3
```

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `A` can be a numeric array, logical array, character array, datetime array, duration array, or calendar duration array. Inputs `A` and `B` must be the same size unless one is a scalar. You can add a scalar value to any other value.

If one input is a datetime array, duration array, or calendar duration array, then numeric values in the other input are treated as a number of 24-hour days.

If one input is a datetime array, then the other input also can be a date string or a cell array containing date strings.

Complex Number Support: Yes

B — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `B` can be a numeric array, logical array, character array, datetime array, duration array, or calendar duration array. Inputs `A` and `B` must be the same size unless one is a scalar. You can add a scalar value to any other value.

If one input is a datetime array, duration array, or calendar duration array, then numeric values in the other input are treated as a number of 24-hour days.

If one input is a datetime array, then the other input also can be a date string or a cell array containing date strings.
Complex Number Support: Yes

More About

- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`diff` | `plus` | `uminus`

minute

Minute number

Syntax

```
m = minute(t)
```

Description

`m = minute(t)` returns the minute numbers of the datetime values in `t`. The `m` output is a `double` array the same size as `t` and contains integer values from 0 to 59.

The `minute` function returns the minute numbers of datetime values. To assign minute values to a datetime array, `t`, use `t.Minute` and modify the `Minute` property.

Examples

Find Minute Number of Datetime Values

```
t1 = datetime('now');
t = t1 + minutes(2:4)
```

```
t =

   23-Feb-2015 10:05:11   23-Feb-2015 10:06:11   23-Feb-2015 10:07:11
```

```
m = minute(t)
```

```
m =

     5     6     7
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

datetime Properties | `hms` | `hour` | `second` | `timeofday`

Introduced in R2014b

minutes

Duration in minutes

Syntax

```
M = minutes(X)
```

Description

`M = minutes(X)` returns an array of minutes equivalent to the values in `X`.

- If `X` is a numeric array, then `M` is a `duration` array in units of minutes.
- If `X` is a `duration` array, then `M` is a `double` array with each element equal to the number of minutes in the corresponding element of X.

The `minutes` function converts between `duration` and `double` values. To display a duration in units of minutes, set its `Format` property to `'m'`.

Examples

Create Duration Array of Minutes

```
X = magic(4);
M = minutes(X)
```

```
M =

   16 mins    2 mins    3 mins   13 mins
    5 mins   11 mins   10 mins    8 mins
    9 mins    7 mins    6 mins   12 mins
    4 mins   14 mins   15 mins     1 min
```

Convert Durations to Numeric Array of Minutes

Create a `duration` array.

```
X = hours(2:10:38) + minutes(30)

X =

    2.5 hrs   12.5 hrs   22.5 hrs   32.5 hrs
```

Convert each duration in `X` to a number of minutes.

```
M = minutes(X)

M =

         150         750        1350        1950
```

View the data type of `M`.

```
whos M
  Name      Size             Bytes  Class     Attributes

  M         1x4                 32  double
```

Input Arguments

X — Input array

numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

See Also

`duration`

Introduced in R2014b

mislocked

Determine if function is locked in memory

Syntax

```
mislocked
mislocked(fun)
```

Description

`mislocked` by itself returns logical `1` (`true`) if the currently running function is locked in memory, and logical `0` (`false`) otherwise. Functions that are locked cannot be removed with the `clear` function unless you first unlock them using the `munlock` function. You can use locking on functions that reside in MATLAB `.m` files or `.mex` files.

`mislocked(fun)` returns logical `1` (`true`) if the function named *`fun`* is locked in memory, and logical `0` (`false`) otherwise.

See Also

`mlock` | `munlock` | `inmem`

Introduced before R2006a

mkdir

Make new folder

Syntax

```
mkdir('folderName')
mkdir('parentFolder','folderName')
status = mkdir( ___ )
[status,message,messageid] = mkdir( ___ )
```

Description

`mkdir('folderName')` creates the folder `folderName`, where `folderName` can be an absolute or a relative path.

`mkdir('parentFolder','folderName')` creates the folder `folderName` in `parentFolder`, where `parentFolder` is an absolute or relative path. If `parentFolder` does not exist, MATLAB attempts to create it. See the Tips section.

`status = mkdir( ___ )` creates the specified folder. When the operation is successful, it returns a `status` of logical 1. When the operation is unsuccessful, it returns logical 0.

`[status,message,messageid] = mkdir( ___ )` creates the specified folder, and returns the status, message string, and MATLAB message ID. The value given to `status` is logical `1` for success, and logical `0` for error.

Examples

Creating a Subfolder in the Current Folder

Create a subfolder called `newdir` in the current folder:

```
mkdir('newdir')
```

Creating a Subfolder in the Specified Parent Folder

Create a subfolder called `newFolder` in the folder `testdata`, using a relative path, where `newFolder` is at the same level as the current folder:

```
mkdir('../testdata','newFolder')
```

Returning Status When Creating a Folder

In this example, the first attempt to create `newFolder` succeeds, returning a status of `1`, and no error or warning message or message identifier:

```
[s, mess, messid] = mkdir('../testdata', 'newFolder')
s =
   1
mess =
   ''
messid =
   ''
```

Attempt to create the same folder again. `mkdir` again returns a success status, and also a warning and message identifier informing you that the folder exists:

```
[s,mess,messid] = mkdir('../testdata','newFolder')
s =
   1
mess =
   Directory "newFolder" already exists.
messid =
   MATLAB:MKDIR:DirectoryExists
```

More About

Tips

If an argument specifies a path that includes one or more nonexistent folders, MATLAB attempts to create the nonexistent folder. For example, for

```
mkdir('myFolder\folder1\folder2\targetFolder')
```

if `folder1` does not exist, MATLAB creates `folder1`, creates `folder2` within `folder1`, and creates `targetFolder` within `folder2`.

- “Manage Files and Folders”

See Also

`copyfile` | `rmdir` | `cd` | `dir` | `ls` | `movefile`

Introduced before R2006a

mkdir

Class: FTP

Create folder on FTP server

Syntax

```
mkdir(ftpobj,folder)
```

Description

`mkdir(ftpobj,folder)` creates the specified folder on the FTP server associated with ftpobj.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`folder`

String enclosed in single quotation marks that specifies a path relative to the current folder on the FTP server.

Examples

Suppose that a hypothetical host, `ftp.testsite.com`, contains a folder named `testfolder`. Connect to the server and add a subfolder:

```
test=ftp('ftp.testsite.com');
mkdir(test,'testfolder/newfolder');
close(test);
```

See Also

`rmdir | dir | ftp`

Introduced before R2006a

mkpp

Make piecewise polynomial

Syntax

```
pp = mkpp(breaks,coefs)
pp = mkpp(breaks,coefs,d)
```

Description

`pp = mkpp(breaks,coefs)` builds a piecewise polynomial, `pp`, from its breaks and coefficients.

- `breaks` is a vector of length `L+1` with strictly increasing elements which represent the start and end of each of `L` intervals.
- `coefs` is an `L`-by-`k` matrix with each row `coefs(i,:)` containing the local coefficients of an order `k` polynomial on the *i*th interval, `[breaks(i),breaks(i+1)]`. That is, the polynomial `coefs(i,1)*(X-breaks(i))^(k-1) + coefs(i,2)*(X-breaks(i))^(k-2) + ... + coefs(i,k-1)*(X-breaks(i)) + coefs(i,k)`. Notice that `mkpp` shifts the polynomial in each interval down by `(X-breaks(i))`.

`pp = mkpp(breaks,coefs,d)` indicates that the piecewise polynomial `pp` is `d`-vector valued, i.e., the value of each of its coefficients is a vector of length `d`. `breaks` is an increasing vector of length `L+1`. `coefs` is a `d`-by-`L`-by-`k` array with `coefs(r,i,:)` containing the `k` coefficients of the `i`th polynomial piece of the `r`th component of the piecewise polynomial.

Use `ppval` to evaluate the piecewise polynomial at specific points. Use `unmkpp` to extract details of the piecewise polynomial.

Note: The *order* of a polynomial tells you the number of coefficients used in its description. A *k*th order polynomial has the form

$$c_1 x^{k-1} + c_2 x^{k-2} + \ldots + c_{k-1} x + c_k$$

It has k coefficients, some of which can be 0, and maximum exponent $k-1$. So the order of a polynomial is usually one greater than its degree. For example, a cubic polynomial is of order 4.

Examples

Construct and Plot Piecewise Polynomial

The first plot shows the quadratic polynomial

$$1-\left(\frac{x}{2}-1\right)^2=\frac{-x^2}{4}+x$$

shifted to the interval [-8,-4]. The second plot shows its negative

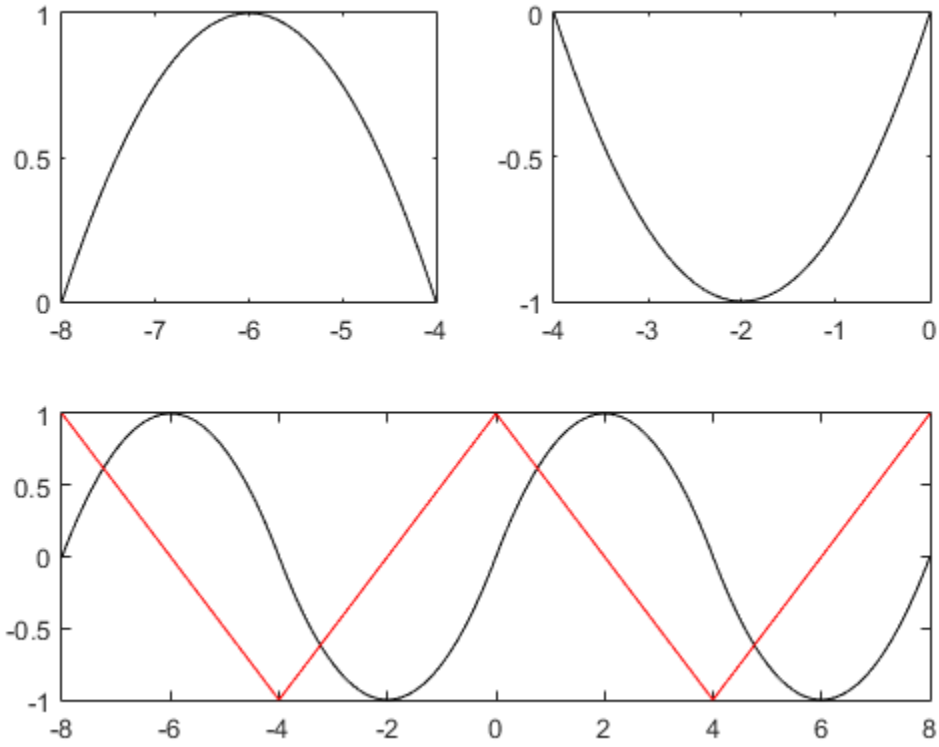
$$\left(\frac{x}{2}-1\right)^2-1=\frac{x^2}{4}-x$$

but shifted to the interval [-4,0].

The last plot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. It also shows its first derivative, which was constructed after breaking the piecewise polynomial apart using `unmkpp`.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
[breaks,coefs,l,k,d] = unmkpp(pp);
dpp = mkpp(breaks,repmat(k-1:-1:1,d*l,1).*coefs(:,1:k-1),d);
```

```
hold on, plot(xx,ppval(dpp,xx),'r-'), hold off
```



See Also

ppval | spline | unmkpp

Introduced before R2006a

mldivide, \

Solve systems of linear equations $Ax = B$ for x

Syntax

```
x = A\B
x = mldivide(A,B)
```

Description

`x = A\B` solves the system of linear equations `A*x = B`. The matrices A and B must have the same number of rows. MATLAB displays a warning message if A is badly scaled or nearly singular, but performs the calculation regardless.

- If A is a scalar, then `A\B` is equivalent to `A.\B`.
- If A is a square `n`-by-`n` matrix and B is a matrix with `n` rows, then `x = A\B` is a solution to the equation `A*x = B`, if it exists.
- If A is a rectangular `m`-by-`n` matrix with `m ~= n`, and B is a matrix with `m` rows, then A\B returns a least-squares solution to the system of equations `A*x= B`.

`x = mldivide(A,B)` is an alternative way to execute x = A\B, but is rarely used. It enables operator overloading for classes.

Examples

System of Equations

Solve a simple system of linear equations, `A*x = B`.

```
A = magic(3);
B = [15; 15; 15];
x = A\B
```

```
x =

    1.0000
    1.0000
```

```
    1.0000
```

Linear System with Singular Matrix

Solve a linear system of equations `A*x = b` involving a singular matrix, `A`.

```
A = magic(4);
b = [34; 34; 34; 34];
x = A\b
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND
1.306145e-17.

x =

    1.5000
    2.5000
   -0.5000
    0.5000
```

When `rcond` is between `0` and `eps`, MATLAB issues a nearly singular warning, but proceeds with the calculation. When working with ill-conditioned matrices, an unreliable solution can result even though the residual `(b-A*x)` is relatively small. In this particular example, the norm of the residual is zero, and an exact solution is obtained, although `rcond` is small.

When `rcond` is equal to `0`, the singular warning appears.

```
A = [1 0; 0 0];
b = [1; 1];
x = A\b
```

```
Warning: Matrix is singular to working precision.

x =

     1
   Inf
```

In this case, division by zero leads to computations with Inf and/or NaN, making the computed result unreliable.

Least-Squares Solution of Underdetermined System

Solve a system of linear equations, `A*x = b`.

```
A = [1 2 0; 0 4 3];
b = [8; 18];
x = A\b

ans =

         0
    4.0000
    0.6667
```

Linear System with Sparse Matrix

Solve a simple system of linear equations using sparse matrices.

Consider the matrix equation `A*x = B`.

```
A = sparse([0 2 0 1 0; 4 -1 -1 0 0; 0 0 0 3 -6; -2 0 0 0 2; 0 0 4 2 0]);
B = sparse([8; -1; -18; 8; 20]);
x = A\B

x =

   (1,1)       1.0000
   (2,1)       2.0000
   (3,1)       3.0000
   (4,1)       4.0000
   (5,1)       5.0000
```

Input Arguments

A — Coefficient matrix

vector | full matrix | sparse matrix

Coefficient matrix, specified as a vector, full matrix, or sparse matrix. If A has `m` rows, then B must have `m` rows.

Data Types: `single` | `double`
Complex Number Support: Yes

B — Right-hand side

vector | full matrix | sparse matrix

Right-hand side, specified as a vector, full matrix, or sparse matrix. If B has `m` rows, then A must have `m` rows.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

x — Solution

vector | full matrix | sparse matrix

Solution, returned as a vector, full matrix, or sparse matrix. If A is an `m`-by-`n` matrix and B is an `m`-by-`p` matrix, then x is an `n`-by-`p` matrix, including the case when `p==1`.

If `A` has full storage, `x` is also full. If `A` is sparse, then `x` has the same storage as `B`.

More About

Tips

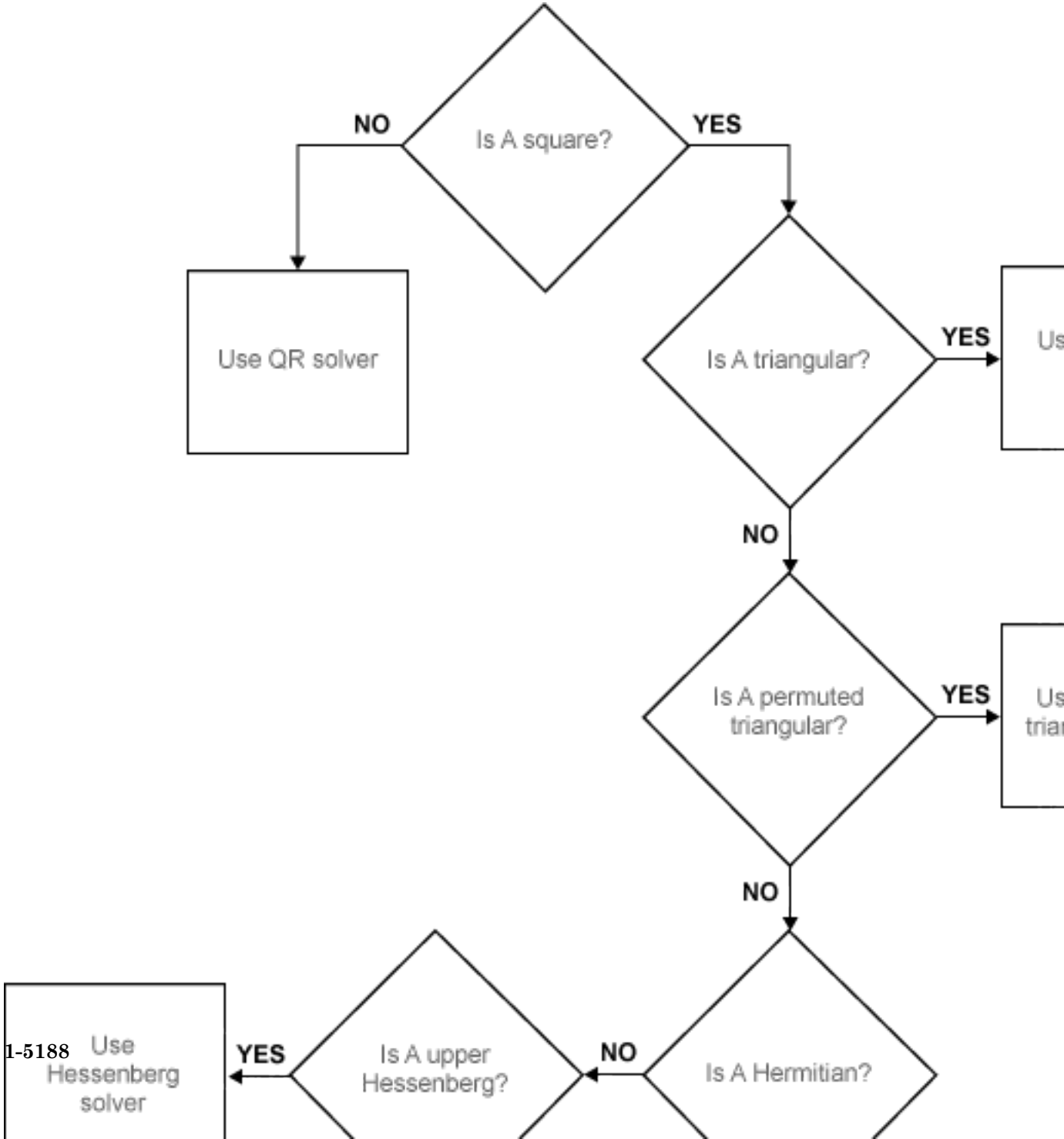
- If A is a square matrix, A\B is roughly equal to `inv(A)*B`, but MATLAB processes A\B differently and more robustly.
- If the rank of A is less than the number of columns in A, then x = A\B is not necessarily the minimum norm solution. The more computationally expensive `x = pinv(A)*B` computes the minimum norm least-squares solution.
- For full singular inputs, you can compute the least-squares solution using the function `linsolve`.

Algorithms

The versatility of `mldivide` in solving linear systems stems from its ability to take advantage of symmetries in the problem by dispatching to an appropriate solver. This approach aims to minimize computation time. The first distinction the function makes is between *full* (also called "*dense*") and *sparse* input arrays.

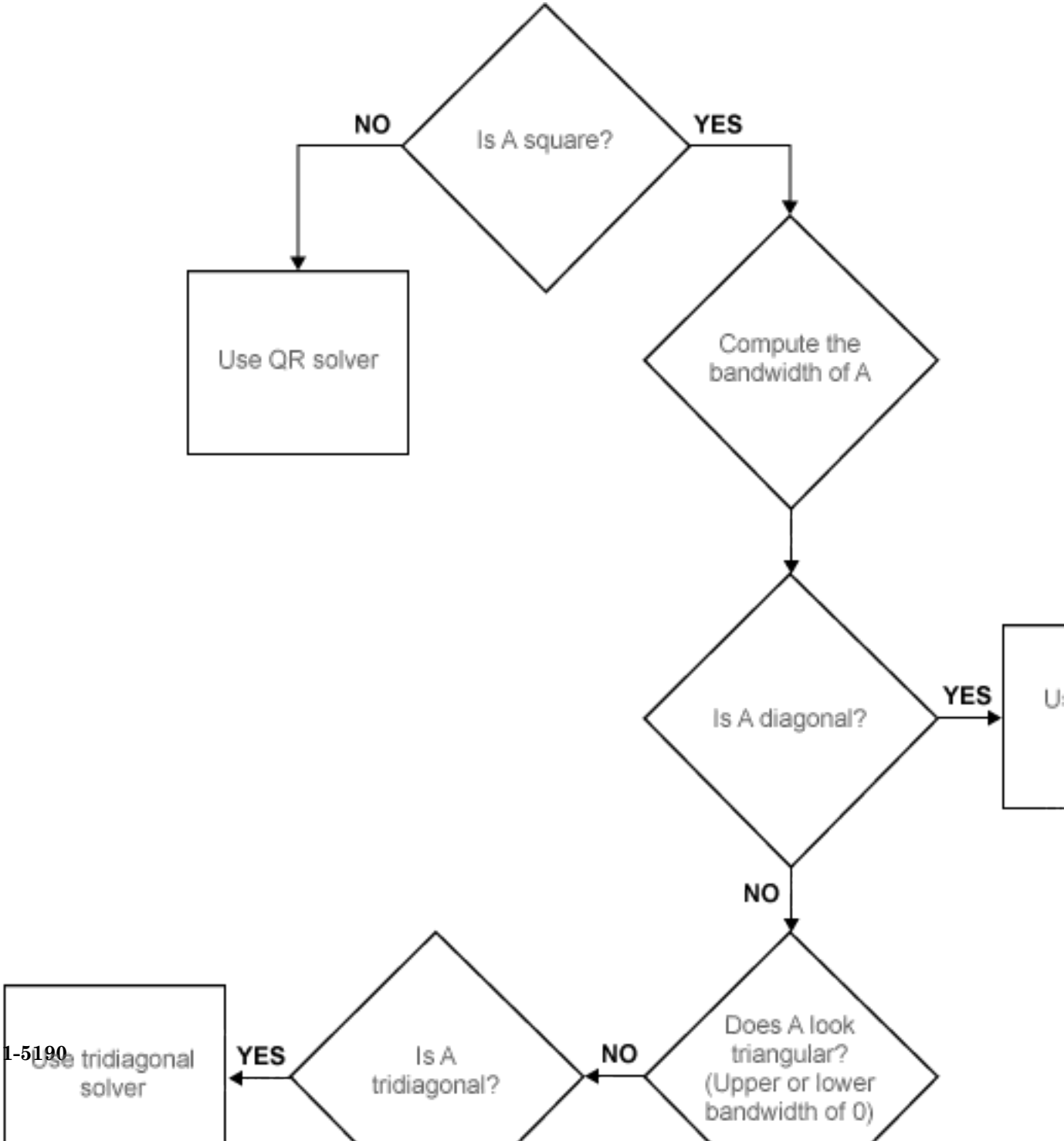
Algorithm for Full Inputs

The flow chart below shows the algorithm path when inputs `A` and `B` are **full**.


Is A square?
NO
YES
Use QR solver
Is A triangular?
YES
NO
Is A permuted triangular?
YES
NO
Is A Hermitian?
NO
Is A upper Hessenberg?
YES
Use Hessenberg solver

Algorithm for Sparse Inputs

If A is full and B is sparse then `mldivide` converts B to a full matrix and uses the full algorithm path (above) to compute a solution with full storage. If A is sparse, the storage of the solution x is the same as that of B and `mldivide` follows the algorithm path for **sparse** inputs, shown below.


Is A square?
NO
YES
Use QR solver
Compute the bandwidth of A
Is A diagonal?
YES
NO
Does A look triangular? (Upper or lower bandwidth of 0)
NO
Is A tridiagonal?
YES
Use tridiagonal solver

- “Array vs. Matrix Operations”
- “Operator Precedence”
- “Systems of Linear Equations”
- “Class Operator Implementations”

See Also

`chol` | `inv` | `ldivide` | `ldl` | `linsolve` | `lu` | `mrdivide` | `pinv` | `qr` | `rdivide`

Introduced before R2006a

mrdivide, /

Solve systems of linear equations $xA = B$ for x

Syntax

```
x = B/A
x = mrdivide(B,A)
```

Description

`x = B/A` solves the system of linear equations `x*A = B` for x. The matrices A and B must contain the same number of columns. MATLAB displays a warning message if A is badly scaled or nearly singular, but performs the calculation regardless.

- If A is a scalar, then `B/A` is equivalent to `B./A`.
- If A is a square `n`-by-`n` matrix and B is a matrix with `n` columns, then `x = B/A` is a solution to the equation `x*A = B`, if it exists.
- If A is a rectangular `m`-by-`n` matrix with `m ~= n`, and B is a matrix with `n` columns, then x = B/A returns a least-squares solution of the system of equations `x*A = B`.

`x = mrdivide(B,A)` is an alternative way to execute x = B/A, but is rarely used. It enables operator overloading for classes.

Examples

System of Equations

Solve a system of equations that has a unique solution, `x*A = B`.

```
A = [1 1 3; 2 0 4; -1 6 -1];
B = [2 19 8];
x = B/A
```

```
x =
```

```
    1.0000    2.0000    3.0000
```

Least-Squares on an Underdetermined System

Solve an underdetermined system, `x*C = D`.

```
C = [1 0; 2 0; 1 0];
D = [1 2];
x = D/C
```

```
Warning: Rank deficient, rank = 1, tol =  6.280370e-16.

x =

         0    0.5000         0
```

MATLAB issues a warning but proceeds with calculation.

Verify that `x` is not an exact solution.

```
x*C-D
```

```
ans =

     0    -2
```

Input Arguments

A — Coefficient matrix

vector | full matrix | sparse matrix

Coefficient matrix, specified as a vector, full matrix, or sparse matrix. If A has `n` columns, then B must have `n` columns.

Data Types: `single` | `double`
Complex Number Support: Yes

B — Right-hand side

vector | full matrix | sparse matrix

Right-hand side, specified as a vector, full matrix, or sparse matrix. If B has `n` columns, then A must have `n` columns.

Data Types: `single` | `double`

Complex Number Support: Yes

Output Arguments

x — Solution

vector | full matrix | sparse matrix

Solution, returned as a vector, full matrix, or sparse matrix. If A is an `m`-by-`n` matrix and B is a `p`-by-`n` matrix, then x is a `p`-by-`m` matrix.

x is sparse only if both A and B are sparse matrices.

More About

Tips

- The operators `/` and `\` are related to each other by the equation `B/A = (A'\B')'`.
- If A is a square matrix, `B/A` is roughly equal to `B*inv(A)`, but MATLAB processes B/A differently and more robustly.

- "Array vs. Matrix Operations"
- "Operator Precedence"
- "Systems of Linear Equations"

See Also

`inv` | `ldivide` | `mldivide` | `rdivide` | `transpose`

Introduced before R2006a

mlint

Check MATLAB code files for possible problems

Note: `mlint` is not recommended. Use `checkcode` instead.

Alternatives

For information on using the graphical user interface to the Code Analyzer, see "Check Code for Errors and Warnings".

Syntax

```
mlint('filename')
mlint('filename','-config=settings.txt')
mlint('filename','-config=factory')
inform=mlint('filename','-struct')
msg=mlint('filename','-string')
[inform,filepaths]=mlint('filename')
inform=mlint('filename','-id')
inform=mlint('filename','-fullpath')
inform=mlint('filename','-notok')
mlint('filename','-cyc')
mlint('filename','-codegen')
mlint('filename','-eml')
```

Description

`mlint('filename')` displays Code Analyzer messages about `filename`, where the message reports potential problems and opportunities for code improvement. The line number in the message is a hyperlink that opens the file in the Editor, scrolled to that line. If `filename` is a cell array, information is displayed for each file. For `mlint(F1,F2,F3,...)`, where each input is a character array, MATLAB software

displays information about each input file name. You cannot combine cell arrays and character arrays of file names. Note that the exact text of the `mlint` messages is subject to some change between versions.

`mlint('filename','-config=settings.txt')` overrides the default active settings file with the settings that enable or suppress messages as indicated in the specified `settings.txt` file.

Note: If used, you must specify the full path to the `settings.txt` file specified with the `-config` option.

For information about creating a `settings.txt` file, see "Save and Reuse Code Analyzer Message Settings". If you specify an invalid file, `mlint` returns a message indicating that it cannot open or read the file you specified. In that case, `mlint` uses the factory default settings.

`mlint('filename','-config=factory')` ignores all settings files and uses the factory default preference settings.

`inform=mlint('filename','-struct')` returns the information in a structure array whose length is the number of messages found. The structure has the fields that follow.

Field	Description
`message`	Message describing the suspicious construct that code analysis caught.
`line`	Vector of file line numbers to which the message refers.
`column`	Two-column array of file columns (column extents) to which the message applies. The first column of the array specifies the column in the Editor where the message begins. The second column of the array specifies the column in the Editor where the message ends. There is one row in the two-column array for each occurrence of a message.

If you specify multiple file names as input, or if you specify a cell array as input, `inform` contains a cell array of structures.

`msg=mlint('filename','-string')` returns the information as a string to the variable `msg`. If you specify multiple file names as input, or if you specify a cell array as

input, `msg` contains a string where each file's information is separated by 10 equal sign characters (=), a space, the file name, a space, and 10 equal sign characters.

If you omit the **`-struct`** or **`-string`** argument and you specify an output argument, the default behavior is **`-struct`**. If you omit the argument and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepaths]=mlint('filename')` additionally returns `filepaths`, the absolute paths to the file names, in the same order as you specified them.

`inform=mlint('filename','-id')` requests the message ID, where ID is a string of the form `ABC...`. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=mlint('filename','-fullpath')` assumes that the input file names are absolute paths, so that `mlint` does not try to locate them.

`inform=mlint('filename','-notok')` runs `mlint` for all lines in `filename`, even those lines that end with the `mlint` suppression directive, `%#ok`.

`mlint('filename','-cyc')` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`mlint('filename','-codegen')` enables code generation messages for display in the Command Window.

`mlint('filename','-eml')` `'-eml'` is not recommended. Use `'-codegen'` instead.

Examples

The following examples use `lengthofline.m`, which is a sample file with MATLAB code that can be improved. You can find it in *`matlabroot`*`/help/techdoc/matlab_env/examples`. If you want to run the examples, save a copy of `lengthofline.m` to a location on your MATLAB path.

Running mlint on a File with No Options

To run `mlint` on the example file, `lengthofline.m`, run

```
mlint('lengthofline')
```

MATLAB displays the M-Lint messages for `lengthofline.m` in the Command Window:

```
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop.Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses to group, if needed.
```

For details about these messages and how to improve the code, see “Changing Code Based on Code Analyzer Messages” in the MATLAB Desktop Tools and Development Environment documentation.

Running mlint with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=mlint('lengthofline', '-id')
```

MATLAB returns

```
inform =

19x1 struct array with fields:
    message
    line
    column
    id
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans =

    message: 'The value assigned here to variable 'nothandle' might never be used.'
       line: 22
     column: [1 9]
         id: 'NASGU'
```

Here, the message is for the value that appears on line 22 that extends from column 1–9 in the file.NASGU is the ID for the message `'The value assigned here to variable 'nothandle' might never be used.'`.

Displaying McCabe Complexity with mlint

To display the McCabe complexity of a MATLAB code file, run `mlint` with the `-cyc` option, as shown in the following example (assuming you have saved `lengthofline.m` to a local folder).

```
mlint lengthofline.m -cyc
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the M-Lint messages, as shown here:

```
L 1 (C 23-34): The McCabe complexity of 'lengthofline' is 12.
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for more information.
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary.  Use parentheses to group, if needed.
```

See Also

`mlintrpt`, `profile`

How To

- For information on the suppression directive, `%#ok`, and suppressing messages from within your program, see “Adjust Code Analyzer Message Indicators and Messages ”.

Introduced before R2006a

mlintrpt

Run `checkcode` for file or folder, reporting results in browser

Syntax

```
mlintrpt
mlintrpt('filename','file')
mlintrpt('dirname','dir')
mlintrpt('filename','file','settings.txt')
mlintrpt('dirname','dir','settings.txt')
```

Description

`mlintrpt` scans all files with an `.m` file extension in the current folder for Code Analyzer messages and reports the results in a MATLAB Web browser.

`mlintrpt('filename','file')` scans `filename` for Code Analyzer messages and reports results. You can omit `'file'` in this form of the syntax because it is the default.

`mlintrpt('dirname','dir')` scans the specified folder. Here, `dirname` can be in the current folder or can be a full path.

`mlintrpt('filename','file','settings.txt')` applies the Code Analyzer preference settings to enable or suppress messages as indicated in the specified `settings.txt` file.

`mlintrpt('dirname','dir','settings.txt')` applies the settings indicated in the specified `settings.txt` file.

Note: If you specify a `settings.txt` file, you must specify the full path to the file.

Examples

`lengthofline.m` is an example file with code that can be improved. It is found in *`matlabroot`*`/matlab/help/techdoc/matlab_env/examples`.

Run Report for All Files in a Folder

Run

```
mlintrpt(fullfile(matlabroot,'help','techdoc','matlab_env','examples'),'dir')
```

and MATLAB displays a report of potential problems and improvements for all files with an `.m` file extension in the `examples` folder.

For details about these messages and how to improve the code, see "Changing Code Based on Code Analyzer Messages".

Run Report Using Code Analyzer Preference Settings

You can save preference settings to a text file by clicking the **Preferences** button in the **Environment** section on the **Home** tab and selecting **Code Analyzer** in the left pane. To save a preferences file, select **Save as** under the  drop-down list. To apply those settings when you run `mlintrpt`, use the `file` option and supply the full path to the settings file name as shown in this example:

```
mlintrpt('lengthofline.m', 'file', ...
'C:\WINNT\Profiles\me\Application Data\MathWorks\MATLAB\R2012b\mymlint.txt')
```

Alternatively, use `fullfile` if the settings file is stored in the preferences folder:

```
 mlintrpt('lengthofline.m', 'file', fullfile(prefdir,'mymlint.txt'))
```

Assuming that in that example `mymlint.txt` file, the setting for `Terminate statement with semicolon to suppress output` has been disabled, the results of `mlintrpt` for `lengthofline` do not show that message for line 49.

When `mlintrpt` cannot locate the settings file, the first message in the report is

```
0: Unable to open or read the configuration file 'mymlint.txt'--using default settings.
```

More About

- "Check Code for Errors and Warnings"

See Also

`checkcode`

Introduced before R2006a

mlock

Prevent clearing function from memory

Syntax

```
mlock
```

Description

`mlock` locks the currently running function in memory so that subsequent `clear` functions do not remove it. Locking a function in memory also prevents any `persistent` variables defined in the file from getting reinitialized.

Use the `munlock` function to return the file to its normal, clearable state.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked('testfun')
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock('testfun')
```

```
mislocked('testfun')
ans =
     0
```

See Also

`mislocked` | `inmem` | `munlock` | `persistent`

Introduced before R2006a

mmfileinfo

Information about multimedia file

Syntax

```
info = mmfileinfo(filename)
```

Description

`info = mmfileinfo(filename)` returns a structure, `info`, with fields containing information about the contents of the multimedia file identified by `filename`. The `filename` input is a string enclosed in single quotation marks.

If `filename` is a URL, `mmfileinfo` might take a long time to return because it must first download the file. For large files, downloading can take several minutes. To avoid blocking the MATLAB command line while this processing takes place, download the file before calling `mmfileinfo`.

The `info` structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Filename`	String indicating the name of the file.
`Path`	String indicating the absolute path to the file.
`Duration`	Length of the file in seconds.
`Audio`	Structure containing information about the audio data in the file. See “Audio Data” on page 1-5206 for more information about this data structure.
`Video`	Structure containing information about the video data in the file. See “Video Data” on page 1-5206 for more information about this data structure.

Audio Data

The `Audio` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain audio data, the fields in the structure are empty.

Field	Description
`Format`	Text string, indicating the audio format.
`NumberOfChannels`	Number of audio channels.

Video Data

The `Video` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain video data, the fields in the structure are empty.

Field	Description
`Format`	Text string, indicating the video format.
`Height`	Height of the video frame.
`Width`	Width of the video frame.

Examples

Display information about the example file `xylophone.mpg`:

```
info = mmfileinfo('xylophone.mpg')
audio = info.Audio
video = info.Video
```

MATLAB returns:

```
info =
    Filename: 'xylophone.mpg'
        Path: 'matlabroot\toolbox\matlab\audiovideo'
    Duration: 4.7020
       Audio: [1x1 struct]
       Video: [1x1 struct]

audio =
              Format: 'MPEG'
```

```
    NumberOfChannels: 2

video =
    Format: 'MPEG1'
    Height: 240
     Width: 320
```

where `Path` is system-dependent.

See Also

`get` | `VideoReader`

Introduced before R2006a

mmreader class

Create object for reading video files

Note: `mmreader` has been removed. Use `VideoReader` instead.

Description

Use `mmreader` with the `read` method to read video data from a multimedia file into the MATLAB workspace.

The file formats that `mmreader` supports vary by platform, as follows (with no restrictions on file extensions):

All Platforms	Motion JPEG 2000 (`.mj2`)
Windows	AVI (`.avi`), MPEG-1 (`.mpg`), Windows Media Video (`.wmv`, `.asf`, `.asx`), and any format supported by Microsoft DirectShow.
Macintosh	AVI (`.avi`), MPEG-1 (`.mpg`), MPEG-4 (`.mp4`, `.m4v`), Apple QuickTime Movie (`.mov`), and any format supported by QuickTime as listed on http://support.apple.com/kb/HT3775.
Linux	Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on http://gstreamer.freedesktop.org/documentation/plugins.html, including AVI (`.avi`) and Ogg Theora (`.ogg`).

For more information, see “Supported Video File Formats” in the MATLAB Data Import and Export documentation.

Construction

obj = mmreader(*filename*) constructs *obj* to read video data from the file named *filename*. The mmreader constructor searches for the file on the MATLAB path. If it cannot construct the object for any reason, mmreader generates an error.

obj = mmreader(*filename*,'*PropertyName*',*PropertyValue*) constructs the object using options, specified as property name/value pairs. Property name/value pairs can be in any format that the set method supports: name/value string pairs, structures, or name/value cell array pairs.

Properties

BitsPerPixel

Bits per pixel of the video data. (Read-only)

Duration

Total length of the file in seconds. (Read-only)

FrameRate

Frame rate of the video in frames per second. (Read-only)

Height

Height of the video frame in pixels. (Read-only)

Name

Name of the file associated with the object. (Read-only)

NumberOfFrames

Total number of frames in the video stream. (Read-only)

Some files store video at a variable frame rate, including many Windows Media Video files. For these files, mmreader cannot determine the number of frames until you read the last frame. When you construct the object, mmreader returns a warning and does not set the NumberOfFrames property.

To count the number of frames in a variable frame rate file, use the `read` method to read the last frame of the file. For example:

```
vidObj = mmreader('varFrameRateFile.wmv');
lastFrame = read(vidObj, inf);
numFrames = vidObj.NumberOfFrames;
```

`Path`

String containing the full path to the file associated with the reader. (Read-only)

`Tag`

User-defined string to identify the object.

Default: `''`

`Type`

Class name of the object: `'mmreader'`. (Read-only)

`UserData`

Generic field for user-defined data.

Default: `[]`

`VideoFormat`

String indicating the MATLAB representation of the video format, such as `'RGB24'`. (Read-only)

`Width`

Width of the video frame in pixels. (Read-only)

Methods

For backward compatibility, `mmreader` supports the following `VideoReader` methods:

get	Query property values for video reader object

getFileFormats	File formats that `VideoReader` supports
hasFrame	Determine if frame available to read
read	Read video frame data from file
readFrame	Read video frame from video file
set	Set property values for video reader object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Construct an `mmreader` object for the example movie file `xylophone.mpg` and view its properties:

```
xyloObj = mmreader('xylophone.mpg', 'Tag', 'My reader object');
get(xyloObj)
```

Read and play back the movie file `xylophone.mpg`:

```
xyloObj = mmreader('xylophone.mpg');

nFrames = xyloObj.NumberOfFrames;
vidHeight = xyloObj.Height;
vidWidth = xyloObj.Width;

% Preallocate movie structure.
mov(1:nFrames) = ...
    struct('cdata', zeros(vidHeight, vidWidth, 3, 'uint8'),...
           'colormap', []);

% Read one frame at a time.
```

```
for k = 1 : nFrames
    mov(k).cdata = read(xyloObj, k);
end

% Size a figure based on the video's width and height.
hf = figure;
set(hf, 'position', [150 150 vidWidth vidHeight])

% Play back the movie once at the video's frame rate.
movie(hf, mov, 1, xyloObj.FrameRate);
```

See Also

`mmfileinfo` | `VideoReader`

How To

- "Read Video Files"

mod

Remainder after division (modulo operation)

Syntax

```
b = mod(a,m)
```

Description

`b = mod(a,m)` returns the remainder after division of `a` by `m`, where `a` is the dividend and `m` is the divisor. This function is often called the modulo operation and is computed using `b = a - m.*floor(a./m)`. The `mod` function follows the convention that `mod(a,0)` returns `a`.

Examples

Remainder After Division of Scalar

Compute 23 modulo 5.

```
b = mod(23,5)
```

```
b =

     3
```

Remainder After Division of Vector

Find the remainder after division for a vector of integers and the divisor 3.

```
a = 1:5;
m = 3;
b = mod(a,m)
```

```
b =
```

```
     1     2     0     1     2
```

Remainder After Division for Positive and Negative Values

Find the remainder after division for a set of integers including both positive and negative values. Note that nonzero results are always positive if the divisor is positive.

```
a = [-4 -1 7 9];
m = 3;
b = mod(a,m)

b =

     2     2     1     0
```

Remainder After Division for Negative Divisor

Find the remainder after division by a negative divisor for a set of integers including both positive and negative values. Note that nonzero results are always negative if the divisor is negative.

```
a = [-4 -1 7 9];
m = -3;
b = mod(a,m)

b =

    -1    -1    -2     0
```

Remainder After Division for Floating-Point Values

Find the remainder after division for several angles using a modulus of `2*pi`. Note that `mod` attempts to compensate for floating-point round-off effects to produce exact integer results when possible.

```
theta = [0.0 3.5 5.9 6.2 9.0 4*pi];
m = 2*pi;
b = mod(theta,m)

b =
```

```
     0    3.5000    5.9000    6.2000    2.7168         0
```

Input Arguments

a — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a scalar, vector, matrix, or multidimensional array. `a` must be a real-valued array of any numerical type. Inputs `a` and `m` must be the same size unless one is a scalar `double`. If one input has an integer data type, then the other input must be of the same integer data type or be a scalar `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

m — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a scalar, vector, matrix, or multidimensional array. `m` must be a real-valued array of any numerical type. Inputs `a` and `m` must be the same size unless one is a scalar `double`. If one input has an integer data type, then the other input must be of the same integer data type or be a scalar `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Differences Between mod and rem

The concept of remainder after division is not uniquely defined, and the two functions `mod` and `rem` each compute a different variation. The `mod` function produces a result that is either zero or has the same sign as the divisor. The `rem` function produces a result that is either zero or has the same sign as the dividend.

Another difference is the convention when the divisor is zero. The `mod` function follows the convention that `mod(a,0)` returns `a`, whereas the `rem` function follows the convention that `rem(a,0)` returns `NaN`.

Both variants have their uses. For example, in signal processing, the `mod` function is useful in the context of periodic signals because its output is periodic (with period equal to the divisor).

Congruence Relationships

The `mod` function is useful for congruence relationships: `a` and `b` are congruent (mod m) if and only if `mod(a,m) == mod(b,m)`. For example, 23 and 13 are congruent (mod 5).

References

[1] Knuth, Donald E. *The Art of Computer Programming*. Vol. 1. Addison Wesley, 1997 pp.39–40.

See Also

`rem`

Introduced before R2006a

mode

Most frequent values in array

Syntax

```
M = mode(A)
M = mode(A,dim)

[M,F] = mode( ___ )
[M,F,C] = mode( ___ )
```

Description

`M = mode(A)` returns the sample mode of `A`, which is the most frequently occurring value in `A`. When there are multiple values occurring equally frequently, `mode` returns the smallest of those values. For complex inputs, the smallest value is the first value in a sorted list.

- If `A` is a vector, then `mode(A)` returns the most frequent value of `A`.
- If `A` is a nonempty matrix, then `mode(A)` returns a row vector containing the mode of each column of `A`.
- If `A` is an empty 0-by-0 matrix, `mode(A)` returns `NaN`.
- If `A` is a multidimensional array, then `mode(A)` treats the values along the first array dimension whose size does not equal `1` as vectors and returns an array of most frequent values. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.

`M = mode(A,dim)` returns the mode of elements along dimension `dim`. For example, if `A` is a matrix, then `mode(A,2)` is a column vector containing the most frequent value of each row

`[M,F] = mode( ___ )` also returns a frequency array `F`, using any of the input arguments in the previous syntaxes. `F` is the same size as `M`, and each element of `F` represents the number of occurrences of the corresponding element of `M`.

`[M,F,C] = mode(___)` also returns a cell array `C` of the same size as `M` and `F`. Each element of `C` is a sorted vector of all values that have the same frequency as the corresponding element of `M`.

Examples

Mode of Matrix Columns

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
A =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each column.

```
M = mode(A)
```

```
M =

     0     0     1     4
```

Mode of Matrix Rows

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
A =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each row.

```
M = mode(A,2)
```

```
M =
```

```
     3
     0
     0
```

Mode of 3-D Array

Create a 1-by-3-by-4 array of integers between 1 and 10.

```
A = gallery('integerdata',10,[1,3,4],1)
```

```
A(:,:,1) =

    10     8    10


A(:,:,2) =

     6     9     5


A(:,:,3) =

     9     6     1


A(:,:,4) =

     4     9     5
```

Find the most frequent values of this 3-D array along the second dimension.

```
M = mode(A)
```

```
M(:,:,1) =

    10


M(:,:,2) =

     5


M(:,:,3) =
```

```
     1


M(:,:,4) =

     4
```

This operation produces a 1-by-1-by-4 array by finding the most frequent value along the second dimension. The size of the second dimension reduces to 1.

Compute the mode along the first dimension of A.

```
M = mode(A,1);
isequal(A,M)

ans =

     1
```

This returns the same array as A because the size of the first dimension is 1.

Mode of Matrix Columns with Frequency Information

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]

A =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each column, as well as how often it occurs.

```
[M,F] = mode(A)

M =

     0     0     1     4


F =
```

```
     2     1     2     2
```

`F(1)` is `2` since `M(1)` occurs twice in the first column.

Mode of Matrix Rows with Frequency and Multiplicity Information

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
A =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each row, how often it occurs, and which values in that row occur with the same frequency.

```
[M,F,C] = mode(A,2)
```

```
M =

     3
     0
     0


F =

     2
     2
     1


C =

    [         3]
    [2x1 double]
    [4x1 double]
```

`C{2}` is the 2-by-1 vector `[0;1]` since values `0` and `1` in the second row occur with frequency `F(2)`.

`C{3}` is the 4-by-1 vector `[0;1;2;4]` since all values in the third row occur with frequency `F(3)`.

Mode of 16-bit Unsigned Integer Array

Define a 1-by-4 vector of 16-bit unsigned integers.

```
A = gallery('integerdata',10,[1,4],3,'uint16')
```

```
A =

      6      3      2      3
```

Find the most frequent value, as well as the number of times it occurs.

```
[M,F] = mode(A),
class(M)
```

```
M =

      3


F =

     2


ans =

uint16
```

`M` is the same class as the input, `A`.

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. `A` can be a numeric array, categorical array, datetime array, or duration array.

`NaN` or `NaT` (Not a Time) values in the input array, `A`, are ignored. Undefined values in categorical arrays are similar to `NaN`s in numeric arrays.

`dim` — Dimension to operate along

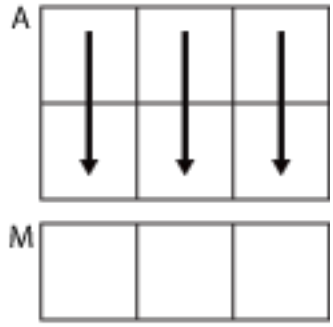
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

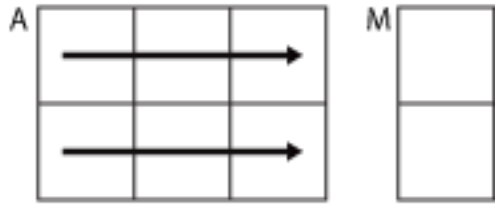
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `mode(A,1)` returns a row vector containing the most frequent value in each column.



mode(A,1)

- If `dim = 2`, then `mode(A,2)` returns a column vector containing the most frequent value in each row.



mode(A,2)

`mode` returns `A` if `dim` is greater than `ndims(A)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`M` — Most frequent values

scalar | vector | matrix | multidimensional array

Most frequent values returned as a scalar, vector, matrix, or multidimensional array. When there are multiple values occurring equally frequently, `mode` returns the smallest of those values. For complex inputs, this is taken to be the first value in a sorted list of values.

The class of `M` is the same as the class of the input array, `A`.

`F` — Frequency array

scalar | vector | matrix | multidimensional array

Frequency array returned as a scalar, vector, matrix, or multidimensional array. The size of `F` is the same as the size of `M`, and each element of `F` represents the number of occurrences of the corresponding element of `M`.

The class of `F` is always `double`.

`C` — Most frequent values with multiplicity

cell array

Most frequent values with multiplicity returned as a cell array. The size of `C` is the same as the size of `M` and `F`, and each element of `C` is a sorted column vector of all values that have the same frequency as the corresponding element of `M`.

More About

Tips

- The `mode` function is most useful with discrete or coarsely rounded data. The mode for a continuous probability distribution is defined as the peak of its density function. Applying the `mode` function to a sample from that distribution is unlikely to provide a good estimate of the peak; it would be better to compute a histogram or density

estimate and calculate the peak of that estimate. Also, the `mode` function is not suitable for finding peaks in distributions having multiple modes.

See Also

`histcounts` | `histogram` | `mean` | `median` | `sort`

Introduced before R2006a

month

Month number and name

Syntax

```
m = month(t)
m = month(t,monthType)
```

Description

`m = month(t)` returns the month numbers of the datetime values in `t`. The `m` output contains integer values from 1 to 12.

`m = month(t,monthType)` returns the type of month number or name specified by `monthType`.

The `month` function returns the month numbers or names of datetime values. To assign month numbers to datetime array `t`, use `t.Month` and modify the `Month` property.

Examples

Extract Month Number from Dates

Extract the month numbers from an array of dates.

```
t = datetime(2014,05,31):caldays(35):datetime(2014,10,15)
```

```
t =

   31-May-2014   05-Jul-2014   09-Aug-2014   13-Sep-2014
```

```
m = month(t)
```

```
m =

     5     7     8     9
```

Find Month Names of Dates

Get the month names from an array of dates.

```
t = datetime(2013,01,01):calweeks(12):datetime(2013,12,31)
```

```
t =

   01-Jan-2013   26-Mar-2013   18-Jun-2013   10-Sep-2013   03-Dec-2013
```

```
m = month(t,'name')
```

```
m =

    'January'    'March'    'June'    'September'    'December'
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`monthType` — Type of month values

`'monthofyear'` (default) | `'name'` | `'shortname'`

Type of month values, specified as one of the following strings.

Value of `monthType`	Description
`'monthofyear'`	Month-of-year number
`'name'`	Full month names, for example, `August` or `September`.

Value of `monthType`	Description
`'shortname'`	Abbreviated month names, for example, `Aug` or `Sep`.

Output Arguments

`m` — Month number or name

`double` array | cell array of strings

Month number or name, returned as a numeric array of type `double`, or a cell array of strings. `m` is the same size as `t`.

See Also

datetime Properties | `day` | `quarter` | `week` | `year` | `ymd`

Introduced in R2014b

more

Control paged output for Command Window

Syntax

```
more on
more off
more(n)
A = more(state)
```

Description

`more on` enables paging of the output in the MATLAB Command Window. MATLAB displays output one page at a time. Use the keys defined in the table below to control paging.

`more off` disables paging of the output in the MATLAB Command Window.

`more(n)` defines the length of a page to be `n` lines.

`A = more(state)` returns in `A` the number of lines that are currently defined to be a page. The `state` input can be one of the quoted strings `'on'` or `'off'`, or the number of lines to set as the new page length.

By default, the length of a page is equal to the number of lines available for display in the MATLAB Command Window. Manually changing the size of the command window adjusts the page length accordingly.

If you set the page length to a specific value, MATLAB uses that value for the page size, regardless of the size of the command window. To have MATLAB return to matching page size to window size, type `more off` followed by `more on`.

To see the status of `more`, type `get(0,'More')`. MATLAB returns either `on` or `off`, indicating the `more` status.

When you have enabled `more` and are examining output, you can do the following.

Press the...	To...
Return key	Advance to the next line of output.
Space bar	Advance to the next page of output.
Q (for quit) key	Terminate display of the text. Do not use **Ctrl+C** to terminate `more` or you might generate error messages in the Command Window.

`more` is in the **`off`** state, by default.

See Also

`diary`

Introduced before R2006a

morebins

Increase number of histogram bins

Syntax

```
N = morebins(h)
```

Description

`N = morebins(h)` increases the number of bins in histogram `h` by 10% (rounded up to the nearest integer) and returns the new number of bins.

Examples

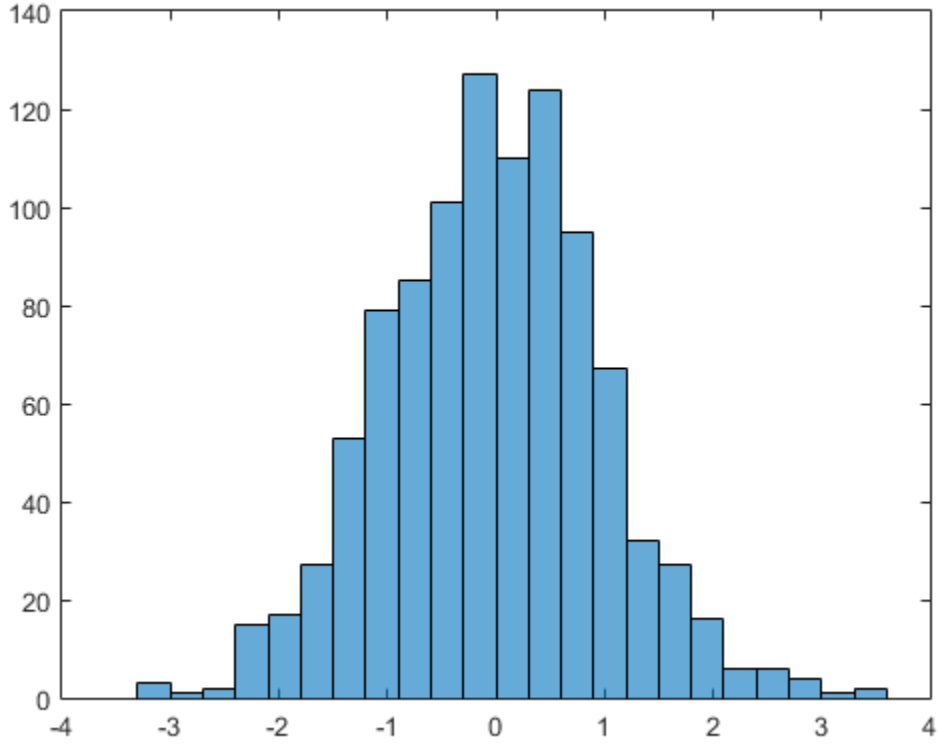
Increase Number of Histogram Bins

Plot a histogram of 1,000 random numbers and return a handle to the histogram object.

```
x = randn(1000,1);
h = histogram(x)
```

```
h =

  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]
```

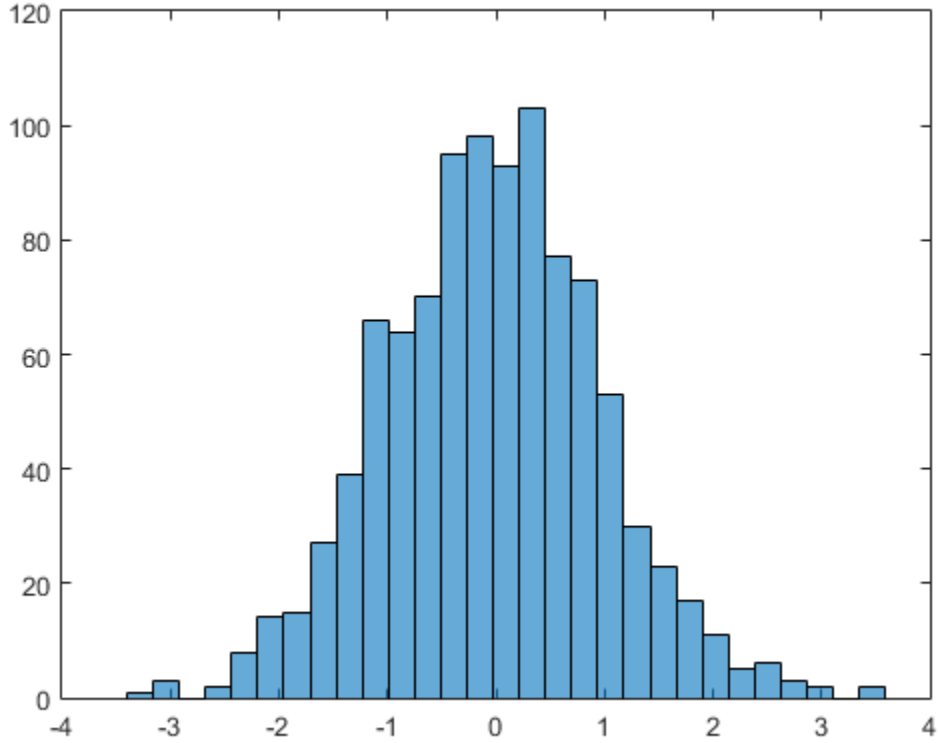
```
Use GET to show all properties
```



Use `morebins` to increase the number of bins in the histogram.

```
morebins(h);
morebins(h)
```

```
ans =

    29
```



Input Arguments

h — Input histogram

histogram object

Input histogram, specified as a histogram object. For more information, see Using histogram Objects.

Output Arguments

`N` — Number of bins

scalar

Number of bins, returned as a scalar. `N` is the new number of bins for the histogram after increase.

More About

- Using histogram Objects
- Histogram Properties

See Also

`fewerbins` | `histcounts` | `histogram`

Introduced in R2014b

move

Move or resize control in parent window

Syntax

```
V = move(h,position)
```

Description

`V = move(h,position)` moves the control to the position specified by the `position` argument. When you use `move` with only the handle argument, `h`, it returns a four-element vector indicating the current position of the control.

The position argument is a four-element vector specifying the position and size of the control in the parent figure window. The elements of the vector are:

```
[x, y, width, height]
```

where `x` and `y` are offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control, and `width` and `height` are the size of the control itself.

Examples

This example moves the control.

```
f = figure('Position',[100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1',[0 0 200 200],f);
pos = move(h,[50 50 200 200])
```

```
pos =
    50    50   200   200
```

The next example resizes the control to always be centered in the figure as you resize the figure window. Start by creating the script `resizectrl.m` that contains:

```
% Get the new position and size of the figure window
  fpos = get(gcbo,'position');
```

```
% Resize the control accordingly
  move(h,[0 0 fpos(3) fpos(4)]);
```

Now execute the following:

```
f = figure('Position',[100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1',[0 0 200 200]);
set(f,'ResizeFcn','resizectrl');
```

As you resize the figure window, notice that the circle moves so that it is always positioned in the center of the window.

See Also

`set (COM)` | `get (COM)`

Introduced before R2006a

movefile

Move file or folder

Syntax

```
movefile('source')
movefile('source','destination')
movefile('source','destination','f')
[status,message,messageid] = movefile( ___ )
```

Description

`movefile('source')` moves the file or folder named `source` to the current folder, where `source` is the absolute or relative path name for the folder or file. To move multiple files or folders, use one or more wildcard characters (`*`) after the last file separator in `source`. The `source` argument permits a wildcard character in a path string. `movefile` does not preserve the archive attribute of `source`.

`movefile('source','destination')` moves the file or folder named `source` to the location `destination`, where `source` and `destination` are the absolute or relative paths for the folder or file. To move multiple files or folders, you can use one or more wildcard characters (`*`) after the last file separator in `source`. You cannot use a wildcard character in `destination`. To rename a file or folder when moving it, make `destination` a different name than `source`, and specify only one file for `source`. When `source` and `destination` have the same location, `movefile` renames `source` to `destination`.

`movefile('source','destination','f')` moves the file or folder named `source` to the location `destination`, regardless of the read-only attribute of `destination`.

`[status,message,messageid] = movefile( ___ )` moves the file or folder named `source` to the location `destination`, returning the status, a message, and the MATLAB message ID. Here, `status` is logical `1` for success or logical `0` for error. `movefile` requires only one output argument.

Examples

Moving a File to the Current Folder

Assuming `myfiles` is a subfolder within the current folder, move the file `myfunction.m` to the current folder:

```
movefile('myfiles/myfunction.m')
```

Assuming `projects/testcases` is the current folder, move `projects/myfiles` and its contents to the current folder:

```
movefile('../myfiles')
```

Renaming a File in the Current Folder

In the current folder, rename `oldname.m` to `newname.m`:

```
movefile('oldname.m','newname.m')
```

Using a Wildcard to Move All Matching Files

Assuming `myfiles` is a subfolder of the current folder, move all files whose names that begin with `my` from the `myfiles` folder, to the current folder:

```
movefile('myfiles/my*')
```

Moving a File to a Different Folder

Assuming `projects` and the current folder are at the same level, move the file `myfunction.m` from the current folder to the folder `projects`:

```
movefile('myfunction.m','../projects')
```

Moving a Folder Down One Level

Assuming `projects` is a subfolder of the current folder, move the folder `projects/testcases` and all its contents down a level in `projects` into `projects/myfiles`:

```
movefile('projects/testcases','projects/myfiles/')
```

Moving a File to a Read-Only Folder and Renaming the File

Move the file `myfile.m` from the current folder to `d:/work/restricted`, assigning it the name `test1.m`, where `restricted` is a read-only folder:

```
movefile('myfile.m','d:/work/restricted/test1.m','f')
```

The read-only file `myfile.m` is no longer in the current folder. The file `test1.m` is in `d:/work/restricted` and is read only.

Returning Status When Moving Files

Move all files in the folder `myfiles` whose names start with `new` to the current folder, when there is an error. You mistype `new*` as `nex*` and no items in the current folder start with `nex*`:

```
[s,mess,messid] = movefile('myfiles/nex*')

s =
     0

mess =

No matching files were found.

messid =

MATLAB:MOVEFILE:FileDoesNotExist
```

More About

- "Manage Files and Folders"

See Also

`cd` | `copyfile` | `delete` | `dir` | `fileattrib` | `ls` | `mkdir` | `rmdir`

Introduced before R2006a

movegui

Move UI figure to specified location on screen

Syntax

```
movegui(h,'position')
movegui(position)
movegui(h)
movegui
```

Description

`movegui(h,'position')` moves the figure identified by handle `h` to the specified screen location, preserving the figure's size. The *`position`* argument is either a string or a two-element vector, as defined in the tables that follow.

`movegui(position)` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the specified position.

`movegui(h)` moves the figure identified by the handle `h` to the `onscreen` position.

`movegui` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the `onscreen` position. You can specify `'movegui'` as a CreateFcn callback for a figure. Doing so ensures after you save a figure, that figure appears on screen when you reload it, regardless of its saved position. See the following example.

When it is a string, *`position`* is one of the following descriptors.

Position String	Description
`north`	Top center edge of screen
`south`	Bottom center edge of screen
`east`	Right center edge of screen
`west`	Left center edge of screen
`northeast`	Top right corner of screen

Position String	Description
`northwest`	Top left corner of screen
`southeast`	Bottom right corner of screen
`southwest`	Bottom left corner
`center`	Centered on screen
`onscreen`	Nearest location to current location that is entirely on screen

You can also specify the *`position`* argument as a two-element vector, `[h,v]`. Depending on sign, `h` specifies the figure's offset from the left or right edge of the screen, and `v` specifies the figure's offset from the top or bottom of the screen, in pixels. The following table summarizes the possible values.

`h` (for `h >= 0`)	Offset of left side from left edge of screen
`h` (for `h < 0`)	Offset of right side from right edge of screen
`v` (for `v >= 0`)	Offset of bottom edge from bottom of screen
`v` (for `v < 0`)	Offset of top edge from top of screen

Applying `movegui` to a maximized figure window moves the window towards the task bar and creates a gap on the opposite side of the screen about as wide as the task bar. The window might shrink in size by a few pixels. If you use the `onscreen` option with a maximized figure window, then `movegui` creates a gap on both the left and upper sides of the screen so that the top-left corner of the figure is visible.

`GUIDE` and `openfig` call `movegui` when loading figures to ensure they are visible.

Examples

Ensure that a saved UI window appears on screen when you reload it, regardless of the target computer screen size and resolution. Create a figure that is off the screen, assign `movegui` as its CreateFcn callback, save the figure, and then reload it.

```
f = figure('Position',[10000,10000,400,300]);
% The figure does not display because
% it is created offscreen.
```

```
f.CreateFcn = @movegui;
hgsave(f,'onscreenfig');
close(f);
f2 = hgload('onscreenfig');
% The reloaded figure is now visible
```

Move a figure to the bottom left corner of the screen.

```
f = figure;
movegui(f,'southwest');
```

Move a figure so that it is offset 100 pixels from the bottom and left side of the screen.

```
f = figure;
movegui(f,[100,100]);
```

See Also

`guide` | `openfig`

Introduced before R2006a

movie

Play recorded movie frames

Syntax

```
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

Description

The `movie` function plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once, using the current axes as the default target. If you want to play the movie in the figure instead of the axes, specify the figure handle (or `gcf`) as the first argument: `movie(figure_handle,...)`. `M` must be an array of movie frames (usually from `getframe`).

`movie(M,n)` plays the movie `n` times. If `n` is negative, each cycle is shown forward then backward. If `n` is a vector, the first element is the number of times to play the movie, and the remaining elements make up a list of frames to play in the movie.

For example, if `M` has four frames then `n = [10 4 4 2 1]` plays the movie ten times, and the movie consists of frame `4` followed by frame `4` again, followed by frame `2` and finally frame `1`.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie centered in the figure or axes identified by the handle `h`. Specifying the figure or axes enables MATLAB to fit the movie to the available size.

`movie(h,M,n,fps,loc)` specifies `loc`, a four-element location vector, `[x y 0 0]`, where the lower left corner of the movie frame is anchored (only the first two elements in

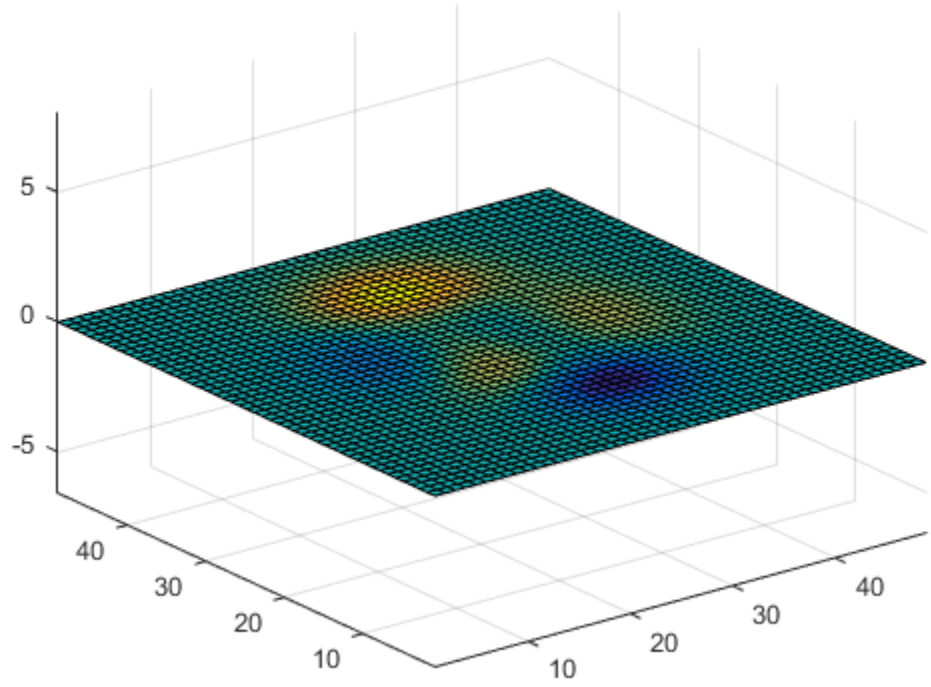
the vector are used). The location is relative to the lower left corner of the figure or axes specified by `handle h` and in units of pixels, regardless of the object's `Units` property.

Examples

Record Frames and Play Movie

Use the `getframe` function in a loop to record frames of the `peaks` function vibrating. Preallocate an array to store the movie frames.

```
figure
Z = peaks;
surf(Z)
axis tight manual
ax = gca;
ax.NextPlot = 'replaceChildren';


loops = 40;
F(loops) = struct('cdata',[],'colormap',[]);
for j = 1:loops
    X = sin(j*pi/10)*Z;
    surf(X,Z)
    drawnow
    F(j) = getframe;
end
```



To play the movie two times, use `movie(F,2)`.

More About

Tips

The `movie` function uses a default figure size of 560-by-420 and does not resize figures to fit movies with larger or smaller frames. To accommodate other frame sizes, you can resize the figure to fit the movie, as shown in the second example below.

`movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale or 24–bit truecolor image frames.

Buffering the movie places all frames in memory. As a result, on Microsoft Windows and perhaps other platforms, a long movie (on the order of several hundred frames) can exhaust memory, depending on system resources. In such cases an error message is issued:

```
??? Error using ==> movie
Could not create movie frame
```

You can abort a movie by typing **Ctrl-C**.

`movie` is not a built-in function. Therefore, you cannot call `movie` using the `builtin` function.

Limitations with Renderer on Windows Systems

Setting the figure `Renderer` property to `painters` works around limitations of using `getframe` with the `OpenGL` renderer on some Windows systems.

- "Record Animation for Playback"

See Also

`getframe` | `frame2im` | `im2frame` | `VideoReader` | `VideoWriter`

Introduced before R2006a

movie2avi

Create Audio/Video Interleaved (AVI) file from MATLAB movie

Note: `movie2avi` will be removed in a future release. Use `VideoWriter` instead.

Syntax

```
movie2avi(mov, filename)
movie2avi(mov, filename, ParameterName, ParameterValue)
```

Description

`movie2avi(mov, filename)` creates the AVI file *`filename`* from the MATLAB movie *`mov`*. The *`filename`* input is a string. The *`mov`* input is a 1-by-*`n`* structure array, where *`n`* is the number of frames. Each frame is a structure with two fields: `cdata` and `colormap`. For more information, see `getframe`.

`movie2avi(mov, filename, ParameterName, ParameterValue)` accepts one or more comma-separated parameter name/value pairs. The following table lists the available parameters and values.

Parameter Name	Value	Default
`'colormap'`	An *`m`*-by-3 matrix defining the colormap for indexed AVI movies, where *`m`* is no more than 256 (236 for `Indeo` compression). Valid only when the `'compression'` is `'MSVC'`, `'RLE'`, or `'None'`.	No default
`'compression'`	A text string specifying the compression codec to use. To create an uncompressed file, specify a value of `'None'`. On UNIX operating systems, the only valid value is `'None'`.	`'Indeo5'` on Windows systems. `'None'` on UNIX systems.

Parameter Name	Value	Default
	On Windows systems, valid values include: • `'MSVC'` • `'RLE'` • `'Cinepak'` on 32-bit systems. • `'Indeo3'` or `'Indeo5'` on 32-bit Windows XP systems. Alternatively, specify a custom compression codec on Windows systems using the four-character code that identifies the codec (typically included in the codec documentation). If MATLAB cannot find the specified codec, it returns an error.	
`'fps'`	A scalar value specifying the speed of the AVI movie in frames per second (fps).	15 fps
`'keyframe'`	For compressors that support temporal compression, the number of key frames per second.	2.1429 key frames per second
`'quality'`	A number from 0 through 100. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes. Valid only for compressed movies.	75
`'videoname'`	A descriptive name for the video stream, no more than 64 characters.	*`filename`*

Examples

Create a movie and write to an uncompressed AVI file, `myPeaks.avi`:

```
nFrames = 20;

% Preallocate movie structure.
mov(1:nFrames) = struct('cdata', [],...
                        'colormap', []);
```

```
% Create movie.
Z = peaks; surf(Z);
axis tight manual
set(gca,'nextplot','replacechildren');
for k = 1:nFrames
   surf(sin(2*pi*k/20)*Z,Z)
   mov(k) = getframe(gcf);
end

% Create AVI file.
movie2avi(mov, 'myPeaks.avi', 'compression', 'None');
```

More About

Tips

- On some Windows systems, including all 64-bit systems, the default Indeo 5 codec is not available. MATLAB issues a warning, and creates an uncompressed file.
- On 32-bit Windows XP systems, MATLAB can create AVI files compressed with Indeo 3 and Indeo 5 codecs. However, Microsoft Windows XP Service Pack 3 (SP3) with Security Update 954157 disables playback of Indeo 3 and Indeo 5 codecs in Windows Media Player and Internet Explorer. Consider specifying a `compression` value of `'None'`.

See Also

`VideoWriter` | `VideoReader` | `mmfileinfo` | `movie`

Introduced before R2006a

mpower, ^

Matrix power

Syntax

```
C = A^B
C = mpower(A,B)
```

Description

`C = A^B` computes `A` to the `B` power and returns the result in `C`.

`C = mpower(A,B)` is an alternate way to execute `A^B`, but is rarely used. It enables operator overloading for classes.

Examples

Square a Matrix

Create a 2-by-2 matrix and square it.

```
A = [1 2; 3 4];
C = A^2
```

```
C =

     7    10
    15    22
```

The syntax `A^2` is equivalent to `A*A`.

Matrix Exponents

Create a 2-by-2 matrix and use it as the exponent for a scalar.

```
B = [0 1; 1 0];
```

```
C = 2^B

C =

    1.2500    0.7500
    0.7500    1.2500
```

Compute `C` by first finding the eigenvalues `D` and eigenvectors `V` of the matrix `B`.

```
[V,D] = eig(B)

V =

   -0.7071    0.7071
    0.7071    0.7071


D =

    -1     0
     0     1
```

Next, use the formula `2^B = V*2^D/V` to compute the power.

```
C = V*2^D/V

C =

    1.2500    0.7500
    0.7500    1.2500
```

Input Arguments

A — Base

scalar | matrix

Base, specified as a scalar or matrix. Inputs `A` and `B` must be one of the following:

- Base `A` is a square matrix and exponent `B` is a scalar. If `B` is a positive integer, the power is computed by repeated squaring. For other values of `B` the calculation involves eigenvalues and eigenvectors.
- Base `A` is a scalar and exponent `B` is a square matrix. The calculation uses eigenvalues and eigenvectors.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

B — Exponent

scalar | matrix

Exponent, specified as a scalar or matrix. Inputs A and B must be one of the following:

- Base A is a square matrix and exponent B is a scalar. If B is a positive integer, the power is computed by repeated squaring. For other values of B the calculation involves eigenvalues and eigenvectors.
- Base A is a scalar and exponent B is a square matrix. The calculation uses eigenvalues and eigenvectors.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

More About

- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`mtimes` | `power` | `times`

mput

Class: FTP

Upload file or folder to FTP server

Syntax

```
mput(ftpobj,contents)
paths = mput(ftpobj,contents)
```

Description

`mput(ftpobj,contents)` uploads the file or folder specified by contents to the current folder on an FTP server.

`paths = mput(ftpobj,contents)` returns a cell array that lists the paths to the uploaded files on the server.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`contents`

String enclosed in single quotation marks that specifies either a file name or a folder name. Can include a wildcard character (`*`).

Output Arguments

`paths`

Cell array that includes the paths to the uploaded files on the server.

Examples

Suppose that your current MATLAB folder contains files `myfile1.m` through `myfile10.m`, and that you want to upload to a hypothetical FTP server, `ftp.testsite.com`. Connect to the server and upload the files:

```
test = ftp('ftp.testsite.com');
mput(test, 'myfile*.m');
close(test);
```

See Also

`mkdir` | `mget` | `ftp` | `rename`

Introduced before R2006a

msgbox

Create message dialog box

Syntax

```
h = msgbox(Message)
h = msgbox(Message,Title)
h = msgbox(Message,Title,Icon)
h = msgbox(Message,Title,'custom',IconData,IconCMap)
h = msgbox(___,CreateMode)
```

Description

`h = msgbox(Message)` creates a message dialog box that automatically wraps Message to fit an appropriately sized figure.

`h = msgbox(Message,Title)` specifies the title of the message box.

`h = msgbox(Message,Title,Icon)` specifies which built-in icon to display in the message dialog box.

`h = msgbox(Message,Title,'custom',IconData,IconCMap)` specifies a custom icon to include in the message dialog box. IconData is the image data that defines the icon. IconCMap is the colormap used for the image. If IconData is a true-color image, you do not need to specify an IconCMap.

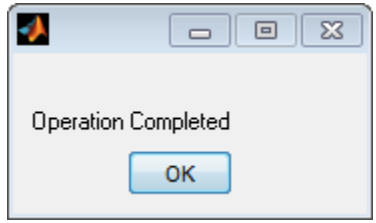
`h = msgbox(___,CreateMode)` specifies whether the message box is modal. Additionally, you can specify a TeX interpreter for Message and Title.

Examples

Simple Message Dialog Box

Specify the text you want displayed in the message dialog box.

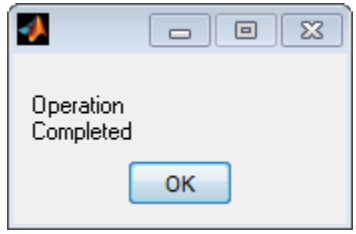
```
h = msgbox('Operation Completed');
```



Message Dialog Box Text with Line Breaks

Specify the message dialog box text using a cell array of strings to insert line breaks between the display of each string in the cell array.

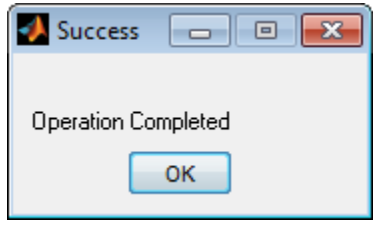
```
h = msgbox({'Operation' 'Completed'});
```



Message Dialog Box with a Title

Specify the message dialog box text and give the dialog box a title, `Success`.

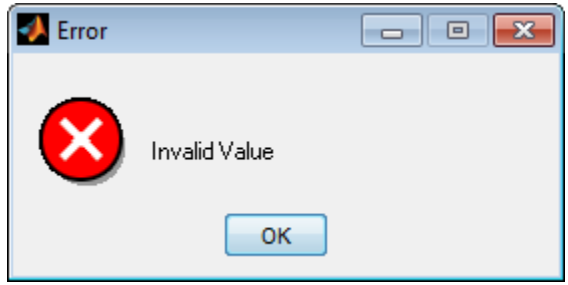
```
h = msgbox('Operation Completed','Success');
```



Message Dialog Box That Uses a Built-in Icon

Include a built-in error icon with an error message in a message dialog box entitled Error.

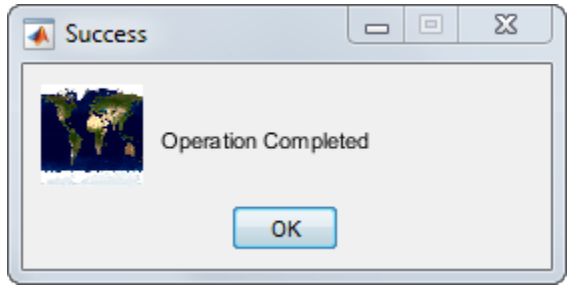
```
h = msgbox('Invalid Value', 'Error','error');
```



Message Dialog Box That Uses a True-Color Custom Icon

Read an RGB image into the workspace. Then, specify it as a custom icon in the dialog box.

```
myicon = imread('landOcean.jpg');
h=msgbox('Operation Completed','Success','custom',myicon);
```



Message Dialog Box That Uses an Indexed Color Icon

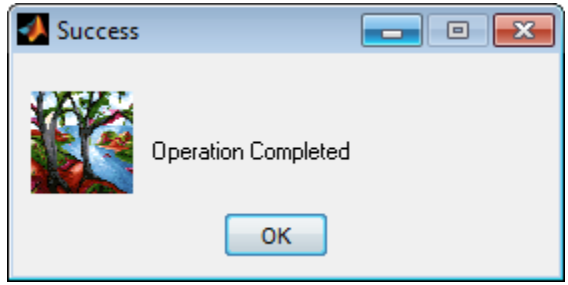
Use `trees.tiff` (which is on the MATLAB path) as an icon in your message dialog box. Because `.tiff` images use a colormap to define the colors, you must specify a colormap. Change the colormap to change the image colors.

Determine the value to specify for IconData by passing the image file to `imread`.

```
[cdata,map] = imread('trees.tif');
```

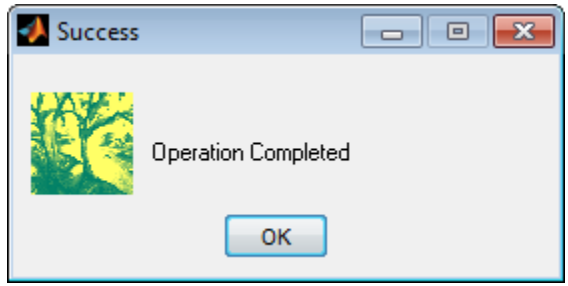
Create the message dialog box, including the custom icon.

```
h=msgbox('Operation Completed',...
         'Success','custom',cdata,map);
```



Adjust the image colors by specifying a different colormap. For instance, specify the MATLAB built-in colormap, `summer`

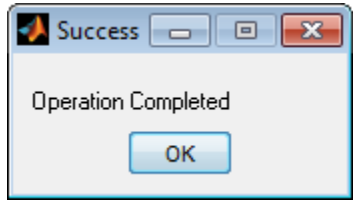
```
h=msgbox('Operation Completed','Success','custom',...
         cdata,summer);
```



Modal Message Dialog Box

Create a modal message dialog box, wrapping the call to `msgbox` with `uiwait` to make the message dialog box block MATLAB execution until the user responds to the message dialog box.

```
uiwait(msgbox('Operation Completed','Success','modal'));
```



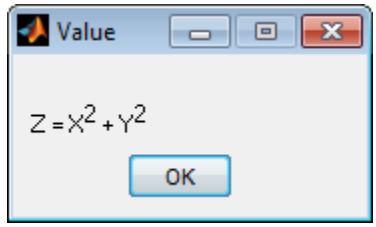
Modal Message Dialog Box That Uses a TeX Formatted Message

Create a structure to specify that the user must click OK before interacting with another window and that MATLAB interpret the message text as TeX format.

```
CreateStruct.Interpreter = 'tex';
CreateStruct.WindowStyle = 'modal';
```

Create the message dialog box.

```
h=msgbox('Z = X^2 + Y^2','Value',CreateStruct);
```



Input Arguments

`Message` — Message dialog box text

string vector | string matrix | cell array

Dialog box text specified as a string vector, string matrix, or cell array.

Example: `'Operation Completed'`

Example: `['Operation ', 'Completed']`

Example: `{'Operation', 'Completed'}`

`Title` — Message dialog box title bar text

string vector

Dialog box title bar text specified as a string vector.

Example: `'Success'`

`Icon` — Icon to include in message dialog box

`'none'` (default) | `'error'` | `'help'` | `'warn'` | `'custom'`

Icon to include in message dialog box specified as a string.

Built-in icons appear as follows:

-  Error
-  Help
-  Warn

`IconData` — Image data defining a custom icon

matrix

Image data defining a custom icon specified as a matrix. Each element of the matrix specifies the color of a rectangular segment in the image. Use `imread` to get the IconData value for an image that you want to use as a message dialog box icon.

Example: [1:64]'*[1:64])/64

`IconCMap` — Colormap for a custom icon that is not true-color

m–by–3 matrix | built-in colormap

Colormap for a custom icon that is not true-color, specified as an m-by-3 matrix of real numbers between `0.0` and `1.0`, or as a MATLAB built-in colormap. Use `imread` to get the IconCMap value for an image that you want to use as a message dialog box icon.

Example: [0.5 0.5 0.5]

Example: hot(64)

`CreateMode` — Message dialog box mode

`'nonmodal'` (default) | structure | `'modal'` | `'replace'`

Mode in which message dialog box is created, specified as a string or a structure.

- If CreateMode is a structure, it can have the fields `WindowStyle` and `Interpreter`. The `WindowStyle` field must be one of the strings listed in the list items that

follow this one. The `Interpreter` field must be the string `'tex'` or `'none'`. If the `Interpreter` value is `'tex'`, MATLAB interprets the Message and Title values as TeX. The default value for `Interpreter` is `'none'`.

- If CreateMode is `'nonmodal'`, MATLAB creates a new nonmodal message box with the specified parameters. Existing message boxes with the same Title remain.
- If CreateMode is `'modal'`, MATLAB replaces the existing message box with the specified Title that was last created or clicked on with the specified modal dialog box. MATLAB deletes all other message boxes with the same title. The replaced message box can be either modal or nonmodal.
- If CreateMode is `'replace'`, MATLAB replaces the message box having the specified Title that was last created or clicked on with a nonmodal message box as specified. MATLAB deletes all other message boxes with the same title. The replaced message box can be either modal or nonmodal.

Example: CreateStruct.Interpreter='tex';

Output Arguments

h — Message dialog box handle

scalar

Message dialog box handle returned as a scalar. This is a unique identifier, which you can use to query and modify the properties of a specific message dialog box.

More About

modal dialog box

A modal dialog box prevents a user from interacting with other windows before responding to the modal dialog box.

For more information about modal dialog boxes, see WindowStyle in the Figure Properties topic.

Tips

- Program execution continues even when a modal dialog box is active. To block MATLAB program execution until the user responds to the modal dialog box, use the `uiwait` function.

- Modal dialogs (created using `errordlg`, `msgbox`, or `warndlg`) replace any existing dialogs created with these functions that also have the same name.
- "Reading Image Data"

See Also

`errordlg` | `helpdlg` | `imread` | `warndlg`

Introduced before R2006a

mtimes, *

Matrix Multiplication

Syntax

```
C = A*B
C = mtimes(A,B)
```

Description

`C = A*B` is the matrix product of `A` and `B`. If `A` is an m-by-p and `B` is a p-by-n matrix, then `C` is an m-by-n matrix defined by

$$C(i,j) = \sum_{k=1}^{p} A(i,k)B(k,j).$$

This definition says that `C(i,j)` is the inner product of the `i`th row of `A` with the `j`th column of `B`. You can write this definition using the MATLAB colon operator as

```
C(i,j) = A(i,:)*B(:,j)
```

For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. Matrix multiplication is *not* universally commutative for nonscalar inputs. That is, `A*B` is typically not equal to `B*A`. If at least one input is scalar, then `A*B` is equivalent to `A.*B` and is commutative.

`C = mtimes(A,B)` is an alternative way to execute `A*B`, but is rarely used. It enables operator overloading for classes.

Examples

Multiply Two Vectors

Create a 1-by-4 row vector, `A`, and a 4-by-1 column vector, `B`.

```
A = [1 1 0 0];
B = [1; 2; 3; 4];
```

Multiply A times B.

```
C = A*B
```

```
C =

     3
```

The result is a 1-by-1 scalar, also called the *dot product* or *inner product* of the vectors A and B. Alternatively, you can calculate the dot product A · B with the syntax `dot(A,B)`.

Multiply B times A.

```
C = B*A
```

```
C =

     1     1     0     0
     2     2     0     0
     3     3     0     0
     4     4     0     0
```

The result is a 4-by-4 matrix, also called the *outer product* of the vectors A and B. The outer product of two vectors, A ⊗ B, returns a matrix.

Multiply Two Arrays

Create two arrays, A and B.

```
A = [1 3 5; 2 4 7];
B = [-5 8 11; 3 9 21; 4 0 8];
```

Calculate the product of A and B.

```
C = A*B
```

```
C =

    24    35   114
    30    52   162
```

Calculate the inner product of the second row of A and the third column of B.

```
A(2,:)*B(:,3)
```

```
ans =
```

```
162
```

This answer is the same as `C(2,3)`.

Input Arguments

`A` — Left Array

scalar | vector | matrix

Left Array, specified as a scalar, vector, or matrix. For nonscalar inputs, the number of columns in `A` must be equal to the number of rows in `B`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `duration` | `calendarDuration`
Complex Number Support: Yes

`B` — Right Array

scalar | vector | matrix

Right Array, specified as a scalar, vector, or matrix. For nonscalar inputs, the number of columns in `A` must be equal to the number of rows in `B`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `duration` | `calendarDuration`
Complex Number Support: Yes

Output Arguments

`C` — Product Array

scalar | vector | matrix

Product Array, returned as a scalar, vector, or matrix. Array `C` has the same number of rows as input `A` and the same number of columns as input `B`. For example, if `A` is an m-by-0 empty matrix and `B` is a 0-by-n empty matrix, then `A*B` is an m-by-n matrix of zeros.

More About

- “Array vs. Matrix Operations”

- “Operator Precedence”

See Also

`colon` | `cross` | `dot` | `times`

Introduced before R2006a

mu2lin

Convert mu-law audio signal to linear

Syntax

```
y = mu2lin(mu)
```

Description

`y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as “flints” in the range $0 \le$ `mu` ≤ 255, to linear signal amplitude in the range `-s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. "Flints" are MATLAB integers — floating-point numbers whose values are integers.

See Also

`auread` | `lin2mu`

Introduced before R2006a

multibandread

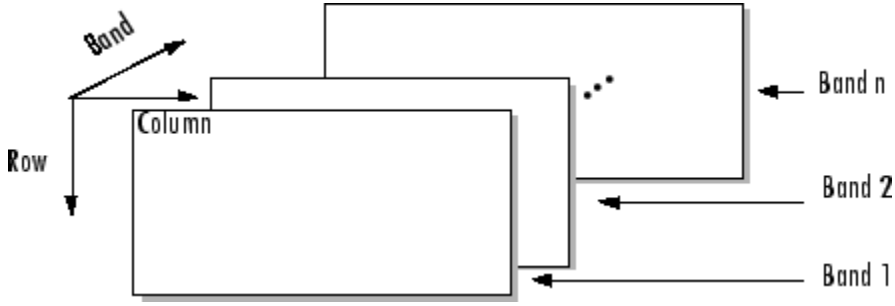
Read band-interleaved data from binary file

Syntax

```
X = multibandread(filename, size, precision, offset, interleave,
byteorder)
X = multibandread(...,subset1,subset2,subset3)
```

Description

`X = multibandread(filename, size, precision, offset, interleave, byteorder)` reads band-sequential (BSQ), band-interleaved-by-line (BIL), or band-interleaved-by-pixel (BIP) data from the binary file `filename`. The `filename` input is a string enclosed in single quotes. This function defines *band* as the third dimension in a 3-D array, as shown in this figure.



You can use the parameters to `multibandread` to specify many aspects of the read operation, such as which bands to read. See “Parameters” on page 1-5269 for more information.

`X` is a 2-D array if only one band is read; otherwise it is 3-D. `X` is returned as an array of data type `double` by default. Use the `precision` parameter to map the data to a different data type.

`X = multibandread(...,subset1,subset2,subset3)` reads a subset of the data in the file. You can use up to three subsetting parameters to specify the data subset along row, column, and band dimensions. See "Subsetting Parameters" on page 1-5270 for more information.

Note: In addition to BSQ, BIL, and BIP files, multiband imagery may be stored using the TIFF file format. In that case, use the `imread` function to import the data.

Parameters

This table describes the arguments accepted by `multibandread`.

Argument	Description
`filename`	String containing the name of the file to be read.
`size`	Three-element vector of integers consisting of `[height, width, N]`, where • `height` is the total number of rows • `width` is the total number of elements in each row • `N` is the total number of bands. This will be the dimensions of the data if it is read in its entirety.
`precision`	String specifying the format of the data to be read, such as `'uint8'`, `'double'`, `'integer*4'`, or any of the other precisions supported by the `fread` function. Note: You can also use the `precision` parameter to specify the format of the output data. For example, to read `uint8` data and output a `uint8` array, specify a precision of `'uint8=>uint8'` (or `'*uint8'`). To read `uint8` data and output it in the MATLAB software in single precision, specify `'uint8=>single'`. See `fread` for more information.
`offset`	Scalar specifying the zero-based location of the first data element in the file. This value represents the number of bytes from the beginning of the file to where the data begins.
`interleave`	String specifying the format in which the data is stored

Argument	Description
	• `'bsq'` — Band-Sequential • `'bil'` — Band-Interleaved-by-Line • `'bip'` — Band-Interleaved-by-Pixel For more information about these interleave methods, see the `multibandwrite` reference page.
`byteorder`	String specifying the byte ordering (machine format) in which the data is stored, such as • `'ieee-le'` — Little-endian • `'ieee-be'` — Big-endian See `fopen` for a complete list of supported formats.

Subsetting Parameters

You can specify up to three subsetting parameters. Each subsetting parameter is a three-element cell array, {*dim*,*method*,index}, where

Parameter	Description
dim	Text string specifying the dimension to subset along. It can have any of these values: • `'Column'` • `'Row'` • `'Band'`
method	Text string specifying the subsetting method. It can have either of these values: • `'Direct'` • `'Range'` If you leave out this element of the subset cell array, `multibandread` uses `'Direct'` as the default.

Parameter	Description
`index`	If `method` is `'Direct'`, `index` is a vector specifying the indices to read along the Band dimension. If `method` is `'Range'`, `index` is a three-element vector of `[start, increment, stop]` specifying the range and step size to read along the dimension specified in `dim`. If `index` is a two-element vector, `multibandread` assumes that the value of `increment` is `1`.

Examples

Example 1

Setup initial parameters for a data set.

```
rows=3; cols=3; bands=5;
filename = tempname;
```

Define the data set.

```
fid = fopen(filename, 'w', 'ieee-le');
fwrite(fid, 1:rows*cols*bands, 'double');
fclose(fid);
```

Read every other band of the data using the Band-Sequential format.

```
im1 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bsq', 'ieee-le', ...
                    {'Band', 'Range', [1 2 bands]} )
```

Read the first two rows and columns of data using Band-Interleaved-by-Pixel format.

```
im2 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bip', 'ieee-le', ...
                    {'Row', 'Range', [1 2]}, ...
                    {'Column', 'Range', [1 2]} )
```

Read the data using Band-Interleaved-by-Line format.

```
im3 = multibandread(filename, [rows cols bands], ...
```

```
                    'double', 0, 'bil', 'ieee-le')
```

Delete the file created in this example.

```
delete(filename);
```

Example 2

Read `int16` BIL data from the FITS file `tst0012.fits`, starting at byte 74880.

```
im4 = multibandread('tst0012.fits', [31 73 5], ...
                    'int16', 74880, 'bil', 'ieee-be', ...
                    {'Band', 'Range', [1 3]} );
im5 = double(im4)/max(max(max(im4)));
imagesc(im5);
```

See Also

`fread` | `fwrite` | `imread` | `memmapfile` | `multibandwrite`

Introduced before R2006a

multibandwrite

Write band-interleaved data to file

Syntax

```
multibandwrite(data,filename,interleave)
multibandwrite(data,filename,interleave,start,totalsize)
multibandwrite(...,param,value...)
```

Description

`multibandwrite(data,filename,interleave)` writes `data`, a two- or three-dimensional numeric or logical array, to the binary file specified by `filename`. The `filename` input is a string enclosed in single quotes. The length of the third dimension of `data` determines the number of bands written to the file. The bands are written to the file in the form specified by `interleave`. See "Interleave Methods" on page 1-5274 for more information about this argument.

If filename already exists, `multibandwrite` overwrites it unless you specify the optional `offset` parameter. See the last alternate syntax for `multibandwrite` for information about other optional parameters.

`multibandwrite(data,filename,interleave,start,totalsize)` writes `data` to the binary file `filename` in chunks. In this syntax, `data` is a subset of the complete data set.

`start` is a 1-by-3 array `[firstrow firstcolumn firstband]` that specifies the location to start writing data. `firstrow` and `firstcolumn` specify the location of the upper left image pixel. `firstband` gives the index of the first band to write. For example, `data(I,J,K)` contains the data for the pixel at `[firstrow+I-1, firstcolumn+J-1]` in the `(firstband+K-1)`-th band.

`totalsize` is a 1-by-3 array, `[totalrows,totalcolumns,totalbands]`, which specifies the full, three-dimensional size of the data to be written to the file.

Note In this syntax, you must call `multibandwrite` multiple times to write all the data to the file. The first time it is called, `multibandwrite` writes the complete file,

using the fill value for all values outside the data subset. In each subsequent call, `multibandwrite` overwrites these fill values with the data subset in `data`. The parameters `filename`, `interleave`, `offset`, and `totalsize` must remain constant throughout the writing of the file.

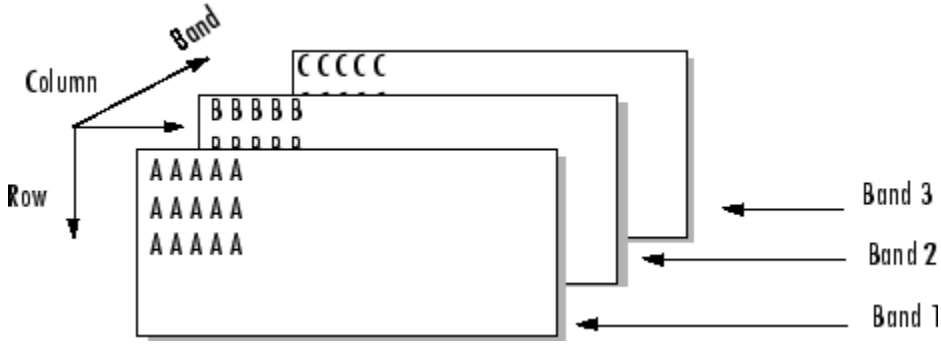
`multibandwrite(...,param,value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

Parameter	Description
`'precision'`	String specifying the form and size of each element written to the file. See the help for `fwrite` for a list of valid values. The default precision is the class of the data.
`'offset'`	The number of bytes to skip before the first data element. If the file does not already exist, `multibandwrite` writes ASCII null values to fill the space. To specify a different fill value, use the parameter `'fillvalue'`. This option is useful when you are writing a header to the file before or after writing the data. When writing the header to the file after the data is written, open the file with `fopen` using `'r+'` permission.
`'machfmt'`	String to control the format in which the data is written to the file. Typical values are `'ieee-le'` for little endian and `'ieee-be'` for big endian. See the help for `fopen` for a complete list of available formats. The default machine format is the local machine format.
`'fillvalue'`	A number specifying the value to use in place of missing data. `'fillvalue'` can be a single number, specifying the fill value for all missing data, or a 1-by-Number-of-bands vector of numbers specifying the fill value for each band. This value is used to fill space when data is written in chunks.

Interleave Methods

`interleave` is a string that specifies how `multibandwrite` interleaves the bands as it writes data to the file. If `data` is two-dimensional, `multibandwrite` ignores the

`interleave` argument. The following table lists the supported methods and uses this example multiband file to illustrate each method.



Supported methods of interleaving bands include those listed below.

Method	String	Description	Example
Band-Interleaved-by-Line	`'bil'`	Write an entire row from each band	AAAAABBBBBCCCCC AAAAABBBBBCCCCC AAAAABBBBBCCCCC
Band-Interleaved-by-Pixel	`'bip'`	Write a pixel from each band	ABCABCABCABCABC...
Band-Sequential	`'bsq'`	Write each band in its entirety	AAAAA AAAAA AAAAA BBBBB BBBBB BBBBB CCCCC CCCCC CCCCC

Examples

Note To run these examples successfully, you must be in a writable directory.

Example 1

Write all data (interleaved by line) to the file in one call.

```
data = reshape(uint16(1:600), [10 20 3]);
multibandwrite(data,'data.bil','bil');
```

Example 2

Write a single-band tiled image with one call for each tile. This is only useful if a subset of each band is available at each call to `multibandwrite`.

```
numBands = 1;
dataDims = [1024 1024 numBands];
data = reshape(uint32(1:(1024 * 1024 * numBands)), dataDims);

for band = 1:numBands
   for row = 1:2
      for col = 1:2

         subsetRows = ((row - 1) * 512 + 1):(row * 512);
         subsetCols = ((col - 1) * 512 + 1):(col * 512);

         upperLeft = [subsetRows(1), subsetCols(1), band];
         multibandwrite(data(subsetRows, subsetCols, band), ...
                        'banddata.bsq', 'bsq', upperLeft, dataDims);

      end
   end
end
```

See Also

`multibandread` | `fwrite` | `fread`

Introduced before R2006a

munlock

Allow clearing functions from memory

Syntax

```
munlock
munlock fun
munlock('fun')
```

Description

`munlock` unlocks the currently running `.m` or `.mex` function in memory so that subsequent `clear` functions can remove it.

`munlock fun` unlocks the `.m` or `.mex` file named `fun` from memory. By default, these files are unlocked so that changes to the file are picked up. Calls to munlock are needed only to unlock `.m` or `.mex` functions that have been locked with `mlock`.

`munlock('fun')` is the function form of `munlock`.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked testfun
ans =
```

```
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock testfun

mislocked testfun
ans =
     0
```

See Also

`mlock` | `mislocked` | `inmem` | `persistent`

Introduced before R2006a

namelengthmax

Maximum identifier length

Syntax

```
len = namelengthmax
```

Description

`len = namelengthmax` returns the maximum length allowed for MATLAB identifiers, which include:

- Variable names
- Structure field names
- Script, function, and class names
- Model names

Rather than hard-coding a specific maximum name length into your programs, use the `namelengthmax` function. This saves you the trouble of having to update these limits should the identifier length change in some future MATLAB release.

Examples

Call `namelengthmax` to get the maximum identifier length:

```
maxid = namelengthmax
maxid =
    63
```

See Also

`isvarname` | `matlab.lang.makeValidName` | `matlab.lang.makeUniqueStrings`

Introduced before R2006a

NaN

Not-a-Number

Syntax

```
NaN
N = NaN(n)
N = NaN(sz1,...,szN)
N = NaN(sz)

N = NaN(classname)
N = NaN(n,classname)
N = NaN(sz1,...szN,classname)
N = NaN(sz,classname)

N = NaN('like',p)
N = NaN(n,'like',p)
N = NaN(sz1,...szN,'like',p)
N = NaN(sz,'like',p)
```

Description

`NaN` returns the IEEE arithmetic representation for Not-a-Number (`NaN`). These values result from operations which have undefined numerical results.

`N = NaN(n)` is an `n`-by-`n` matrix of `NaN` values.

`N = NaN(sz1,...,szN)` is a `sz1`-by-...-by-`szN` array of `NaN` values where `sz1,...,szN` indicates the size of each dimension. For example, `NaN(3,4)` returns a 3-by-4 array of `NaN` values.

`N = NaN(sz)` is an array of `NaN` values where the size vector, `sz`, defines `size(N)`. For example, `NaN([3,4])` returns a 3-by-4 array of `NaN` values.

Note The size inputs `sz1,...,szN`, as well as the elements of the size vector `sz`, should be nonnegative integers. Negative integers are treated as 0.

`N = NaN(classname)` returns a `NaN` value where the string, `classname`, specifies the data type. `classname` can be either `'single'` or `'double'`.

`N = NaN(n,classname)` returns an `n`-by-`n` array of `NaN` values of data type `classname`.

`N = NaN(sz1,...szN,classname)` returns a `sz1`-by-...-by-`szN` array of `NaN` values of data type `classname`.

`N = NaN(sz,classname)` returns an array of `NaN` values where the size vector, `sz`, defines `size(N)` and `classname` defines `class(N)`.

`N = NaN('like',p)` returns a `NaN` value of the same data type, sparsity, and complexity (real or complex) as the numeric variable, `p`.

`N = NaN(n,'like',p)` returns an `n`-by-`n` array of `NaN` values like `p`.

`N = NaN(sz1,...szN,'like',p)` returns a `sz1`-by-...-by-`szN` array of `NaN` values like `p`.

`N = NaN(sz,'like',p)` returns an array of `NaN` values like `p` where the size vector, `sz`, defines `size(N)`.

Examples

These operations produce `NaN`:

- Any arithmetic operation on a `NaN`, such as `sqrt(NaN)`
- Addition or subtraction, such as magnitude subtraction of infinities as `(+Inf)+(-Inf)`
- Multiplication, such as `0*Inf`
- Division, such as `0/0` and `Inf/Inf`
- Remainder, such as `rem(x,y)` where `y` is zero or `x` is infinity

More About

Tips

Because two `NaN`s are not equal to each other, logical operations involving `NaN`s always return false, except ~= (not equal). Consequently,

```
NaN ~= NaN
ans =
     1
NaN == NaN
ans =
     0
```

and the NaNs in a vector are treated as different unique elements.

```
unique([1 1 NaN NaN])
ans =
     1 NaN NaN
```

Use the `isnan` function to detect NaNs in an array.

```
isnan([1 1 NaN NaN])
ans =
     0     0     1     1
```

- "Class Support for Array-Creation Functions"

See Also

`inf` | `isfinite` | `isfloat` | `isnan`

Introduced before R2006a

nargchk

Validate number of input arguments

Note: `nargchk` will be removed in a future version. Use `narginchk` instead.

Syntax

```
msgstring = nargchk(minargs, maxargs, numargs)
msgstring = nargchk(minargs, maxargs, numargs, 'string')
msgstruct = nargchk(minargs, maxargs, numargs, 'struct')
```

Description

Use `nargchk` inside a function to check that the desired number of input arguments is specified in the call to that function.

`msgstring = nargchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of inputs specified in the call `numargs` is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty matrix.

It is common to use the `nargin` function to determine the number of input arguments specified in the call.

`msgstring = nargchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargchk` returns a string by default.

`msgstruct = nargchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty structure.

When too few inputs are supplied, the message string and identifier are

```
message: 'Not enough input arguments.'
```

```
    identifier: 'MATLAB:nargchk:notEnoughInputs'
```

When too many inputs are supplied, the message string and identifier are

```
       message: 'Too many input arguments.'
    identifier: 'MATLAB:nargchk:tooManyInputs'
```

Examples

Given the function `CheckInputs`,

```
function CheckInputs(x, y, z)
error(nargchk(2, 3, nargin))
```

Then typing `CheckInputs(1)` produces

```
Not enough input arguments.
```

More About

Tips

`nargchk` is often used together with the `error` function. The `error` function accepts either type of return value from `nargchk`: a message string or message structure. For example, this command provides the `error` function with a message string and identifier regarding which error was caught:

```
error(nargchk(2, 4, nargin, 'struct'))
```

If `nargchk` detects no error, it returns an empty string or structure. When `nargchk` is used with the `error` function, as shown here, this empty string or structure is passed as an input to `error`. When `error` receives an empty string or structure, it simply returns and no error is generated.

See Also

`narginchk` | `nargin` | `nargoutchk` | `nargout` | `varargin` | `varargout` | `error`

Introduced before R2006a

nargin

Number of function input arguments

Syntax

```
nargin
nargin(fx)
```

Description

`nargin` returns the number of input arguments passed in the call to the currently executing function. Use this `nargin` syntax only in the body of a function.

`nargin(fx)` returns the number of input arguments that appear in the definition statement for function `fx`. If the function includes `varargin` in its definition, then `nargin` returns the negative of the number of inputs. For example, if function `foo` declares inputs `a`, `b`, and `varargin`, then `nargin('foo')` returns `-3`.

Input Arguments

`fx`

Either a function handle or a string in single quotes that specifies the name of a function.

Examples

Inputs to Current Function

Create a function in a file named `addme.m` that accepts up to two inputs, and identify the number of inputs with `nargin`.

```
function c = addme(a,b)

switch nargin
    case 2
```

```
        c = a + b;
    case 1
        c = a + a;
    otherwise
        c = 0;
end
```

Inputs Defined for a Function

Determine how many inputs a function can accept.

The function `addme` created in the previous example has two inputs in its declaration statement (`a` and `b`).

```
fx = 'addme';
nargin(fx)
```

```
ans =
     2
```

Function with varargin Input

Determine how many inputs a function that uses `varargin` can accept.

Define a function in a file named `mynewplot.m` that accepts numeric inputs `x` and `y` and any number of additional plot inputs using `varargin`.

```
function mynewplot(x,y,varargin)
   figure
   plot(x,y,varargin{:})
   title('My New Plot')
```

At the command line, query how many inputs `newplot` can accept.

```
fx = 'mynewplot';
nargin(fx)
```

```
ans =
    -3
```

The minus sign indicates that the third input is `varargin`. The `mynewplot` function can accept an indeterminate number of additional input arguments.

See Also

`nargout` | `narginchk` | `nargoutchk` | `varargin` | `varargout` | `inputname`

Introduced before R2006a

narginchk

Validate number of input arguments

Syntax

```
narginchk(minargs, maxargs)
```

Description

`narginchk(minargs, maxargs)` throws an error if the number of inputs specified in the call to the currently executing function is less than `minargs` or greater than `maxargs`. If the number of inputs is between `minargs` and `maxargs` (inclusive), `narginchk` does nothing.

When too few inputs are supplied, the message identifier and message are:

```
    identifier: 'MATLAB:narginchk:notEnoughInputs'
       message: 'Not enough input arguments.'
```

When too many inputs are supplied, the message identifier and message are:

```
    identifier: 'MATLAB:narginchk:tooManyInputs'
       message: 'Too many input arguments.'
```

Examples

This function uses `narginchk` to verify that a minimum of 2 and maximum of 5 input arguments are received from the calling function:

```
function check_inputs(A, B, varargin)
minargs=2;  maxargs=5;

% Number of inputs must be >=minargs and <=maxargs.
narginchk(minargs, maxargs)

fprintf('Received 2 required, %d optional inputs.\n\n', ...
    size(varargin, 2))
```

Call the example function, passing 1 input argument:

```
check_inputs(23)
Error using check_inputs
Not enough input arguments.
```

Call the function, passing 5 arguments:

```
check_inputs(23, 9, 15, 34, 62)
Received 2 required, 3 optional inputs.
```

Call the function, passing 6 arguments:

```
check_inputs(23, 9, 15, 34, 62, 6)
Error using check_inputs
Too many input arguments.
```

More About

Tips

- To verify that there are a minimum of `N` arguments, specify `inf` as `maxargs`. For example: `narginchk(5,inf)` throws an error when there are not at least five inputs.

See Also

`nargin` | `nargoutchk` | `nargout` | `varargin` | `varargout`

nargout

Number of function output arguments

Syntax

```
nargout
nargout(fx)
```

Description

`nargout` returns the number of output arguments specified in the call to the currently executing function. Use this `nargout` syntax only in the body of a function.

`nargout(fx)` returns the number of outputs that appear in the definition statement of function `fx`. If the function includes `varargout` in its definition, then `nargout` returns the negative of the number of outputs. For example, if function `foo` declares outputs `a`, `b`, and `varargout`, then `nargout('foo')` returns `-3`.

Input Arguments

`fx`

Either a function handle or a string in single quotes that specifies the name of a function.

Examples

Outputs for Current Function

Create a function in a file named `subtract.m` that calculates a second return value only when requested.

```
function [dif,absdif] = subtract(y,x)
```

```
dif = y - x;
if nargout > 1
   disp('Calculating absolute value')
   absdif = abs(dif);
end
```

Outputs Defined for a Function

Determine how many outputs a function can return.

The function named `subtract` created in the previous example has two outputs in its declaration statement (`dif` and `absdif`).

```
fx = 'subtract';
nargout(fx)

ans =
     2
```

Function with varargout Output

Determine how many outputs a function that uses `varargout` can return.

Define a function in a file named `mysize.m` that returns a vector of dimensions from the `size` function and the individual dimensions using `varargout`.

```
function [sizeVector,varargout] = mysize(x)
   sizeVector = size(x);
   varargout = cell(1,nargout-1);
   for k = 1:length(varargout)
      varargout{k} = sizeVector(k);
   end
```

At the command line, query how many outputs `mysize` can return.

```
fx = 'mysize';
nargout(fx)

ans =
    -2
```

The minus sign indicates that the second output is `varargout`. The `mysize` function can return an indeterminate number of additional outputs.

More About

Tips

- When you use a function as part of an expression, MATLAB calls the function with one output argument, so `nargout` within the function returns `1`. For example, given the following `if` statement and the `subtract` function defined in the Examples section, the value of `nargout` within the `subtract` function is `1`.

```
a = 1;
b = 2;
if subtract(a,b) < 0
   disp('Result is negative')
end
```

See Also

`nargin` | `nargoutchk` | `narginchk` | `varargout` | `varargin` | `inputname`

Introduced before R2006a

nargoutchk

Validate number of output arguments

Syntax

```
nargoutchk(minargs, maxargs)
```

Description

`nargoutchk(minargs, maxargs)` throws an error if the number of outputs specified in the call is less than `minargs` or greater than `maxargs`. If the number of outputs is between `minargs` and `maxargs` (inclusive), `nargoutchk` does nothing.

When too few outputs are supplied, the identifier and message are:

```
identifier: 'MATLAB:nargoutchk:notEnoughOutputs'
       message: 'Not enough output arguments.'
```

When too many outputs are supplied, the identifier and message are:

```
identifier: 'MATLAB:nargoutchk:tooManyOutputs'
       message: 'Too many output arguments.'
```

Examples

This function uses `nargoutchk` to verify that a minimum of 2 and maximum of 5 input arguments are passed back to the calling function:

```
function [varargout] = check_outputs(array_in)
minargs=2;  maxargs=5;

% Number of outputs must be >=minargs and <=maxargs.
nargoutchk(minargs, maxargs)

for k=1:nargout
    varargout{k} = array_in(k)*3;
end
```

Initialize input array X to a vector of 6 elements:

```
X = 5:7:40
X =
     5    12    19    26    33    40
```

Call the example function with 1 output argument. This is less than the minimum (2) that was specified by `nargoutchk` and results in an error:

```
A = check_outputs(X);

Error using check_outputs
Not enough output arguments.
```

Call the function with 4 output arguments. This is within the allowable bounds (2 to 5) specified by `nargoutchk`:

```
[A, B, C, D] = check_outputs(X);

[A, B, C, D]
ans =
    15    36    57    78
```

Call the function with 6 output arguments. This exceeds the maximum (5) that was specified by `nargoutchk` and results in an error:

```
[A, B, C, D, E, F] = check_outputs(X);

Error using check_outputs
Too many output arguments.
```

More About

Tips

- To verify that there are a minimum of `N` arguments, specify `inf` as `maxargs`. For example: `nargoutchk(5,inf)` throws an error when there are not at least five outputs.

See Also

`narginchk` | `nargin` | `nargout` | `varargout` | `varargin`

Introduced before R2006a

native2unicode

Convert numeric bytes to Unicode character representation

Syntax

```
unicodestr = native2unicode(bytes)
unicodestr = native2unicode(bytes, encoding)
```

Description

`unicodestr = native2unicode(bytes)` converts a numeric vector, `bytes`, from the user default encoding to Unicode character representation. `bytes` is treated as a stream of 8-bit bytes, and each value must be in the range [0,255]. Return value `unicodestr` is a `char` vector having the same general array shape as `bytes`.

`unicodestr = native2unicode(bytes, encoding)` converts `bytes` to Unicode representation assuming that the byte stream is in the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. If `encoding` is unspecified or is the empty string (`''`), the default encoding scheme is used.

Note If `bytes` is a `char` vector, it is returned unchanged.

Examples

This example begins with a vector of bytes in an unknown character encoding scheme. The user-written function `detect_encoding` determines the encoding scheme. If successful, it returns the encoding scheme name or alias as a string. If unsuccessful, it throws an error represented by an MException object, `ME`. The example calls `native2unicode` to convert the bytes to Unicode representation:

```
try
   enc = detect_encoding(bytes);
```

```
    str = native2unicode(bytes, enc);
    disp(str);
catch ME
    rethrow(ME);
end
```

Note that the computer must be configured to display text in a language represented by the detected encoding scheme for the output of `disp(str)` to be correct.

See Also

`unicode2native`

Introduced before R2006a

nchoosek

Binomial coefficient or all combinations

Syntax

```
b = nchoosek(n,k)
C = nchoosek(v,k)
```

Description

`b = nchoosek(n,k)` returns the binomial coefficient, defined as $n!/((n–k)!\,k!)$. This is the number of combinations of `n` items taken `k` at a time.

`C = nchoosek(v,k)` returns a matrix containing all possible combinations of the elements of vector `v` taken `k` at a time. Matrix `C` has `k` columns and $n!/((n–k)!\,k!)$ rows, where n is `length(v)`.

Examples

Binomial Coefficient, "5 Choose 4"

```
b = nchoosek(5,4)
```

```
b =

     5
```

All Combinations of Five Numbers Taken Four at a Time

```
v = 2:2:10;
C = nchoosek(v,4)
```

```
C =

     2     4     6     8
     2     4     6    10
     2     4     8    10
```

```
     2     6     8    10
     4     6     8    10
```

All Combinations of Three Unsigned Integers Taken Two at a Time

```
v = uint16([10 20 30]);
C = nchoosek(v,uint16(2))
```

```
C =

     10     20
     10     30
     20     30
```

Input Arguments

n — Number of possible choices

scalar, real, nonnegative value

Number of possible choices, specified as a scalar value of any numeric type that is real and nonnegative.

Example: `10`

Example: `int16(10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

k — Number of selected choices

scalar, real, nonnegative value

Number of selected choices, specified as a scalar value that is real and nonnegative. `k` can be any numeric type. However, `nchoosek(n,k)` requires that `n` and `k` be the same type or that at least one of them be of type `double`.

There are no restrictions on combining inputs of different types for `nchoosek(v,k)`.

Example: `3`

Example: `int16(3)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

v — Set of all choices

vector of numeric, `logical`, or `char` values

Set of all choices, specified as a vector of numeric, `logical`, or `char` values.

Example: `[1 2 3 4 5]`

Example: `[1+1i 2+1i 3+1i 4+1i]`

Example: `int16([1 2 3 4 5])`

Example: `[true false true false]`

Example: `['abcd']`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Output Arguments

b — Binomial coefficient

nonnegative scalar value

Binomial coefficient, returned as a nonnegative scalar value. `b` is the same type as `n` and `k`. If `n` and `k` are of different types, then `b` is returned as the nondouble type.

C — All combinations of v

matrix

All combinations of `v`, returned as a matrix of the same type as `v`. Matrix `C` has `k` columns and $n!/((n–k)!\ k!)$ rows, where n is `length(v)`.

Each row of `C` contains a combination of `k` items chosen from `v`. The elements in each row of `C` are listed in the same order as they appear in `v`.

Limitations

- When `b = nchoosek(n,k)` is sufficiently large, `nchoosek` displays a warning that the result might not be exact. In this case, the result is only accurate to 15 digits for double-precision inputs, or 8 digits for single-precision inputs.

- `C = nchoosek(v,k)` is only practical for situations where `length(v)` is less than about `15`.

See Also

`perms`

Introduced before R2006a

ndgrid

Rectangular grid in N-D space

Syntax

```
[X1,X2,X3,...,Xn] = ndgrid(x1gv,x2gv,x3gv,...,xngv)
[X1,X2,...,Xn] = ndgrid(xgv)
```

Description

`[X1,X2,X3,...,Xn] = ndgrid(x1gv,x2gv,x3gv,...,xngv)` replicates the grid vectors `x1gv`,`x2gv`,`x3gv`,`...`,`xngv` to produce a full grid. This grid is represented by the output coordinate arrays `X1`,`X2`,`X3`,`...`,`Xn`. The ith dimension of any output array `Xi` contains copies of the grid vector `xigv`.

`[X1,X2,...,Xn] = ndgrid(xgv)` is the same as `[X1,X2,...,Xn] = ndgrid(xgv,xgv,...,xgv)`. In other words, you can reuse the same grid vector in each respective dimension. The dimensionality of the output arrays is determined by the number of output arguments.

The coordinate arrays `[X1,X2,X3,...,Xn]` are typically used to evaluate functions of several variables and to create surface and volumetric plots.

Input Arguments

`xigv`

Grid vector specifying a series of grid point coordinates in the ith dimension.

`xgv`

Generic grid vector specifying a series of point coordinates.

Output Arguments

`Xi`

The `i`th dimension of the output array `Xi` are copies of elements of the grid vector `xigv`. The output arrays specify the full grid.

Examples

Evaluate Function Over Gridded Domain

Evaluate the function $x_1 e^{-x_1^2 - x_2^2}$ over the gridded domain $-2 < x_1 < 2$ and $-2 < x_2 < 2$.

Create a grid of values for the domain.

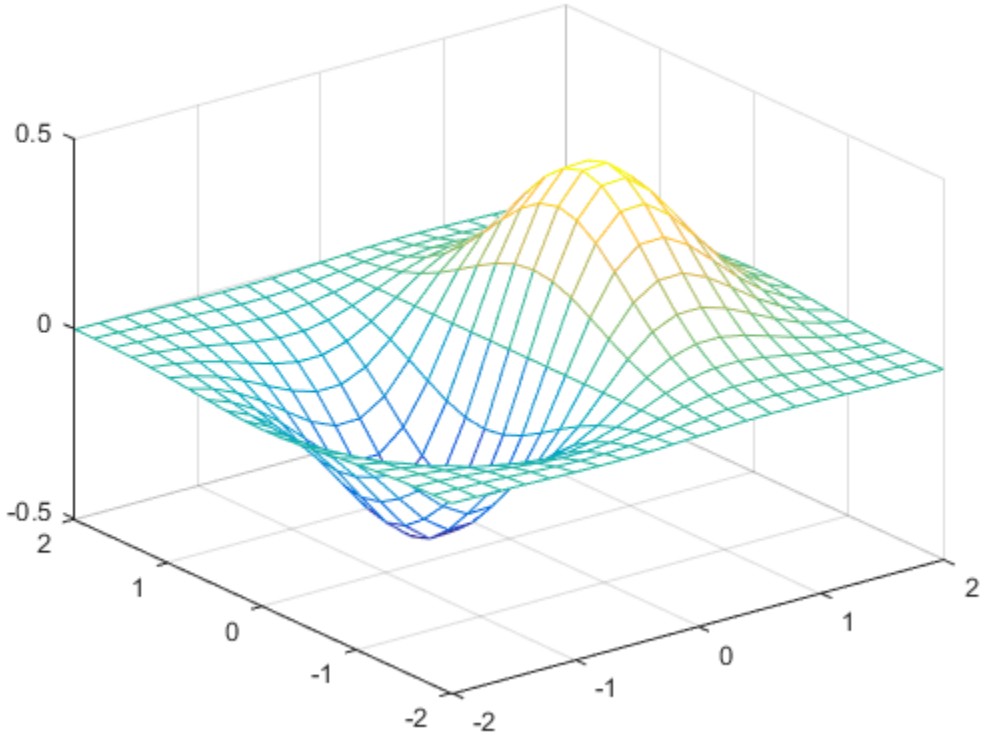
```
[X1,X2] = ndgrid(-2:.2:2, -2:.2:2);
```

Evaluate the function over the domain.

```
Z = X1 .* exp(-X1.^2 - X2.^2);
```

Generate a mesh plot of the function.

```
mesh(X1,X2,Z)
```



More About

Tips

The `ndgrid` function is similar to `meshgrid`, however `ndgrid` supports 1-D to N-D while `meshgrid` is restricted to 2-D and 3-D. The coordinates output by each function are the same, but the shape of the output arrays in the first two dimensions are different. For grid vectors `x1gv`, `x2gv` and `x3gv` of length `M`, `N` and `P` respectively, `ndgrid(x1gv, x2gv)` will output arrays of size `M`-by-`N` while `meshgrid(x1gv, x2gv)` outputs arrays of size `N`-by-`M`. Similarly, `ndgrid(x1gv, x2gv, x3gv)` will output arrays of size `M`-by-`N`-by-`P` while `meshgrid(x1gv, x2gv, x3gv)` outputs arrays of size `N`-by-`M`-by-`P`.

See “Grid Representation” in the MATLAB Mathematics documentation for more information.

- “Interpolating Gridded Data”

See Also

`griddedInterpolant` | `meshgrid` | `mesh` | `surf`

Introduced before R2006a

ndims

Number of array dimensions

Syntax

```
N = ndims(A)
```

Description

`N = ndims(A)` returns the number of dimensions in the array `A`. The number of dimensions is always greater than or equal to *2*. The function ignores trailing singleton dimensions, for which `size(A,dim) = 1`.

Examples

Find Dimensions of Vector

Create a row vector.

```
A = 1:5;
```

Find the number of dimensions in the vector.

```
ndims(A)
```

```
ans =

     2
```

The result is `2` because the vector has a size of 1-by-5.

Find Dimensions of Cell Array

Create a cell array of strings.

```
C{1,1,1} = 'cell_1';
C{1,1,2} = 'cell_2';
C{1,1,3} = 'cell_3'
```

```
C(:,:,1) =

    'cell_1'


C(:,:,2) =

    'cell_2'


C(:,:,3) =

    'cell_3'
```

Find the number of dimensions of the cell array.

```
ndims(A)
```

```
ans =

     3
```

The result is `3` because the cell array has a size of 1-by-1-by-3.

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. This includes numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, calendarDuration arrays, tables, structure arrays, cell arrays, and object arrays.

More About

Algorithms

The number of dimensions in an array is the same as the length of the size vector of the array. In other words, `ndims(A) = length(size(A))`.

See Also

`length` | `size`

Introduced before R2006a

ne, ~=

Determine inequality

Syntax

```
A ~= B
ne(A,B)
```

Description

`A ~= B` returns a logical array with elements set to logical `1` (`true`) where arrays `A` and `B` are not equal; otherwise, it returns logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays. `ne` returns logical `1` (`true`) where `A` or `B` have NaN or undefined categorical elements.

`ne(A,B)` is an alternative way to execute `A ~= B`, but is rarely used. It enables operator overloading for classes.

Examples

Inequality of Two Vectors

Create two vectors containing both real and imaginary numbers.

```
A = [1+i 3 2 4+i];
B = [1 3+i 2 4+i];
```

Compare the two vectors for inequality.

```
A ~= B
```

```
ans =

     1     1     0     0
```

The `ne` function tests both real and imaginary parts for inequality, and returns logical `1` (`true`) where one or both parts are not equal.

Find Characters in String

Create a string of characters.

```
M = 'masterpiece';
```

Test the string for the presence of a specific character using `~=`.

```
M ~= 'n'
```

```
ans =

     1     1     1     1     1     1     1     1     1     1     1
```

The value of logical `1` (`true`) in the vector indicates the absence of the character `'n'`. The character is not present in the string.

Find Values in Categorical Array

Create a categorical array.

```
A = categorical({'heads' 'heads' 'tails'; 'tails' 'heads' 'tails'})
```

```
A =

     heads      heads      tails
     tails      heads      tails
```

The array has two categories: `'heads'` and `'tails'`.

Find all values not in the `'heads'` category.

```
A ~= 'heads'
```

```
ans =

     0     0     1
     1     0     1
```

A value of logical `1` (`true`) indicates a value not in the category. Since `A` only has two categories, `A ~= 'heads'` returns the same answer as `A == 'tails'`.

Compare the rows of A for inequality.

```
A(1,:) ~= A(2,:)
```

```
ans =

     1     0     0
```

The function returns logical 1 (true) where the rows have unequal category values.

Compare Floating-Point Numbers

Some floating-point numbers cannot be represented exactly in binary form. This leads to small differences in results that the ~= operator reflects.

Perform a few subtraction operations on a floating-point number and store the result in C.

```
C = 0.5-0.4-0.1
```

```
C =

  -2.7756e-17
```

Intuitively, C should be equal to *exactly* 0. Its small value is due to the nature of floating-point arithmetic.

Compare C to zero for inequality.

```
C ~= 0
```

```
ans =

     1
```

The result is logical 1 (true).

Compare floating-point numbers using a tolerance, tol, instead of ~=.

```
tol = eps;
abs(C-0) > tol
```

```
ans =

     0
```

The two numbers, C and 0, are closer to one another than two consecutive floating-point numbers. They are essentially equal.

Inequality of Two Datetime Arrays

Compare the elements of two `datetime` arrays for inequality.

Create two `datetime` arrays in different time zones.

```
t1 = [2014,04,14,9,0,0;2014,04,14,10,0,0];
A = datetime(t1,'TimeZone','America/Los_Angeles');
A.Format = 'd-MMM-y HH:mm:ss Z'
```

```
A =

   14-Apr-2014 09:00:00 -0700
   14-Apr-2014 10:00:00 -0700
```

```
t2 = [2014,04,14,12,0,0;2014,04,14,12,30,0];
B = datetime(t2,'TimeZone','America/New_York');
B.Format = 'd-MMM-y HH:mm:ss Z'
```

```
B =

   14-Apr-2014 12:00:00 -0400
   14-Apr-2014 12:30:00 -0400
```

Check where elements in A and B are not equal.

```
A~=B
```

```
ans =

     0
     1
```

Input Arguments

A — Left array

numeric array | logical array | character array | categorical array | datetime array | duration array

Left array, specified as a numeric array, logical array, character array, categorical array, datetime array, or duration array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is a categorical array, the other input can be a categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. If both inputs are categorical arrays that are not ordinal, they can have different sets of categories. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.

Complex Number Support: Yes

B — Right array

numeric array | logical array | character array | categorical array | datetime array | duration array

Right array, specified as a numeric array, logical array, character array, or categorical array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar input expands into an array of the same size as the other input.

If one input is a categorical array, the other input can be a categorical array, a cell array of strings, or a single string. A single string expands into a cell array of strings of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. If both inputs are categorical arrays that are not ordinal, they can have different sets of categories. See "Compare Categorical Array Elements" for more details.

If one input is a datetime array, the other input can be a datetime array, a date string, or a cell array of date strings.

If one input is a duration array, the other input can be a duration array or a numeric array. `eq` treats each numeric value as a number of standard (86400 s) days.
Complex Number Support: Yes

See Also

`eq` | `ge` | `gt` | `le` | `lt`

Introduced before R2006a

nearestNeighbor

Class: DelaunayTri

(Will be removed) Point closest to specified location

Note: `nearestNeighbor(DelaunayTri)` will be removed in a future release. Use `nearestNeighbor(triangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
PI = nearestNeighbor(DT,QX)
PI = nearestNeighbor(DT,QX,QY)
PI = nearestNeighbor(DT,QX,QY,QZ)
[PI,D] = nearestNeighbor(DT,QX,...)
```

Description

`PI = nearestNeighbor(DT,QX)` returns the index of the nearest point in `DT.X` for each query point location in `QX`.

`PI = nearestNeighbor(DT,QX,QY)` and `PI = nearestNeighbor(DT,QX,QY,QZ)` allow the query points to be specified in column vector format when working in 2-D and 3-D.

`[PI,D] = nearestNeighbor(DT,QX,...)` returns the index of the nearest point in `DT.X` for each query point location in `QX`. The corresponding Euclidean distances between the query points and their nearest neighbors are returned in `D`.

Note: `nearestNeighbor` is not supported for 2-D triangulations that have constrained edges.

Input Arguments

`DT`	Delaunay triangulation.
`QX`	The matrix `QX` is of size `mpts`-by-`ndim`, `mpts` being the number of query points and `ndim` the dimension of the space where the points reside.

Output Arguments

`PI`	`PI` is a column vector of point indices that index into the points `DT.X`. The length of `PI` is equal to the number of query points `mpts`
`D`	`D` is a column vector of length `mpts`.

Examples

Create a Delaunay triangulation:

```
x = rand(10,1);
y = rand(10,1);
dt = DelaunayTri(x,y);
```

Create query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
```

Find the nearest neighbors to the query points:

```
pid = nearestNeighbor(dt, qrypts)
```

See Also

`delaunayTriangulation` | `pointLocation` | `triangulation`

neighbors

Class: TriRep

(Will be removed) Simplex neighbor information

Note: `neighbors(TriRep)` will be removed in a future release. Use `neighbors(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
SN = neighbors(TR, SI)
```

Description

`SN = neighbors(TR, SI)` returns the simplex neighbor information for the specified simplices `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	`SI` is a column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the neighbor information for the entire triangulation is returned, where the neighbors associated with simplex `i` are defined by the `i`'th row of `SN`.

Output Arguments

`SN`	`SN` is an `m`-by-`n` matrix, where `m = length(SI)` , the number of specified simplices, and `n` is the number of neighbors per simplex. Each row `SN(i,:)` represents the neighbors of the simplex `SI(i)`.

By convention, the simplex opposite `vertex(j)` of simplex `SI(i)` is `SN(i,j)`. If a simplex has one or more boundary facets, the nonexistent neighbors are represented by `NaN`.

Definitions

A *simplex* is a triangle/tetrahedron or higher-dimensional equivalent. A *facet* is an edge of a triangle or a face of a tetrahedron.

Examples

Example 1

Load a 3-D triangulation and use `TriRep` to compute the neighbors of all tetrahedra.

```
load tetmesh
trep = TriRep(tet, X)
nbrs = neighbors(trep)
```

Example 2

Query a 2-D triangulation created using `DelaunayTri`.

```
x = rand(10,1)
y = rand(10,1)
dt = DelaunayTri(x,y)
```

Find the neighbors of the first triangle:

```
n1 = neighbors(dt, 1)
```

See Also

`triangulation` | `delaunayTriangulation`

NET

Summary of functions in MATLAB .NET interface

Description

Use the following functions to bring assemblies from the Microsoft .NET Framework into the MATLAB environment. The functions are implemented as a package called NET. To use these functions, prefix the function name with package name NET.

BeginInvoke	Initiate asynchronous .NET delegate call
cell	Create cell array
Combine	Convenience function for static .NET System.Delegate Combine method
enableNETfromNetworkDrive	Enable access to .NET commands from network drive
EndInvoke	Retrieve result of asynchronous call initiated by .NET System.Delegate BeginInvoke method
NET.addAssembly	Make .NET assembly visible to MATLAB
NET.Assembly	Members of .NET assembly
NET.convertArray	Convert numeric MATLAB array to .NET array
NET.createArray	Array for nonprimitive .NET types
NET.createGeneric	Create instance of specialized .NET generic type

NET.disableAutoRelease	Lock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release COM object
NET.enableAutoRelease	Unlock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB releases COM object
NET.GenericClass	Represent parameterized generic type definitions
NET.invokeGenericMethod	Invoke generic method of object
NET.isNETSupported	Check for supported Microsoft .NET Framework
NET.NetException	Capture error information for .NET exception
NET.setStaticProperty	Static property or field name
Remove	Convenience function for static .NET System.Delegate Remove method
RemoveAll	Convenience function for static .NET System.Delegate RemoveAll method

More About

- “Call .NET Libraries”

NET.addAssembly

Package: NET

Make .NET assembly visible to MATLAB

Syntax

```
asmInfo = NET.addAssembly(globalName)
asmInfo = NET.addAssembly(privateName)
```

Description

`asmInfo = NET.addAssembly(globalName)` loads a global .NET assembly into MATLAB.

`asmInfo = NET.addAssembly(privateName)` loads a private .NET assembly.

Limitations

- `NET.addAssembly` does not support assemblies generated by the MATLAB Compiler SDK product.

Input Arguments

`globalName`

One of the following:

- String representing the name of a global assembly.
- Instance of `System.Reflection.AssemblyName` class.

Default:

`privateName`

String representing the full path of a private assembly.

Default:

Output Arguments

`asmInfo`

`NET.Assembly` object containing names of the members of the assembly.

Examples

Display today's date using `System.DateTime` in the `mscorlib` assembly.

```
System.DateTime.Now.ToLongDateString
```

Call the `System.Windows.Forms.MessageBox.Show` method in the global assembly `System.Windows.Forms`.

```
asm = NET.addAssembly('System.Windows.Forms');
import System.Windows.Forms.*;
MessageBox.Show('Simple Message Box')
```

Display classes in the private assembly `NetSample.dll`.

```
asm = NET.addAssembly('c:\work\NetSample.dll');
asm.Classes
```

More About

Tips

- MATLAB dynamically loads the `mscorlib.dll` and `system.dll` assemblies from the .NET Framework class library the first time you type `"NET."` or `"System."`. You do not need to call `NET.addAssembly` to access classes in these assemblies.
- Refer to your .NET product documentation for the name of the assembly and its deployment type (global or private).

- "Assembly is Library of .NET Classes"
- MSDN AssemblyName Class

See Also

`NET.Assembly`

Introduced in R2009a

NET.Assembly class

Package: NET

Members of .NET assembly

Description

`NET.Assembly` object returns names of the members of an assembly.

Construction

The `NET.addAssembly` function creates an instance of this class.

Properties

`AssemblyHandle`

Instance of `System.Reflection.Assembly` class of the added assembly.

`Classes`

`nClass`x`1` cell array of class names of the assembly, where `nClass` is the number of classes

`Enums`

`nEnum`x`1` cell array of enums of the assembly, where `nEnum` is the number of enums

`Structures`

`nStruct`x`1` cell array of structures of the assembly, where `nStruct` is the number of structures

`GenericTypes`

`nGenType`x`1` cell array of generic types of the assembly, where `nGenType` is the number of generic types

`Interfaces`

`nInterface`x`1` cell array of interface names of the assembly, where `nInterface` is the number of interfaces

`Delegates`

`nDelegate`x`1` cell array of delegates of the assembly, where `nDelegate` is the number of delegates

See Also

`NET.addAssembly`

How To

- "What Classes Are in a .NET Assembly?"

NET.convertArray

Package: NET

Convert numeric MATLAB array to .NET array

Note: MATLAB automatically converts arrays to .NET types. For information, see "Using Arrays with .NET Applications".

Syntax

```
arrObj = NET.convertArray(V,'arrType',[m,n])
```

Description

`arrObj = NET.convertArray(V,'arrType',[m,n])` converts a MATLAB array `V` to a .NET array. Optional value `arrType` is a string representing a namespace-qualified .NET array type. To convert a MATLAB vector to a two-dimensional .NET array (either 1-by-n or m-by-1), use optional values `m,n`. If `V` is a MATLAB vector and you do not specify the number of dimensions and their sizes, the output `arrObj` is a one-dimensional .NET array.

If you do not specify `arrType`, MATLAB converts the type according to the MATLAB Primitive Type Conversion Table. See "Pass Primitive .NET Types".

Examples

Create a list `aList` of random `System.Int32` integers using the `System.Collections.Generic.List` class, and then sort the results:

```
% Create array R of random integers
nInt = 5;
R = randi(100,1,nInt);
%Create .NET array A
```

```
A = NET.convertArray(R,'System.Int32');
%Put A into aList, a generic collections list
aList = NET.createGeneric(...
    'System.Collections.Generic.List',...
    {'System.Int32'},A.Length);
aList.AddRange(A);
%Sort the values in aList
aList.Sort;
```

See Also

NET.createArray

Introduced in R2009a

NET.createArray

Package: NET

Array for nonprimitive .NET types

Syntax

```
array = NET.createArray(typeName,[m,n,p,...])
array = NET.createArray(typeName,m,n,p,...)
```

Description

`array = NET.createArray(typeName,[m,n,p,...])` creates an `m`-by-`n`-by-`p`-by-`...` `array` of type `typeName`, which is either a fully qualified .NET array type name (`namespace` and array type name) or an instance of the `NET.GenericClass` class, in case of arrays of generic type. `m,n,p,...` are the number of elements in each dimension of the array.

`array = NET.createArray(typeName,m,n,p,...)` alternative syntax for creating an array.

You cannot specify the lower bound of an array.

Examples

Create .NET Array of Generic Type

This example creates a .NET array of `List<Int32>` generic type.

```
genType = NET.GenericClass('System.Collections.Generic.List',...
  'System.Int32');
arr = NET.createArray(genType, 5)
```

```
arr =
```

```
  List<System*Int32>[] with properties:

          Length: 5
      LongLength: 5
            Rank: 1
        SyncRoot: [1x1 System.Collections.Generic.List<System*Int32>[]]
      IsReadOnly: 0
     IsFixedSize: 1
  IsSynchronized: 0
```

Create and Initialize Jagged Array

This example creates a jagged .NET array of 3 elements.

```
jaggedArray = NET.createArray('System.Double[]', 3)
```

```
jaggedArray =

  Double[][] with properties

          Length: 3
      LongLength: 3
            Rank: 1
        SyncRoot: [1x1 System.Double[][]]
      IsReadOnly: 0
     IsFixedSize: 1
  IsSynchronized: 0
```

Assign values:

```
jaggedArray(1) = [1, 3, 5, 7, 9];
jaggedArray(2) = [0, 2, 4, 6];
jaggedArray(3) = [11, 22];
```

Access first value of 3rd array:

```
jaggedArray(3,1)
```

```
ans =
    11
```

Create Jagged Array of Generic Type

This example creates a jagged array of `List<Double>` generic type.

```
genCls = NET.GenericClass('System.Collections.Generic.List[]',...
  'System.Double');
```

Create the array, `genArr`.

```
genArr = NET.createArray(genCls,3)
```

```
genArr =

  List<System*Double>[][] with properties:

           Length: 3
       LongLength: 3
             Rank: 1
         SyncRoot: [1x1 System.Collections.Generic.List`1[][]]
       IsReadOnly: 0
      IsFixedSize: 1
    IsSynchronized: 0
```

Create Nested Jagged Array

This command creates a jagged array of type `System.Double[][][]`.

```
netArr = NET.createArray('System.Double[][]', 3)
```

```
netArr =

  Double[][][] with properties:

           Length: 3
       LongLength: 3
             Rank: 1
         SyncRoot: [1x1 System.Double[][][]]
       IsReadOnly: 0
      IsFixedSize: 1
    IsSynchronized: 0
```

See Also

`NET.convertArray` | `NET.createGeneric`

Introduced in R2009a

NET.createGeneric

Package: NET

Create instance of specialized .NET generic type

Syntax

```
genObj = createGeneric(className,paramTypes,varargin ctorArgs)
```

Description

`genObj = createGeneric(className,paramTypes,varargin ctorArgs)` creates an instance `genObj` of generic type `className`.

Input Arguments

`className`	Fully qualified string with the generic type name.
`paramTypes`	Allowed cell types are: strings with fully qualified parameter type names and instances of the `NET.GenericClass` class when parameterization with another parameterized type is needed.
`ctorArgs`	Optional, variable length (0 to N) list of constructor arguments matching the arguments of the .NET generic class constructor intended to be invoked.

Output Arguments

`genObj`	Handle to the specialized generic class instance.

Examples

Create a List of System.Double Objects

Create a strongly typed list `dblLst` of objects of type `System.Double`:

```
t = NET.createGeneric('System.Collections.Generic.List',...
  {'System.Double'},10);
```

Create a List with Key/Value Pairs

Create the `kvpType` generic association where `Key` is of `System.Int32` type and `Value` is a `System.String`:

```
pe = NET.GenericClass('System.Collections.Generic.KeyValuePair',...
  'System.Int32','System.String');
```

Create the list `kvpList` with initial storage capacity for 10 key-value pairs:

```
st = NET.createGeneric('System.Collections.Generic.List',...
  {kvpType},10);
```

Add an Item to the List

Create a `KeyValuePair` item.

```
kvpItem = NET.createGeneric('System.Collections.Generic.KeyValuePair',...
  {'System.Int32','System.String'},42,'myString');
```

Add this item to the list `kvpList`.

```
kvpList.Add(kvpItem);
```

See Also

`NET.GenericClass`

Introduced in R2009a

NET.disableAutoRelease

Package: NET

Lock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release COM object

Syntax

```
A = NET.disableAutoRelease(obj)
```

Description

`A = NET.disableAutoRelease(obj)` locks a .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release the COM object. `obj` is a .NET object representing a COM Wrapper.

Before passing a .NET object representing a COM Wrapper to another process, lock the object using this function so that MATLAB does not release it. After using the object, call `NET.enableAutoRelease` to release the COM object.

Examples

The following user-defined function, `GetComApp.m`, has access to a COM object defined in the **pseudo-class** `ComNamespace.ComClass`. One of its methods is `readData`, with the signature:

```
System.String RetVal readData(ComNamespace.ComClass this, System.String strIn)
```

The input argument is defined in the **pseudo-class** `NetDocTest.MyClass`, which has a property named `MyApp`.

```
function GetComApp(obj)
comObj = ComNamespace.ComClass;
obj.MyApp = comObj;
% To pass a COM object to another process, lock the object
NET.disableAutoRelease(comObj);
end
```

The example in `NET.enableAutoRelease` shows how to call the `GetComApp` function.

More About

- "How MATLAB Handles System.__ComObject"

See Also

`NET.enableAutoRelease`

Introduced in R2010b

NET.enableAutoRelease

Package: NET

Unlock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB releases COM object

Syntax

```
A = NET.enableAutoRelease(obj)
```

Description

`A = NET.enableAutoRelease(obj)` releases the COM wrapper when the object goes out of scope, where `obj` is a .NET object representing a COM Wrapper.

Call this function only if the object was locked using `NET.disableAutoRelease`.

Examples

The following **pseudo-code** shows how to call a function (`GetComApp.m`, described in `NET.disableAutoRelease`) which returns a COM object. The object, `mainObj` of type `NetDocTest.MyClass`, has a property, `MyApp`. Call `GetComApp` to get a COM object, and use its `readData` method.

```
mainObj = NetDocTest.MyClass;
GetComApp(mainObj);
app = mainObj.MyApp;
app.readData('hello');
% Unlock the COM object
NET.enableAutoRelease(mainObj.MyApp);
```

More About

- "How MATLAB Handles System.__ComObject"

See Also

`NET.disableAutoRelease`

Introduced in R2010b

NET.GenericClass class

Package: NET

Represent parameterized generic type definitions

Description

The `NET.createGeneric` function uses instances of this class to create a generic specialization that requires parameterization with another parameterized type.

Construction

```
genType = NET.GenericClass (className, paramTypes)
```

Input Arguments

`className`

Fully qualified string containing the generic type name.

Default:

`paramTypes`

Optional, variable length (1 to N) list of types for the generic class parameterization. Allowed argument types are:

- Fully qualified string containing the generic type name.
- Instance of the `NET.GenericClass` class when deeper nested parameterization with another parameterized type is needed.

Default:

Examples

Create an instance of `System.Collections.Generic.List` of `System.Collections.Generic.KeyValuePair` generic associations where `Key` is of

`System.Int32` type and `Value` is a `System.String` class with initial storage capacity for 10 key-value pairs.

```
kvpType = NET.GenericClass(...
    'System.Collections.Generic.KeyValuePair',...
    'System.Int32', 'System.String');
kvpList = NET.createGeneric('System.Collections.Generic.List',...
    { kvpType }, 10);
```

See Also

`NET.createGeneric` | `NET.createArray` | `NET.invokeGenericMethod`

How To

- ".NET Generic Classes"

Introduced in R2009a

NET.invokeGenericMethod

Package: NET

Invoke generic method of object

Syntax

```
[varargout] = NET.invokeGenericMethod(obj, 'genericMethodName',
paramTypes, args, ...)
```

Description

`[varargout] = NET.invokeGenericMethod(obj, '`*`genericMethodName`*`', paramTypes, args, ...)` calls instance or static generic method *`genericMethodName`*.

Input Arguments

`obj`	Allowed argument types are: • Instances of class containing the generic method • Strings with fully qualified class name, if calling static generic methods • Instances of `NET.GenericClass` definitions, if calling static generic methods of a generic class
`genericMethodName`	Generic method name to invoke
`paramTypes`	Cell vector (1 to N) with the types for generic method parameterization, where allowed cell types are: • Strings with fully qualified parameter type name. • Instances of `NET.GenericClass` definitions, if using nested parameterization with another parameterized type
`args`	Optional, variable length (0 to N) list of method arguments

Output Arguments

`varargout`	Variable-length output argument list, `varargout`, from method *`genericMethodName`*

Examples

The following syntax calls a generic method that takes two parameterized types and returns a parameterized type:

```
a = NET.invokeGenericMethod(obj, ...
  'myGenericSwapMethod', ...
  {'System.Double', 'System.Double'}, ...
  5, 6);
```

To display generic methods in MATLAB, see the example “Display .NET Generic Methods Using Reflection”.

More About

- “Call .NET Generic Methods”

See Also

`NET.GenericClass` | `NET.createGeneric` | `varargout`

Introduced in R2009b

NET.isNETSupported

Check for supported Microsoft .NET Framework

Syntax

```
tf = NET.isNETSupported
```

Description

`tf = NET.isNETSupported` returns logical `1` (`true`) if a supported version of the Microsoft .NET Framework is found. Otherwise, it returns logical `0` (`false`) and you cannot use the .NET Framework in MATLAB.

Introduced in R2013a

NET.NetException class

Package: NET

Capture error information for .NET exception

Description

Process information from a `NET.NetException` object to handle .NET errors. This class is derived from `MException`.

Construction

`e = NET.NetException(msgID,errMsg,netObj)` constructs instance e of `NET.NetException` class.

Input Arguments

`msgID`

message identifier

`errMsg`

error message string

`netObj`

`System.Exception` object that caused the exception

Properties

`ExceptionObject`

`System.Exception` class causing the error.

Methods

Inherited Methods

See the methods of the base class `MException`.

Examples

Display error information after trying to load an unknown assembly:

```
try
  NET.addAssembly('C:\Work\invalidfile.dll')
catch e
  e.message;
  if(isa(e, 'NET.NetException'))
    eObj = e.ExceptionObject
  end
end
```

```
ans =
Message: Could not load file or assembly
  'file:///C:\Work\invalidfile.dll' or
  one of its dependencies. The system cannot
  find the file specified.
Source: mscorlib
HelpLink:

eObj =
  FileNotFoundException with properties:

           Message: [1x1 System.String]
          FileName: [1x1 System.String]
         FusionLog: [1x1 System.String]
              Data: [1x1 System.Collections.ListDictionaryInternal]
    InnerException: []
        TargetSite: [1x1 System.Reflection.RuntimeMethodInfo]
        StackTrace: [1x1 System.String]
          HelpLink: []
            Source: [1x1 System.String]
```

See Also

`MException`

How To

- Class Attributes
- Property Attributes

Introduced in R2009b

NET.setStaticProperty

Package: NET

Static property or field name

Syntax

```
NET.setStaticProperty('propName', value)
```

Description

`NET.setStaticProperty('propName', value)` sets the static property or field name specified in the string `propName` to the given `value`.

Examples

To set the `myStaticProperty` in the given class and namespace, use the syntax:

```
NET.setStaticProperty('MyTestObject.MyClass.myStaticProperty', 5);
```

Introduced in R2009b

nccreate

Create variable in NetCDF file

Syntax

```
nccreate(filename,varname)
nccreate(filename,varname,Name,Value)
```

Description

`nccreate(filename,varname)` creates a scalar `double` variable named `varname` in the NetCDF file specified by `filename`. If `filename` does not exist, then `nccreate` creates the file using the `netcdf4_classic` format.

`nccreate(filename,varname,Name,Value)` creates a variable with additional options specified by one or more name-value pair arguments. For example, to create a nonscalar variable, use the `Dimensions` name-value pair argument.

Examples

Create New Variables in NetCDF File

Create a NetCDF file named `myexample.nc` that contains a variable named `Var1`.

```
nccreate('myexample.nc','Var1')
```

Create a second variable in the same file.

```
nccreate('myexample.nc','Var2')
```

Display the contents of the NetCDF file.

```
ncdisp('myexample.nc')
```

```
Source:
```

```
           pwd\myexample.nc
Format:
           netcdf4_classic
Variables:
    Var1
           Size:       1x1
           Dimensions:
           Datatype:   double
    Var2
           Size:       1x1
           Dimensions:
           Datatype:   double
```

Create Variable and Specify Dimensions and File Format

Create a new two-dimensional variable named `peaks` in a classic (NetCDF 3) format file named `myncclassic.nc`. Use the `'Dimensions'` name-value pair argument to specify the names and lengths of the two dimensions. Use the `'Format'` name-value pair argument to specify the file format.

```
nccreate('myncclassic.nc','peaks',...
          'Dimensions',{'r',200,'c',200},...
          'Format','classic')
```

Write data to the variable.

```
ncwrite('myncclassic.nc','peaks',peaks(200))
```

Display the contents of the NetCDF file.

```
ncdisp('myncclassic.nc')
```

```
Source:
           pwd\myncclassic.nc
Format:
           classic
Dimensions:
           r = 200
           c = 200
Variables:
    peaks
           Size:       200x200
           Dimensions: r,c
```

```
            Datatype:   double
```

Input Arguments

`filename` — File name

string

File name, specified as a string. The file is an existing NetCDF file, or the name you want to assign to a new NetCDF file.

Example: `'myFile.nc'`

`varname` — Name of new variable

string

Name of the new variable, specified as a string.

Example: `'myVar'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`nccreate('myFile.nc','Var1','Datatype','double','Format','classic')` creates a variable named `Var1` of type `NC_DOUBLE` in a NetCDF 3 file named `myFile.nc`.

`'Dimensions'` — Dimensions of variable

cell array

Dimensions of the new variable, specified as the comma-separated pair consisting of `'Dimensions'` and a cell array. The cell array lists the dimension name as a string followed by its numerical length, in this form: `{dname1,dlength1,dname2,dlength2, ...}`. The `dname1` input is the name of the first dimension, `dlength1` is the length of the first dimension, `dname2` is the name of the second dimension, and so on. If a dimension exists, specifying its length is optional. A variable with a single dimension is always treated as a column vector.

Use `Inf` to specify an unlimited dimension. A `netcdf4` format file can have any number of unlimited dimensions in any order. All other formats can have only one unlimited dimension per file and it must be specified last in the cell array.

`nccreate` creates the dimension at the same location as the variable. For `netcdf4` format files, you can specify a different location for the dimension using a fully qualified dimension name.

Example: `'Dimensions',{'dim1',100,'dim2',150,'dim3','Inf'}`

`'Datatype'` — MATLAB data type

`'double'` (default) | string

MATLAB data type, specified as the comma-separated pair consisting of `'Datatype'` and a string. When `nccreate` creates the variable in the NetCDF file, it uses a corresponding NetCDF datatype. This table lists valid values for `'Datatype'` and the corresponding NetCDF variable type that `nccreate` creates.

Value of `Datatype`	NetCDF Variable Type
`'double'`	`NC_DOUBLE`
`'single'`	`NC_FLOAT`
`'int64'`	`NC_INT64*`
`'uint64'`	`NC_UINT64*`
`'int32'`	`NC_INT`
`'uint32'`	`NC_UINT*`
`'int16'`	`NC_SHORT`
`'uint16'`	`NC_USHORT*`
`'int8'`	`NC_BYTE`
`'uint8'`	`NC_UBYTE*`
`'char'`	`NC_CHAR`

* These data types are only available when the file is a `netcdf4` format file.

Example: `'Datatype','uint16'`

`'Format'` — NetCDF file format

`'netcdf4_classic'` (default) | string

NetCDF file format, specified as the comma-separated pair consisting of `'Format'` and one of the following strings.

Value of `Format`	Description
`'classic'`	NetCDF 3
`'64bit'`	NetCDF 3, with 64-bit offsets
`'netcdf4_classic'`	NetCDF 4 classic model
`'netcdf4'`	NetCDF 4 model (Use this format to enable group hierarchy)

If `varname` specifies a group (for example, `'/grid3/temperature'`), then `nccreate` sets the value of `Format` to `'netcdf4'`.

Example: `'Format','classic'`

`'FillValue'` — Replacement value for missing values

scalar | `'disable'`

Replacement value for missing values, specified as the comma-separated pair consisting of `'FillValue'` and a scalar or `'disable'`. The default value is specified by the NetCDF library. To disable replacement values, specify `'FillValue','disable'`.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'FillValue',NaN`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

`'ChunkSize'` — Chunk size along each dimension

vector

Chunk size along each dimension, specified as the comma-separated pair consisting of `'ChunkSize'` and a vector. The first element specifies the number of rows, the second element specifies the number of columns, the third element specifies the length of the third dimension, and so on. The default value is specified by the NetCDF library.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'ChunkSize',[5 6 9]`

Data Types: `double`

`'DeflateLevel'` — Amount of compression

`0` (default) | scalar value between `0` and `9`

Amount of compression, specified as the comma-separated pair consisting of `'DeflateLevel'` and a scalar value between `0` and `9`. `0` indicates no compression and `9` indicates the most compression.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'DeflateLevel',5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'Shuffle'` — Status of shuffle filter

`false` (default) | `true`

Status of the shuffle filter, specified as the comma-separated pair consisting of `'Shuffle'` and `false` or `true`. `false` disables the shuffle filter and `true` enables it. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'Shuffle',true`

Data Types: `logical`

More About

- "Export to NetCDF Files"

See Also

`ncdisp` | `ncinfo` | `ncwrite` | `ncwriteschema` | `netcdf`

Introduced in R2011a

ncdisp

Display contents of NetCDF data source in Command Window

Syntax

```
ncdisp(source)
ncdisp(source,location)
ncdisp(source,location,dispFormat)
```

Description

`ncdisp(source)` displays all the groups, dimensions, variable definitions, and all attributes in the NetCDF data source, `source`, as text in the Command Window.

`ncdisp(source,location)` displays information about the variable or group specified by `location`.

`ncdisp(source,location,dispFormat)` displays the contents of the NetCDF data source, in the display format specified by `dispFormat`.

Examples

Display Contents of NetCDF File

Display the contents of the example NetCDF file, `example.nc`.

```
ncdisp('example.nc')
```

```
Source:
           matlabroot\toolbox\matlab\demos\example.nc
Format:
           netcdf4
Global Attributes:
           creation_date = '29-Mar-2010'
Dimensions:
           x = 50
           y = 50
           z = 5
```

```
Variables:
    avagadros_number
           Size:       1x1
           Dimensions:
           Datatype:   double
           Attributes:
                       description = 'this variable has no dimensions'
    temperature
           Size:       50x1
           Dimensions: x
           Datatype:   int16
           Attributes:
                       scale_factor = 1.8
                       add_offset   = 32
                       units        = 'degrees_fahrenheight'
    peaks
           Size:       50x50
           Dimensions: x,y
           Datatype:   int16
           Attributes:
                       description = 'z = peaks(50);'
Groups:
    /grid1/
        Attributes:
                   description = 'This is a group attribute.'
        Dimensions:
                   x    = 360
                   y    = 180
                   time = 0     (UNLIMITED)
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16

    /grid2/
        Attributes:
                   description = 'This is another group attribute.'
        Dimensions:
                   x    = 360
                   y    = 180
                   time = 0     (UNLIMITED)
        Variables:
            temp
```

```
                Size:       []
                Dimensions: x,y,time
                Datatype:   int16
```

MATLAB displays all the groups, dimensions, and variable definitions in `example.nc`.

Display Contents of NetCDF Variable

Display the contents of the variable `peaks` in the file, `example.nc`.

```
ncdisp('example.nc','peaks')
```

```
Source:
           matlabroot\toolbox\matlab\demos\example.nc
Format:
           netcdf4
Dimensions:
           x = 50
           y = 50
Variables:
    peaks
           Size:       50x50
           Dimensions: x,y
           Datatype:   int16
           Attributes:
                       description = 'z = peaks(50);'
```

Display Contents of NetCDF File and Hide Attributes

Display only the group hierarchy and variable definitions of the example file, `example.nc`.

```
ncdisp('example.nc','/','min')
```

```
Source:
           matlabroot\toolbox\matlab\demos\example.nc
Format:
           netcdf4
Variables:
    avagadros_number
           Size:       1x1
           Dimensions:
           Datatype:   double
    temperature
           Size:       50x1
           Dimensions: x
```

```
            Datatype:   int16
    peaks
            Size:       50x50
            Dimensions: x,y
            Datatype:   int16
Groups:
    /grid1/
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16

    /grid2/
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16
```

Input Arguments

`source` — Name of NetCDF file

string

Name of a NetCDF file, specified as a string. `source` also can be the URL of an OPeNDAP NetCDF data source that resolves to a NetCDF file or a variable in a NetCDF file.

Example: `'myNetCDFfile.nc'`

Data Types: `char`

`location` — Location of variable or group

`'/'` (default) | string

Location of a variable or group in the NetCDF file, specified as a string. Set `location` to `'/'`(forward slash) to display the entire contents of the file.

Data Types: `char`

`dispFormat` — Display format

`'full'` (default) | `'min'`

Display format, specified as one of the following strings.

`'full'`	Display group hierarchy with dimensions, attributes, and variable definitions.
`'min'`	Display group hierarchy and variable definitions.

Data Types: `char`

More About

Tips

- If `source` is an OPeNDAP URL with string constraints, use the syntax, `ncdisp(source)` with no other input arguments.
- "Import NetCDF Files and OPeNDAP Data"

See Also

`ncinfo` | `ncread` | `ncreadatt` | `ncwrite` | `netcdf`

Introduced in R2011a

ncinfo

Return information about NetCDF data source

Syntax

```
finfo = ncinfo(source)
vinfo = ncinfo(source,varname)
ginfo = ncinfo(source,groupname)
```

Description

`finfo = ncinfo(source)` returns information in the structure `finfo` about the entire NetCDF data source specified by `source`, where source can be the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

`vinfo = ncinfo(source,varname)` returns information in the structure `vinfo` about the variable `varname` in `source`.

`ginfo = ncinfo(source,groupname)` returns information in the structure `ginfo` about the group `groupname` in `source` (only NetCDF4 data sources).

Note: Use `ncdisp` for visual inspection of a NetCDF source.

Input Arguments

`source`

Text string specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

Default:

`varname`

Text string specifying the name of a variable in a NetCDF file or OPeNDAP data source.

Default:

`groupname`

Text string specifying the name of a group in a NetCDF file or OPeNDAP data source.

Default:

Output Arguments

`finfo`

A structure with the following fields.

Field	Description	
`Filename`	NetCDF file name or OPeNDAP URL	
`Name`	`'/'`, indicating the full file	
`Dimensions`	An array of structures with these fields:	
	`Name`	Dimension name
	`Length`	Current length of dimension
	`Unlimited`	Boolean flag, true for unlimited dimensions
`Variables`	An array of structures with these fields:	
	`Name`	Variable name
	`Dimensions`	Associated dimensions
	`Size`	Current variable size
	`Datatype`	MATLAB datatype
	`Attributes`	Associated variable attributes
	`ChunkSize`	Chunk size, if defined. `[]` otherwise
	`FillValue`	Fill value of the variable.
	`DeflateLevel`	Deflate filter level, if enabled.
	`Shuffle`	Shuffle filter enabled flag

Field	Description	
`Attributes`	An array of global attributes with these fields:	
	`Name`	Attribute name
	`Value`	Attribute value
`Groups`	An array of groups present in the file, for `netcdf4` files; An empty array ([ ]) for all other NetCDF file formats.	
`Format`	The format of the NetCDF file	

`vinfo`

A structure containing only the variable fields from `finfo`.

Field	Description
`Filename`	NetCDF file name
`Name`	Name of the variable
`Dimensions`	Dimensions of the variable
`Size`	Size of the current variable
`Datatype`	MATLAB datatype
`Attributes`	Attributes associated with the variable
`ChunkSize`	Chunk size, if defined. `[ ]` otherwise.
`FillValue`	Fill value used in the variable.
`DeflateLevel`	Deflate filter level, if enabled.
`Shuffle`	Shuffle filter enabled flag
`Format`	The format of the NetCDF file

`ginfo`

A structure containing only the group fields from `finfo`.

Field	Description
`Filename`	NetCDF file name
`Name`	Name of the group
`Dimensions`	Only dimensions defined in the specified group

Field	Description
`Variables`	Only variables defined in the specified group
`Attributes`	Attributes associated with the variable
`Groups`	Names of groups, if defined. `[ ]` otherwise.
`Format`	The format of the NetCDF file

Examples

Search for dimensions with names that start with the character `x` in the file.

```
finfo = ncinfo('example.nc');
disp(finfo);
dimNames = {finfo.Dimensions.Name};
dimMatch = strncmpi(dimNames,'x',1);
disp(finfo.Dimensions(dimMatch));
```

Obtain the size of a variable and check if it has any unlimited dimensions.

```
vinfo = ncinfo('example.nc','peaks');
varSize = vinfo.Size;
disp(vinfo);
hasUnLimDim = any([vinfo.Dimensions.Unlimited]);
```

Find all unlimited dimensions defined in a group.

```
ginfo = ncinfo('example.nc','/grid2/');
unlimDims = [ginfo.Dimensions.Unlimited];
disp(ginfo.Dimensions(unlimDims));
```

See Also

`ncdisp` | `ncwrite` | `ncread` | `ncwriteschema` | `netcdf`

Introduced in R2011a

ncread

Read data from variable in NetCDF data source

Syntax

```
vardata = ncread(source,varname)
vardata = ncread(source,varname,start,count,stride)
```

Description

`vardata = ncread(source,varname)` reads data from the variable `varname` in `source`, which can be either the name of a NetCDF file or an OPeNDAP NetCDF data source.

`vardata = ncread(source,varname,start,count,stride)` reads data from the variable `varname` in `source` beginning at the location given by `start`. `count` specifies the number of elements to read along the corresponding dimension. The optional argument `stride` specifies the inter-element spacing along each dimension.

Input Arguments

`source`

Text string specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

Default:

`varname`

Text string specifying the name of a variable in the NetCDF file or OPeNDAP NetCDF data source.

Default:

`start`

For an *N*-dimensional variable, `start` is a vector of length *N* of indices specifying the starting location. Indices are 1-based.

`count`

Vector of length *N* specifying the number of elements to read along the corresponding dimensions. If a particular element of `count` is `Inf`, `ncread` reads data until the end of the corresponding dimension.

`stride`

Optional argument that specifies the inter-element spacing along each dimension.

Default: Vector of 1s (ones)

Output Arguments

`vardata`

The data in the variable. `ncread` uses the MATLAB datatype that is the closest type to the corresponding NetCDF datatype, except when at least one of `_FillValue`, `scale_offset`, and `add_offset` variables attribute is present. `ncread` applies the following attribute conventions, in sequence, to `vardata` if the corresponding attribute exists for this variable:

- If the `_FillValue` attribute exists, `ncread` replaces values in `vardata` equal to the value of `_FillValue` with `NaN`s. If the`_FillValue` attribute does not exist, `ncread` queries the NetCDF library for the variable's fill value.
- If the `scale_factor` attribute exists, `ncread` multiplies `vardata` by the value of the `scale_factor` attribute.
- If the `add_offset` attribute exists, `ncread` adds the value of the `add_offset` attribute to `vardata`.

Examples

Read and display the data in the `peaks` variable in the example file.

```
ncdisp('example.nc','peaks');
peaksData  = ncread('example.nc','peaks');
peaksDesc  = ncreadatt('example.nc','peaks','description');
surf(double(peaksData));
title(peaksDesc);
```

Subsample the peaks data by a factor of 2 (read every other value along each dimension).

```
subsetdata = ncread('example.nc','peaks',...
                    [1 1], [Inf Inf], [2 2]);
surf(double(subsetdata));
```

See Also

ncdisp | ncinfo | netcdf | ncwrite | ncreadatt

Introduced in R2011a

ncreadatt

Read attribute value from NetCDF data source

Syntax

```
attvalue = ncreadatt(source,location,attname)
```

Description

`attvalue = ncreadatt(source,location,attname)` reads the attribute `attname` from the group or variable specified by `location` in `source`, where source is the name of a NetCDF file or the URL of a NetCDF data source.

Input Arguments

`source`

Text string specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

`location`

Text string specifying a group or variable in the NetCDF data source. To read global attributes, set `location` to `'/'` (forward slash).

`attname`

Text string specifying the name of an attribute that you want to read in the NetCDF data source.

Output Arguments

`attvalue`

Data associated with the attribute.

Examples

Read a global attribute.

```
creation_date = ncreadatt('example.nc','/','creation_date');
disp(creation_date);
```

Read an attribute associated with a variable.

```
scale_factor = ncreadatt('example.nc','temperature','scale_factor');
disp(scale_factor);
```

Read an attribute associated with a group (`netcdf4` format files only).

```
desc_value = ncreadatt('example.nc','/grid2','description');
disp(desc_value);
```

See Also

`ncdisp` | `ncinfo` | `ncread` | `netcdf` | `ncwriteatt`

Introduced in R2011a

ncwrite

Write data to NetCDF file

Syntax

```
ncwrite(filename,varname,vardata)
ncwrite(filename,varname,vardata,start,stride)
```

Description

`ncwrite(filename,varname,vardata)` writes the numerical or char data in `vardata` to an existing variable `varname` in the NetCDF file `filename`. `ncwrite` writes the data in `vardata` starting at the beginning of the variable and extends unlimited dimensions automatically, if needed.

If the NetCDF file or the variable do not exist, use `nccreate` to create them first.

`ncwrite(filename,varname,vardata,start,stride)` writes `vardata` to an existing variable `varname` in file `filename` beginning at the location given by `start`. `stride` is an optional argument that specifies the inter-element spacing of the data written. Use this syntax to append data to an existing variable or write partial data.

Input Arguments

`filename`

Text string specifying the name of a NetCDF file. If the file does not exist, use `nccreate` to create it first.

Default:

`varname`

Text string specifying the name of a variable in a NetCDF file. If the variable does not exist, use `nccreate` to create it first.

`vardata`

Data to write to the variable in the NetCDF file.

`start`

For an *N*-dimensional variable, `start` is a vector of indices of length *N* specifying the starting location. Indices are 1-based.

`stride`

(Optional) Vector of length *N*, specifying the inter-element spacing.

Default: Vector of ones

Examples

Create a new `netcdf4_classic` file, and write a scalar variable with no dimensions. Add the creation time as a global attribute.

```
nccreate('myfile.nc','pi');
ncwrite('myfile.nc','pi',3.1);
ncwriteatt('myfile.nc','/','creation_time',datestr(now));
% overwrite existing data
ncwrite('myfile.nc','pi',3.1416);
ncdisp('myfile.nc');
```

Create a `netcdf4_classic` file with a variable defined on an unlimited dimension. Write data incrementally to the variable.

```
nccreate('myncfile.nc','vmark',...
         'Dimensions', {'time', inf, 'cols', 6},...
         'ChunkSize',  [3 3],...
         'DeflateLevel', 2);
ncwrite('myncfile.nc','vmark', eye(3),[1 1]);
varData = ncread('myncfile.nc','vmark');
disp(varData);
ncwrite('myncfile.nc','vmark',fliplr(eye(3)),[1 4]);
varData = ncread('myncfile.nc','vmark');
disp(varData);
```

More About

Tips

- If the variable `varname` already exists, `ncwrite` expects the datatype of `vardata` to match the NetCDF variable data type.
- If the variable `varname` has a `_FillValue`, `scale_factor` or `add_offset` attribute, `ncwrite` expects data in `double` format and casts `vardata` to the NetCDF data type, after applying the following attribute conventions in sequence:

  1. Subtract the value of' the`add_offset` attribute from `vardata`.
  2. Divide `vardata` by the value of the `scale_factor` attribute.
  3. Replace `NaN`s in `vardata` by the value of the `_FillValue` attribute. If this attribute does not exist, `ncwrite` tries to use the fill value for this variable as reported by the NetCDF library.

See Also

`ncdisp` | `ncread` | `ncinfo` | `netcdf` | `ncwriteatt` | `nccreate`

Introduced in R2011a

ncwriteatt

Write attribute to NetCDF file

Syntax

```
ncwriteatt(filename,location,attname,attvalue)
```

Description

`ncwriteatt(filename,location,attname,attvalue)` creates or modifies the attribute specified by `attname` in the group or variable specified by `location`, in the NetCDF file specified by `filename`. `attvalue` can be a numeric vector or a string.

Input Arguments

`filename`

Text string specifying the name of a NetCDF file

`location`

Text string specifying a group or variable in the NetCDF file. To write global attributes, set `location` to `'/'` (forward slash).

`attname`

Text string specifying the name of an existing attribute in a NetCDF file or the name of the attribute that you want to create.

`attvalue`

Numeric vector or a string.

Examples

Create a global attribute.

```
copyfile(which('example.nc'),'myfile.nc');
fileattrib('myfile.nc','+w');
ncdisp('myfile.nc');
ncwriteatt('myfile.nc','/','creation_date',datestr(now));
ncdisp('myfile.nc');
```

Modify an existing attribute.

```
copyfile(which('example.nc'),'myfile.nc');
fileattrib('myfile.nc','+w');
ncdisp('myfile.nc','peaks');
ncwriteatt('myfile.nc','peaks','description','Output of PEAKS');
ncdisp('myfile.nc','peaks');
```

See Also

ncdisp | ncreadatt | ncwrite | ncread | nccreate | netcdf

Introduced in R2011a

ncwriteschema

Add NetCDF schema definitions to NetCDF file

Syntax

```
ncwriteschema(filename,schema)
```

Description

`ncwriteschema(filename,schema)` creates or adds attributes, dimensions, variable definitions and group structure defined in `schema` to the file `filename`.

Use `ncwriteschema` in combination with `ncinfo` to create a new NetCDF file based on the schema of an existing file. You can also use `ncwriteschema` to add variable definitions, attributes, dimensions, or group structure to an existing file.

Note: `ncwriteschema` does not write variable data. Use `ncwrite` to write data to the created variables. Created unlimited dimensions will have an initial size of 0 until you write data.

Note: `ncwriteschema` cannot change the format of an existing file. It cannot redefine existing variables and dimensions in `filename`. If your schema contains attributes, dimensions, variable definitions, or a group structure that already exist in the file, `writeschema` issues a warning but continues processing.

Input Arguments

`filename`

Text string specifying the name of a NetCDF file. If `filename` does not exist, `ncwriteschema` creates a new file using the `netcdf4_classic` format, unless the `Format` field in `schema` specifies another format.

Default:

`schema`

A structure, or array of structures, representing either a dimension, variable, an entire NetCDF file, or a `netcdf4` group. A group or file schema can contain a dimension or variable schema, or both. You can use the output returned by `ncinfo` as a `schema` structure. The following table lists the fields in the various types of schema structures. Optional fields are marked with asterisk (*).

Schema Type	Structure Field	Description
Group/File Schema	`Name`	Text string identifying the group name. Use `'/'` to indicate the entire file.
	`Dimensions*`	Dimension schema
	`Variables*`	Variable schema
	`Attributes*`	Structure array of group/global attributes with Name and Value fields
	`Format*`	Text string identifying a NetCDF file format
Dimension schema	`Name`	Text string identifying the dimension
	`Length`	Length of the dimension. Can be `Inf`.
	`Unlimited*`	Boolean flag indicating if the dimension is unlimited
	`Format*`	Text string identifying a NetCDF file format
Variable schema	`Name`	Text string identifying a variable name
	`Dimensions`	Variable's dimension schema
	`Datatype`	Text string identifying a MATLAB datatype
	`Attributes*`	Structure array of variable attributes with Name and Value fields
	`ChunkSize*`	Numeric value specifying chunk size of the variable
	`FillValue*`	Character or numeric fill value

Schema Type	Structure Field	Description
	`DeflateValue*`	Deflate compression level
	`Shuffle*`	Boolean flag to turn on the Shuffle filter
	`Format*`	Text string identifying a NetCDF file format

Default:

Examples

Create a classic format file with two dimension definitions.

```
mySchema.Name   = '/';
mySchema.Format = 'classic';
mySchema.Dimensions(1).Name   = 'time';
mySchema.Dimensions(1).Length = Inf;
mySchema.Dimensions(2).Name   = 'rows';
mySchema.Dimensions(2).Length = 10;
ncwriteschema('emptyFile.nc', mySchema);
ncdisp('emptyFile.nc');
```

Create a `netcdf4_classic` format file to store a single variable from an existing file. First use `ncinfo` to get the schema of the peaks variable from the file. Then use `ncwriteschema` to create a NetCDF file, defining the peaks variable. Use `ncread` to get the data associated with the peaks variable and then use `ncwrite` to write the data to the variable in the new NetCDF file.

```
myVarSchema = ncinfo('example.nc','peaks');
ncwriteschema('peaksFile.nc',myVarSchema);
peaksData   = ncread('example.nc','peaks');
ncwrite('peaksFile.nc','peaks',peaksData);
ncdisp('peaksFile.nc');
```

See Also

`ncdisp` | `ncinfo` | `ncwrite` | `ncread` | `netcdf`

Introduced in R2011a

netcdf

Interact directly with NetCDF Library

Description

Functions in the MATLAB `netcdf` package provide interfaces to dozens of functions in the NetCDF library. The MATLAB functions enable reading data from and writing data to NetCDF files (known as *data sets* in NetCDF terminology). To use these functions, you should be familiar with the NetCDF C Interface.

MATLAB supports NetCDF version 4.1.3.

In most cases, the syntax of the MATLAB function matches the syntax of the NetCDF library function. The functions are implemented as a package called `netcdf`. To use these functions, prefix the function name with the package name, `netcdf`. For example, to call the NetCDF library routine used to open existing NetCDF files, use the following MATLAB syntax:

```
ncid = netcdf.open(ncfile,mode);
```

For copyright information, see the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

Note: For information about MATLAB support for the Common Data Format (CDF), which is a completely separate and incompatible format, see `cdflib`.

Library Functions

netcdf.getChunkCache	Retrieve chunk cache settings for NetCDF library
netcdf.inqLibVers	Return NetCDF library version information
netcdf.setChunkCache	Set default chunk cache settings for NetCDF library

netcdf.setDefaultFormat	Change default netCDF file format

File Operations

netcdf.abort	Revert recent netCDF file definitions
netcdf.close	Close netCDF file
netcdf.create	Create new NetCDF dataset
netcdf.endDef	End netCDF file define mode
netcdf.inq	Return information about netCDF file
netcdf.inqFormat	Determine format of NetCDF file
netcdf.inqGrps	Retrieve array of child group IDs
netcdf.inqUnlimDims	Return list of unlimited dimensions in group
netcdf.open	Open NetCDF data source
netcdf.reDef	Put open netCDF file into define mode
netcdf.setFill	Set netCDF fill mode
netcdf.sync	Synchronize netCDF file to disk

Dimensions

netcdf.defDim	Create netCDF dimension

netcdf.inqDim	Return netCDF dimension name and length
netcdf.inqDimID	Return dimension ID
netcdf.renameDim	Change name of netCDF dimension

Variables

netcdf.defVar	Create NetCDF variable
netcdf.defVarChunking	Define chunking behavior for NetCDF variable
netcdf.defVarDeflate	Define compression parameters for NetCDF variable
netcdf.defVarFill	Define fill parameters for NetCDF variable
netcdf.defVarFletcher32	Define checksum parameters for NetCDF variable
netcdf.getVar	Read data from NetCDF variable
netcdf.inqVarChunking	Determine chunking settings for NetCDF variable
netcdf.inqVarDeflate	Determine compression settings for NetCDF variable
netcdf.inqVarFill	Determine values of fill parameters for NetCDF variable

netcdf.inqVarFletcher32	Fletcher32 checksum setting for NetCDF variable
netcdf.inqVar	Information about variable
netcdf.inqVarID	Return ID associated with variable name
netcdf.putVar	Write data to netCDF variable
netcdf.renameVar	Change name of netCDF variable

Attributes

netcdf.copyAtt	Copy attribute to new location
netcdf.delAtt	Delete netCDF attribute
netcdf.getAtt	Return netCDF attribute
netcdf.inqAtt	Return information about netCDF attribute
netcdf.inqAttID	Return ID of netCDF attribute
netcdf.inqAttName	Return name of netCDF attribute
netcdf.putAtt	Write netCDF attribute
netcdf.renameAtt	Change name of attribute

Utilities

netcdf.getConstant	Return numeric value of named constant

netcdf.getConstantNames	Return list of constants known to netCDF library

netcdf.abort

Revert recent netCDF file definitions

Syntax

```
netcdf.abort(ncid)
```

Description

`netcdf.abort(ncid)` reverts a netCDF file to its previous state, backing out any definitions made since the file last entered define mode. A file enters define mode when you create it (using `netcdf.create`) or when you explicitly enter define mode (using `netcdf.redef`). Once you leave define mode (using `netcdf.endDef`), you cannot revert the definitions you made while in define mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. A call to `netcdf.abort` closes the file.

This function corresponds to the `nc_abort` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new file, performs an operation on the file, and then reverts the file back to its original state. To run this example, you must have write permission in your current directory.

```
% Create a netCDF file
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');

% Perform an operation, such as defining a dimension.
dimid = netcdf.defDim(ncid, 'lat', 50);

% Revert the file back to its previous state.
netcdf.abort(ncid)

% Verify that the file is now closed.
```

```
dimid = netcdf.defDim(ncid, 'lat', 50); % should fail
??? Error using ==> netcdflib
NetCDF: Not a valid ID

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);
```

See Also

netcdf.create | netcdf.endDef | netcdf.reDef

netcdf.close

Close netCDF file

Syntax

```
netcdf.close(ncid)
```

Description

`netcdf.close(ncid)` terminates access to the netCDF file identified by `ncid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_close` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new netCDF file, and then closes the file. You must have write permission in your current directory to run this example.

```
ncid = netcdf.open('foo.nc','NC_WRITE')

netcdf.close(ncid)
```

See Also

`netcdf.create` | `netCDF.open`

netcdf.copyAtt

Copy attribute to new location

Syntax

```
netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)
```

Description

`netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)` copies an attribute from one variable to another, possibly across files. `ncid_in` and `ncid_out` are netCDF file identifiers returned by `netcdf.create` or `netcdf.open`. `varid_in` identifies the variable with an attribute that you want to copy. `varid_out` identifies the variable to which you want to associate a copy of the attribute.

This function corresponds to the `nc_copy_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example makes a copy of the attribute associated with the first variable in the netCDF example file, `example.nc`, in a new file. To run this example, you must have write permission in your current directory.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier for a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Create new netCDF file.
ncid2 = netcdf.create('foo.nc','NC_NOCLOBBER');

% Define a dimension in the new file.
dimid2 = netcdf.defDim(ncid2,'x',50);
```

```
% Define a variable in the new file.
varid2 = netcdf.defVar(ncid2,'myvar','double',dimid2);

% Copy the attribute named 'description' into the new file,
% associating the attribute with the new variable.
netcdf.copyAtt(ncid,varid,'description',ncid2,varid2);
%
% Check the name of the attribute in new file.
attname = netcdf.inqAttName(ncid2,varid2,0)

attname =

description
```

See Also

netcdf.inqAtt | netcdf.inqAttID | netcdf.inqAttName | netcdf.putAtt | netcdf.renameAtt

netcdf.create

Create new NetCDF dataset

Syntax

```
ncid = netcdf.create(filename,cmode)
[chunksize_out,ncid] =
netcdf.create(filename,cmode,initsz,chunksize)
```

Description

`ncid = netcdf.create(filename,cmode)` creates a new NetCDF file according to the file creation mode. The return value `ncid` is a file ID. The `cmode` parameter is a text string that describes the type of file access, which can have any of the following values.

Value of `cmode`	Description
`NOCLOBBER`	Prevent overwriting of existing file with the same name.
`CLOBBER`	Overwrite any existing file with the same name.
`SHARE`	Allow synchronous file updates.
`64BIT_OFFSET`	Allow easier creation of files and variables which are larger than two gigabytes.
`NETCDF4`	Create a NetCDF-4/HDF5 file
`CLASSIC_MODEL`	Enforce the classic model; has no effect unless used in a bitwise-or with `NETCDF4`

Note: You can specify the mode as a numeric value, retrieved using the `netcdf.getConstant` function. To specify more than one mode, use a bitwise-OR of the numeric values of the modes.

`[chunksize_out,ncid] = netcdf.create(filename,cmode,initsz,chunksize)` creates a new NetCDF file, but with additional performance tuning parameters. `initsz` sets the initial size of the

file. `chunksize` can affect I/O performance. The actual value chosen by the NetCDF library might not correspond to the input value.

This function corresponds to the `nc_create` and `nc__create` functions in the NetCDF library C API. To use this function, you should be familiar with the NetCDF programming paradigm. See `netcdf` for more information.

Examples

Create NetCDF File Without Overwriting Existing File

Create a NetCDF dataset named `foo.nc`, only if no other file with the same name exists in the current directory. To run this example, you must have write permission in your current directory.

```
ncid = netcdf.create('foo.nc','NOCLOBBER')
```

```
ncid =

       65536
```

`netcdf.create` returns a file identifier.

Close the file

```
netcdf.close(ncid)
```

Create NetCDF-4 File Using Classic Model

Get the numeric values corresponding to the `NETCDF4` and `CLASSIC_MODEL` constants defined by the NetCDF library. Use a bitwise-OR of the numeric values to specify more than one creation mode.

```
cmode = netcdf.getConstant('NETCDF4');
cmode = bitor(cmode,netcdf.getConstant('CLASSIC_MODEL'));
```

Create a NetCDF-4 file that uses the classic model by specifying the creation mode value, `cmode`.

```
ncid = netcdf.create('myfile.nc',cmode);
```

Close the file.

```
netcdf.close(ncid);
```

See Also

`netcdf.getConstant` | `netcdf.open`

netcdf.defDim

Create netCDF dimension

Syntax

```
dimid = netcdf.defDim(ncid,dimname,dimlen)
```

Description

`dimid = netcdf.defDim(ncid,dimname,dimlen)` creates a new dimension in the netCDF file specified by `ncid`, where`dimname` is a character string that specifies the name of the dimension and `dimlen` is a numeric value that specifies its length. To define an unlimited dimension, specify the predefined constant `'NC_UNLIMITED'` for `dimlen`, using `netcdf.getConstant` to retrieve the value.

`netcdf.defDim` returns `dimid`, a numeric ID corresponding to the new dimension.

This function corresponds to the `nc_def_dim` function in the netCDF library C API.To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

Examples

Create a new file and define two dimensions in the file. One dimension is an unlimited dimension. To run this example, you must have write permission in your current folder.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_NOCLOBBER')

% Define a dimension.
lat_dimID = netcdf.defDim(ncid,'latitude',360);

% Define an unlimited dimension.
```

```
long_dimID = netcdf.defDim(ncid,'longitude',...
  netcdf.getConstant('NC_UNLIMITED'));
```

See Also

netcdf.getConstant

netcdf.defGrp

Create group in NetCDF file

Syntax

```
childGrpID = netcdf.defGrp(parentGroupId,childGroupName)
```

Description

`childGrpID = netcdf.defGrp(parentGroupId,childGroupName)` creates a child group with the name specified by `childGroupName`, that is the child of the parent group specified by `parentGroupId`

Input Arguments

`parentGroupId`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`childGroupName`

Text string specifying the name that you want to assign to the group.

Default:

Output Arguments

`childGrpID`

Identifier of a NetCDF group.

Examples

This example creates a NetCDF dataset and then defines a group.

```
ncid = netcdf.create('myfile.nc','netcdf4');
childGroupId = netcdf.defGrp(ncid,'mygroup');
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_grp` function in the NetCDF library C API.

For copyright information, read the files `netcdfcopyright.txt` and `mexnccopyright.txt`.

See Also

`netcdf` | `netcdf.inqGrps`

netcdf.defVar

Create NetCDF variable

Syntax

```
varid = netcdf.defVar(ncid,varname,xtype,dimids)
```

Description

`varid = netcdf.defVar(ncid,varname,xtype,dimids)` creates a new variable in the dataset identified by `ncid`.

`varname` is a character string that specifies the name of the variable.

`xtype` specifies the NetCDF data type of the variable, using one of the following strings.

Value of `xtype`	MATLAB Class
`'NC_BYTE'`	`int8` or `uint8`[a]
`'NC_CHAR'`	`char`
`'NC_SHORT'`	`int16`
`'NC_INT'`	`int32`
`'NC_FLOAT'`	`single`
`'NC_DOUBLE'`	`double`

a. NetCDF interprets byte data as either signed or unsigned.

Alternatively, `xtype` can be the numeric equivalent returned by the `netcdf.getConstant` function.

`dimids` specifies a list of dimension IDs.

`netcdf.defVar` returns `varid`, a numeric identifier for the new variable.

This function corresponds to the `nc_def_var` function in the NetCDF library C API. Because MATLAB uses FORTRAN-style ordering, the fastest-varying dimension comes

first and the slowest comes last. Any unlimited dimension is therefore last in the list of dimension IDs. This ordering is the reverse of that found in the C API.To use this function, you should be familiar with the NetCDF programming paradigm. See `netcdf` for more information.

Examples

Define Dimension and Variable in NetCDF File

Create a new NetCDF file, define a dimension in the file, and then define a variable on that dimension. In NetCDF files, you must create a dimension before you can create a variable. To run this example, you must have write permission in your current folder.

Create a new NetCDF file named `foo.nc`.

```
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');
```

Define a dimension in the new file.

```
dimid = netcdf.defDim(ncid,'x',50);
```

Define a variable in the new file using `netcdf.defVar`.

```
varid = netcdf.defVar(ncid,'myvar','double',dimid)
```

```
varid =

     0
```

`netcdf.defVar` returns a numeric identifier for the new variable.

Close the file.

```
netcdf.close(ncid)
```

See Also

`netCDF.getConstant` | `netCDF.inqVar` | `netCDF.putVar`

netcdf.defVarChunking

Define chunking behavior for NetCDF variable

Syntax

```
netcdf.defVarChunking(ncid,varid,storage,chunkDims)
```

Description

`netcdf.defVarChunking(ncid,varid,storage,chunkDims)`. sets the chunk settings for the variable specified by `varid`. Chunking is a technique to improve performance. `storage` specifies the type of chunking to use and `chunkDims` specifies the extents of the chunk size. You must specify the chunk size used with a variable after creating the variable but before you write data to the variable.

You cannot specify the chunk size for variables in a NetCDF file created with the netCDF-3 mode (`CLASSIC_MODEL`).

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`storage`

Text string specifying whether NetCDF should break the variable into chunks when writing to a file. If set to `CHUNKED`, NetCDF breaks the variable into chunks; if set to `CONTIGUOUS`, NetCDF does not break the data into chunks.

Default:

`chunkDims`

Array specifying the dimensions of the chunk.

Because MATLAB uses FORTRAN-style ordering, the order of dimensions in `chunkdims` is reversed relative to what would be in the C API.

If storage is `CONTIGUOUS`, you can omit `chunkDims`.

Default: Chunk size determined by the NetCDF library.

Examples

This example creates a NetCDF file and specifies the chunking behavior of a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarChunking(ncid,varid,'CHUNKED',[180 360]);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_chunking` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqVarChunking`

netcdf.defVarDeflate

Define compression parameters for NetCDF variable

Syntax

```
netcdf.defVarDeflate(ncid,varid,shuffle,deflate,deflateLevel)
```

Description

`netcdf.defVarDeflate(ncid,varid,shuffle,deflate,deflateLevel)` sets the compression parameters for the NetCDF variable specified by `varid` in the location specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`shuffle`

Boolean value. To turn on the shuffle filter, set this argument to `true`. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

Default:

`deflate`

Boolean value. To turn on compression, set this argument to `true` and set the `deflateLevel` argument to the desired compression level.

Default:

`deflateLevel`

Numeric value between `0` and `9` specifying the amount of compression, where `0` is no compression and `9` is the most compression.

Default:

Examples

This example create a variable with dimensions [1800 3600] and a compression level of 5. This results in a chunked layout that is a 10-by-10 grid.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarDeflate(ncid,varid,true,true,5);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_deflate` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqVarDeflate`

netcdf.defVarFill

Define fill parameters for NetCDF variable

Syntax

```
netcdf.defVarFill(ncid,varid,noFillMode,fillValue)
```

Description

`netcdf.defVarFill(ncid,varid,noFillMode,fillValue)` sets the fill parameters for the NetCDF variable identified by `varid`. `ncid` specifies the location.

For netCDF-4 files, you can only specify fill values when the NetCDF is in definition mode (before calling `netcdf.endDef`). For NetCDF files in classic and 64-bit offset modes, you can turn no-fill mode on and off at any time.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`noFillMode`

Boolean value. When set to `true`, turns off use of fill values for the variable, which can be helpful in high performance applications. When `true`, `netcdf.defVarFill` ignores the value of the `fillValue` argument. To use the fill value, set this to `false`.

Default:

`fillValue`

Specifies the value to use in the variable when no other value is specified. The data type must be the same data type as the variable.

Default:

Examples

This example creates a NetCDF file and defines a fill value for a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
dimid = netcdf.defDim(ncid,'latitude',180);
varid = netcdf.defVar(ncid,'latitude','double',dimid);
netcdf.defVarFill(ncid,varid,false,-999);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_fill` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.setFill` | `netcdf.inqVarFill`

netcdf.defVarFletcher32

Define checksum parameters for NetCDF variable

Syntax

```
netcdf.defVarFletcher32(ncid,varid,setting)
```

Description

`netcdf.defVarFletcher32(ncid,varid,setting)` defines the checksum settings for the NetCDF variable specified by `varid` in the file specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`setting`

Text string specifying whether Fletcher32 checksum error detection is used with the variable. To turn on Fletcher32 checksum, specify the value `FLETCHER32`. To turn off the use of checksum error detection, specify the value `NOCHECKSUM`.

Default:

Examples

This example creates a NetCDF dataset and turns on the Fletcher32 checksum for a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarFletcher32(ncid,varid,'FLETCHER32');
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_fletcher32` function in the NetCDF library C API.

For copyright information, read the files `netcdfcopyright.txt` and `mexnccopyright.txt`.

See Also

`netcdf` | `netcdf.inqVarFletcher32`

netcdf.delAtt

Delete netCDF attribute

Syntax

```
netcdf.delAtt(ncid,varid,attName)
```

Description

`netcdf.delAtt(ncid,varid,attName)` deletes the attribute identified by the text string `attName`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric value that identifies the variable. To delete a global attribute, use `netcdf.getConstant('GLOBAL')` for the `varid`. You must be in define mode to delete an attribute.

This function corresponds to the `nc_del_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Determine number of global attributes in file.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   1
```

```
% Get name of attribute; it is needed for deletion.
attname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

% Put file in define mode to delete an attribute.
netcdf.reDef(ncid);

% Delete the global attribute in the netCDF file.
netcdf.delAtt(ncid,netcdf.getConstant('GLOBAL'),attname);

% Verify that the global attribute was deleted.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   0
```

See Also

netcdf.getConstant | netcdf.inqAttName

netcdf.endDef

End netCDF file define mode

Syntax

```
netcdf.endDef(ncid)
netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)
```

Description

`netcdf.endDef(ncid)` takes a netCDF file out of define mode and into data mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)` takes a netCDF file out of define mode, specifying four additional performance tuning parameters. For example, one reason for using the performance parameters is to reserve extra space in the netCDF file header using the `h_minfree` parameter:

```
ncid = netcdf.endDef(ncid,20000,4,0,4);
```

This reserves 20,000 bytes in the header, which can be used later when adding attributes. This can be extremely efficient when working with very large netCDF 3 files. To understand how to use these performance tuning parameters, see the netCDF library documentation.

This function corresponds to the `nc_enddef` and `nc__enddef` functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

Take File out of Define Mode

When you create a file using `netcdf.create`, the functions opens the file in define mode. This example uses `netcdf.endDef` to take the file out of define mode.

Create a netCDF file.

```
ncid = netcdf.create('foo.c','NC_NOCLOBBER');
```

Define a dimension.

```
dimid = netcdf.defDim(ncid,'lat',50);
```

Leave define mode.

```
netcdf.endDef(ncid)
```

Test if still in define mode.

```
dimid = netcdf.defDim(ncid,'lon',50);
```

```
Error using netcdflib
The NetCDF library encountered an error during execution of 'defDim' function - 'Operat
(NC_ENOTINDEFINE)'.

Error in netcdf.defDim (line 25)
dimid = netcdflib('defDim', ncid,dimname,dimlen);
```

`netcdf.defDim` errors, as expected.

See Also

`netcdf.create` | `netcdf.reDef`

netcdf.getAtt

Return netCDF attribute

Syntax

```
attrvalue = netcdf.getAtt(ncid,varid,attname)
attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)
```

Description

`attrvalue = netcdf.getAtt(ncid,varid,attname)` returns `attrvalue`, the value of the attribute specified by the text string `attname`. When it chooses the data type of `attrvalue`, MATLAB attempts to match the netCDF class of the attribute. For example, if the attribute has the netCDF data type `NC_INT`, MATLAB uses the `int32` class for the output data. If an attribute has the netCDF data type `NC_BYTE`, the class of the output data is `int8` value.

`attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)` returns `attrvalue`, the value of the attribute specified by the text string `attname`, using the output class specified by `output_datatype`. You can specify any of the following strings for the output data type.

`'int'`	`'double'`	`'int16'`
`'short'`	`'single'`	`'int8'`
`'float'`	`'int32'`	`'uint8'`

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets the value of the attribute associated with the first variable. The example also gets the value of the global variable in the file.

```
% Open a netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name of first variable.
[varname vartype vardimIDs varatts] = netcdf.inqVar(ncid,0);

% Get ID of variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get attribute name, given variable id.
attname = netcdf.inqAttName(ncid,varid,0);

% Get value of attribute.
attval = netcdf.getAtt(ncid,varid,attname);

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);

% Get value of global attribute.
gattval = netcdf.getAtt(ncid,netcdf.getConstant('NC_GLOBAL'),gattname)

gattval =

09-Jun-2008
```

See Also

netcdf.inqAtt | netcdf.putAtt

netcdf.getChunkCache

Retrieve chunk cache settings for NetCDF library

Syntax

```
[csize, nelems, premp] = netcdf.getChunkCache()
```

Description

`[csize, nelems, premp] = netcdf.getChunkCache()` returns the default chunk cache settings.

Output Arguments

`csize`

Scalar double specifying the total size of the raw data chunk cache in bytes.

`nelems`

Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.

`premp`

Double, between 0 and 1, inclusive, that specifies how the library handles preempting fully read chunks in the chunk cache. A value of zero means fully read chunks are treated no differently than other chunks, that is, preemption occurs solely based on the Least Recently Used (LRU) algorithm. A value of 1 means fully read chunks are always preempted before other chunks.

Examples

Determine information about the chunk cache size used by the NetCDF library.

```
[csize, nelems, premp] = netcdf.getChunkCache();
```

References

This function corresponds to the `nc_get_chunk_cache` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.setChunkCache`

netcdf.getConstant

Return numeric value of named constant

Syntax

```
val = netcdf.getConstant(param_name)
```

Description

`val = netcdf.getConstant(param_name)` returns the numeric value corresponding to the name of a constant defined by the netCDF library. For example, `netcdf.getConstant('NC_NOCLOBBER')` returns the numeric value corresponding to the netCDF constant `NC_NOCLOBBER`.

The value for `param_name` can be either upper- or lowercase, and does not need to include the leading three characters `'NC_'`. To retrieve a list of all the names defined by the netCDF library, use the `netcdf.getConstantNames` function.

This function has no direct equivalent in the netCDF C interface. To find out more about NetCDF, see `netcdf`.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Determine contents of the file.
[ndims nvars natts dimm] = netcdf.inq(ncid);

% Get name of global attribute.
% Note: You must use netcdf.getConstant to specify NC_GLOBAL.
attname = netcdf.inqattname(ncid,netcdf.getConstant('NC_GLOBAL'),0)

attname =
```

```
creation_date
```

See Also

`netcdf.getConstantNames`

netcdf.getConstantNames

Return list of constants known to netCDF library

Syntax

```
val = netcdf.getConstantNames(param_name)
```

Description

`val = netcdf.getConstantNames(param_name)` returns a list of names of netCDF library constants, definitions, and enumerations. When these strings are supplied as actual parameters to MATLAB netCDF package functions, the functions automatically convert the constant to the appropriate numeric value.

This MATLAB function has no direct equivalent in the netCDF C interface. To find out more about netCDF, see `netcdf`.

Examples

```
nc_constants = netcdf.getConstantNames

nc_constants =

    'NC2_ERR'
    'NC_64BIT_OFFSET'
    'NC_BYTE'
    'NC_CHAR'
    'NC_CLOBBER'
    'NC_DOUBLE'
    'NC_EBADDIM'
    'NC_EBADID'
    'NC_EBADNAME'
    'NC_EBADTYPE'
  ...
```

See Also

`netCDF.getConstantNames`

netcdf.getVar

Read data from NetCDF variable

Syntax

```
data = netcdf.getVar(ncid,varid)
data = netcdf.getVar(ncid,varid,start)
data = netcdf.getVar(ncid,varid,start,count)
data = netcdf.getVar(ncid,varid,start,count,stride)
data = netcdf.getVar( ___ ,output_type)
```

Description

`data = netcdf.getVar(ncid,varid)` returns `data`, the value of the variable specified by `varid`. MATLAB attempts to match the class of the output data to NetCDF class of the variable.

`ncid` is a NetCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`data = netcdf.getVar(ncid,varid,start)` returns a single value starting at the specified index, `start`.

`data = netcdf.getVar(ncid,varid,start,count)` returns a contiguous section of a variable. `start` specifies the starting point and `count` specifies the amount of data to return.

`data = netcdf.getVar(ncid,varid,start,count,stride)` returns a subset of a section of a variable. `start` specifies the starting point, `count` specifies the extent of the section, and `stride` specifies which values to return.

`data = netcdf.getVar( ___ ,output_type)` specifies the data type of the return value `data`. You can specify any of the following strings for the output data type.

`'int8'`
`'uint8'`
`'int16'`

`'int32'`
`'single'`
`'double'`

This function corresponds to several functions in the NetCDF library C API. To use this function, you should be familiar with the NetCDF programming paradigm. See `netcdf` for more information.

Examples

Read Value of Variable in NetCDF File

Open the example file, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');
```

Get the name of the first variable in the file.

```
varname = netcdf.inqVar(ncid,0)
```

```
varname =

avagadros_number
```

Get variable ID of the first variable, given its name.

```
varid = netcdf.inqVarID(ncid,varname)
```

```
varid =

     0
```

Get the value of the variable. Use the variable ID as the second input to the `netcdf.getVar` function.

```
data = netcdf.getVar(ncid,varid)
```

```
data =

   6.0221e+23
```

Display the data type of the output value.

```
whos data
```

```
  Name      Size            Bytes  Class     Attributes

  data      1x1                 8  double
```

Get the value of the `avogadros_number` variable again, specifying that the output data type should be `single`.

```
data = netcdf.getVar(ncid,varid,'single');
```

Display the data type of the output value.

```
whos data
```

```
  Name      Size            Bytes  Class     Attributes

  data      1x1                 4  single
```

Close the NetCDF file.

```
netcdf.close(ncid)
```

See Also

`netcdf.create` | `netcdf.inqVarID` | `netcdf.open`

netcdf.inq

Return information about netCDF file

Syntax

```
[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)
```

Description

`[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)` returns the number of dimensions, variables, and global attributes in a netCDF file. The function also returns the ID of the dimension defined with unlimited length, if one exists.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. You can call `netcdf.inq` in either define mode or data mode.

This function corresponds to the `nc_inq` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses the `netcdf.inq` function to get information about the contents of the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE')

% Get information about the contents of the file.
[numdims, numvars, numglobalatts, unlimdimID] = netcdf.inq(ncid)

numdims =

     4
```

```
numvars =

     4


numglobalatts =

     1


unlimdimID =

     3
```

See Also

netcdf.create | netcdf.open

netcdf.inqDimIDs

Retrieve list of dimension identifiers in group

Syntax

```
dimIDs = netcdf.inqDimIDs(ncid)
dimIDs = netcdf.inqDimIDs(ncid,includeParents)
```

Description

`dimIDs = netcdf.inqDimIDs(ncid)` returns a list of dimension identifiers in the group specified by `ncid`.

`dimIDs = netcdf.inqDimIDs(ncid,includeParents)` includes all dimensions in all parent groups if `includeParents` is `true`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`includeParents`

Boolean value. If set to `true`, `netcdf.inqDimIDs` includes the dimensions of all parent groups.

Default: `false`

Output Arguments

`dimIDs`

Array of dimension IDs

Examples

This example opens the NetCDF sample file and gets the IDs of all the dimensions.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid1');
dimids = netcdf.inqDimIDs(gid);
dimids_all = netcdf.inqDimIDS(gid, true);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_dimids` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqVarIDs`

netcdf.inqFormat

Determine format of NetCDF file

Syntax

```
format = netcdf.inqFormat(ncid)
```

Description

`format = netcdf.inqFormat(ncid)` returns the format for the file specified by NetCDF file identifier, `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`format`

Text string that specifies the format of the NetCDF file. Values include:

Format String	Description
`FORMAT_CLASSIC`	Classic format — Original NetCDF format, used by all NetCDF files created between 1989 and 2004
`FORMAT_64BIT`	Classic format, 64–bit — Original format with 64–bit addressing capability to allow creation and access of much larger files.

Format String	Description
FORMAT_NETCDF4	Enhanced model, HDF5-based — Introduced in 2008, NetCDF, version 4, extends the classic model and is based on HDF5.
FORMAT_NETCDF4_CLASSIC	Classic model, HDF5-based — Introduced in 2008, NetCDF, version 4, implements classic model but is based on HDF5.

Examples

This example opens the sample NetCDF file and determines the format.

```
ncid = netcdf.open('example.nc','NOWRITE');
fmt = netcdf.inqFormat(ncid)

format =

FORMAT_NETCDF4

netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_format` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.getConstant` | `netcdf`

netcdf.inqGrpName

Retrieve name of group

Syntax

```
groupName = netcdf.inqGrpName(ncid)
```

Description

`groupName = netcdf.inqGrpName(ncid)` returns the name of a group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`groupName`

Text string specifying name of a group. The root group has the name `'/'`.

Examples

This example opens the NetCDF sample file and gets the names of groups in the dataset.

```
ncid = netcdf.open('example.nc','nowrite');
name = netcdf.inqGrpName(ncid);
```

```
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grpname` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqGrpNameFull`

netcdf.inqGrpNameFull

Complete pathname of group

Syntax

```
groupName = netcdf.inqGrpNameFull(ncid)
```

Description

`groupName = netcdf.inqGrpNameFull(ncid)` returns the complete pathname of the group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`groupName`

Text string specifying complete path of group.

The root group has the name `'/'`. The names of parent groups and child groups use the forward slash `'/'` separator, as in UNIX folder names, for example, `/group1/subgrp2/subsubgrp3`.

Examples

Open the NetCDF sample dataset and retrieve the names of all groups.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid2');
fullName = netcdf.inqGrpNameFull(gid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grpname_full` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqGrpName`

netcdf.inqGrpParent

Retrieve ID of parent group.

Syntax

```
parentGroupID = netcdf.inqGrpParent(ncid)
```

Description

`parentGroupID = netcdf.inqGrpParent(ncid)` returns the ID of the parent group given the location of the child group, specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`parentGroupID`

Identifier of the NetCDF group or file that is the parent of the specified file or group.

Examples

This example opens the NetCDF sample file and gets the full path of the parent of the specified group.

```
ncid = netcdf.open('example.nc','NOWRITE');
```

```
gid = netcdf.inqNcid(ncid,'grid2');
parentId = netcdf.inqGrpParent(gid);
fullName = netcdf.inqGrpNameFull(parentId);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grp_parent` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqGrps`

netcdf.inqGrps

Retrieve array of child group IDs

Syntax

```
childGrps = netcdf.inqGrps(ncid)
```

Description

`childGrps = netcdf.inqGrps(ncid)` returns all the child group IDs in the parent group, specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`childGrps`

Array containing identifiers of child groups in the specified NetCDF file or group.

Examples

This example opens the sample NetCDF file and then gets information about the groups it contains.

```
ncid = netcdf.open('example.nc','nowrite');
```

```
childGroups = netcdf.inqGrps(ncid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grps` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqNcid`

netcdf.inqNcid

Return ID of named group

Syntax

```
childGroupId = netcdf.inqNcid(ncid,childGroupName)
```

Description

`childGroupId = netcdf.inqNcid(ncid,childGroupName)` returns the ID of the child group, specified by the name `childGroupName`, in the file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`childGroupName`

Text string specifying the name of a NetCDF group.

Default:

Output Arguments

`childGroupID`

Identifier of a NetCDF group.

Examples

This example opens the sample NetCDF dataset and then gets the ID of a group in the dataset.

```
ncid = netcdf.open('example.nc','nowrite');
gid = netcdf.inqNcid(ncid,'grid1');
netcdf.close(ncid);
```

References

This function corresponds to the nc_inq_ncid function in the netCDF library C API. Read the files `netcdfcopyright.txt` and `mexnccopyright.txt` for more information.

See Also

`netcdf` | `netcdf.inqGrpName` | `netcdf.inqGrpNameFull`

netcdf.inqUnlimDims

Return list of unlimited dimensions in group

Syntax

```
unlimdimIDs = netcdf.inqUnlimDims(ncid)
```

Description

`unlimdimIDs = netcdf.inqUnlimDims(ncid)` returns the IDs of all unlimited dimensions in the group specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`unlimDimIDs`

An array containing the identifiers of each unlimited dimension. `unlimDimIDs` is empty if there are no unlimited dimensions.

Examples

This example opens the NetCDF sample dataset and gets the IDs of all the unlimited dimensions.

```
ncid = netcdf.open('example.nc','NOWRITE');
dimids = netcdf.inqUnlimDims(ncid)

dimids =

     []

netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_unlim_dims` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defDim` | `netcdf.inqDim` | `netcdf.inqDimID` | `netcdf.renameDim` | `netcdf.inqDimIDs`

netcdf.inqVarIDs

IDs of all variables in group

Syntax

```
varids = netcdf.inqVarIDs(ncid)
```

Description

`varids = netcdf.inqVarIDs(ncid)` returns IDs of the all the variables in the group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`varids`

Array containing identifiers of variables in a NetCDF file or group.

Examples

This example opens the NetCDF sample file and gets the IDs of all the variables in a group.

```
ncid = netcdf.open('example.nc','NOWRITE');
```

```
gid = netcdf.inqNcid(ncid,'grid1');
varids = netcdf.inqVarIDs(gid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_varids` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqDimIDs` | `netcdf.inqVarID`

netcdf.inqVarChunking

Determine chunking settings for NetCDF variable

Syntax

```
[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)
```

Description

`[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)` returns the type of chunking and the dimensions of a chunk for the NetCDF variable specified by `varid`, in the file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable, returned by `netcdf.defVar`.

Default:

Output Arguments

`storage`

Text string specifying if NetCDF breaks the data into chunks when writing to a file. `CHUNKED` indicates the data is chunked; `CONTIGUOUS` indicates that the data is not chunked.

`chunkSizes`

Array specifying the dimensions of the chunk.

Because MATLAB uses FORTRAN-style ordering, the order of dimensions in `chunkdims` is reversed relative to what would be in the NetCDF C API.

If the storage type specified is `CONTIGUOUS`, `netcdf.inqVarChunking` returns an empty array, `[]`.

Examples

This example opens the NetCDF sample dataset and gets the values of chunking parameters associated with a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
groupid = netcdf.inqNcid(ncid,'grid1');
varid = netcdf.inqVarID(groupid,'temp');
[storage,chunkSize] = netcdf.inqVarChunking(groupid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_chunking` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defVar` | `netcdf.defVarChunking`

netcdf.inqVarDeflate

Determine compression settings for NetCDF variable

Syntax

```
[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid,varid)
```

Description

`[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid,varid)` returns the compression parameters for the NetCDF variable specified by `varid` in the location specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable, returned by `netcdf.defVar`.

Default:

Output Arguments

`shuffle`

Boolean value. `true` indicates that the shuffle filter is enabled for the specified variable. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

`deflate`

Boolean value. `true` indicates that compression is enabled for this variable. The `deflateLevel` argument specifies the level of compression.

`deflateLevel`

Scalar value between `0` and `9` specifying the amount of compression, where `0` is no compression and `9` is the most compression

Examples

This example opens the NetCDF sample file and gets information about variable compression.

```
ncid = netcdf.open('example.nc','NOWRITE');
groupid = netcdf.inqNcid(ncid,'grid1');
varid = netcdf.inqVarID(groupid,'temp');
[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(groupid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_deflate` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defVarDeflate`

netcdf.inqVarFill

Determine values of fill parameters for NetCDF variable

Syntax

```
[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)
```

Description

`[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)` returns the fill mode and the fill value for the variable `varid` in the file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable.

Default:

Output Arguments

`noFillMode`

Boolean value. `true` indicates that use of the fill values for the variable has been disabled.

`fillValue`

Specifies the value to use in the variable when no other value is specified and use of fill values has been enabled.

Examples

This example opens the NetCDF sample dataset and gets the fill mode and fill value used with a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
varid = netcdf.inqVarID(ncid,'temperature');
[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_fill` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defVarFill` | `netcdf.setFill`

netcdf.inqVarFletcher32

Fletcher32 checksum setting for NetCDF variable

Syntax

```
setting = netcdf.inqVarFletcher32(ncid,varid)
```

Description

`setting = netcdf.inqVarFletcher32(ncid,varid)` returns the Fletcher32 checksum setting for the NetCDF variable specified by `varid` in the file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier for NetCDF file, returned by `netcdf.create` or `netcdf.open`, or group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable.

Default:

Output Arguments

`setting`

Text string specifying whether the Fletcher32 checksum is turned on for the specified variable. `netcdf.inqVarFletcher32` returns the text string `FLETCHER32` if the checksum is turned on for the variable; otherwise, `NOCHECKSUM`.

Examples

This example opens the sample NetCDF file and gets information about the checksum setting for a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
varid = netcdf.inqVarID(ncid,'temperature');
setting = netcdf.inqVarFletcher32(ncid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_fletcher32` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` for more information.

See Also

`netcdf` | `netcdf.defVarFletcher32`

netcdf.inqAtt

Return information about netCDF attribute

Syntax

```
[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)
```

Description

`[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)` returns the data type, `xtype`, and length, `attlen`, of the attribute identified by the text string `attname`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable that the attribute is associated with. To get information about a global attribute, specify `netcdf.getConstant('NC_GLOBAL')` in place of `varid`.

This function corresponds to the `nc_inq_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about an attribute in the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NOWRITE');

% Get identifier of a variable in the file, given its name.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Get attribute name, given variable id and attribute number.
attname = netcdf.inqAttName(ncid,varid,0);
```

```
% Get information about the attribute.
[xtype,attlen] = netcdf.inqAtt(ncid,varid,'description')

xtype =

     2


attlen =

    31

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);

% Get information about global attribute.
[gxtype gattlen] = netcdf.inqAtt(ncid,netcdf.getConstant('NC_GLOBAL'),gattname)

gxtype =

     2


gattlen =

    11
```

See Also

netcdf.inqAttID | netcdf.inqAttName

netcdf.inqAttID

Return ID of netCDF attribute

Syntax

```
attnum = netcdf.inqAttID(ncid,varid,attname)
```

Description

`attnum = netcdf.inqAttID(ncid,varid,attname)` retrieves `attnum`, the identifier of the attribute specified by the text string `attname`.

`varid` specifies the variable the attribute is associated with.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_attid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the netCDF example file included with MATLAB, `example.nc`.

```
% Open the netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Retrieve the identifier of the attribute associated with the variable.
attid = netcdf.inqAttID(ncid,varid,'description');
```

See Also

`netcdf.inqAtt` | `netcdf.inqAttName`

netcdf.inqAttName

Return name of netCDF attribute

Syntax

```
attname = netcdf.inqAttName(ncid,varid,attnum)
```

Description

`attname = netcdf.inqAttName(ncid,varid,attnum)` returns `attname`, a text string specifying the name of an attribute.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric identifier of a variable in the file. If you want to get the name of a global attribute in the file, use `netcdf.getConstant('NC_GLOBAL')` in place of `attnum` is a zero-based numeric value specifying the attribute, with 0 indicating the first attribute, 1 the second attribute, and so on.

This function corresponds to the `nc_inq_attname` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number')

% Get the name of the attribute associated with the variable.
attname = netcdf.inqAttName(ncid,varid,0)
```

```
attname =

description

% Get the name of the global attribute associated with the variable.
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

gattname =

creation_date
```

See Also

netcdf.inqAtt | netcdf.inqAttID

netcdf.inqDim

Return netCDF dimension name and length

Syntax

```
[dimname, dimlen] = netcdf.inqDim(ncid,dimid)
```

Description

`[dimname, dimlen] = netcdf.inqDim(ncid,dimid)` returns the name, `dimname`, and length, `dimlen`, of the dimension specified by `dimid`. If `ndims` is the number of dimensions defined for a netCDF file, each dimension has an ID between 0 and `ndims-1`. For example, the dimension identifier of the first dimension is 0, the second dimension is 1, and so on.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dim` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

The example opens the example netCDF file include with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0)

dimname =

x


dimlen =
```

```
    50
```

See Also

`netcdf.inqDimID`

netcdf.inqDimID

Return dimension ID

Syntax

```
dimid = netcdf.inqDimID(ncid,dimname)
```

Description

`dimid = netcdf.inqDimID(ncid,dimname)` returns `dimid`, the identifier of the dimension specified by the character string `dimname`. You can use the `netcdf.inqDim` function to retrieve the dimension name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dimid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0);

% Retrieve identifier of dimension.
dimid = netcdf.inqDimID(ncid,dimname)

dimid =

     0
```

See Also

`netcdf.inqDim`

netcdf.inqLibVers

Return NetCDF library version information

Syntax

```
libvers = netcdf.inqLibVers
```

Description

`libvers = netcdf.inqLibVers` returns a string identifying the version of the NetCDF library.

This function corresponds to the `nc_inq_libvers` function in the NetCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

```
libvers = netcdf.inqLibVers

libvers =

4.1.3
```

netcdf.inqVar

Information about variable

Syntax

```
[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)
```

Description

`[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)` returns information about the variable identified by `varid`. The argument, `ncid`, is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

The output argument, `varname`, is the name of the variable. `xtype` is the data type, `dimids` is the dimension IDs, and `natts` is the number of attributes associated with the variable. Dimension IDs are zero-based.

This function corresponds to the `nc_inq_var` function in the netCDF library C API. Because MATLAB uses FORTRAN-style ordering, however, the order of the dimension IDs is reversed relative to what would be obtained from the C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

Open the example netCDF file included with MATLAB, `example.nc`, and get information about a variable in the file.

```
% Open the example netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about third variable in the file.
[varname, xtype, dimids, numatts] = netcdf.inqVar(ncid,2)

varname =
```

```
peaks

xtype =

     5

dimids =

     0     1

numatts =

     1     1
```

See Also

netcdf.create | netcdf.inqVarID | netcdf.open

netcdf.inqVarID

Return ID associated with variable name

Syntax

```
varid = netcdf.inqVarID(ncid,varname)
```

Description

`varid = netcdf.inqVarID(ncid,varname)` returns `varid`, the ID of a netCDF variable specified by the text string, `varname`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_varid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses several inquiry functions to get the ID of the first variable.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about first variable in the file.
[varname, xtype, dimids, atts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name
varid = netcdf.inqVarID(ncid,varname)

varid =

     0
```

See Also

`netcdf.create` | `netcdf.inqVar` | `netcdf.open`

netcdf.open

Open NetCDF data source

Syntax

```
ncid = netcdf.open(source)
ncid = netcdf.open(source,mode)
[chosen_chunksize, ncid] = netcdf.open(source,mode,chunksize)
```

Description

`ncid = netcdf.open(source)` opens `source`, which can be the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source, for read-only access. Returns a NetCDF ID in `ncid`.

`ncid = netcdf.open(source,mode)` opens `source` with the type of access specified by `mode`, which can have any of the following values.

Value	Description
`'WRITE'`	Read-write access
`'SHARE'`	Synchronous file updates
`'NOWRITE'`	Read-only access (Default)

You can also specify `mode` as a numeric value that can be retrieved using `netcdf.getConstant`. Use these numeric values when you want to specify a bitwise-OR of several modes.

`[chosen_chunksize, ncid] = netcdf.open(source,mode,chunksize)` opens `source`, an existing netCDF data source, specifying the additional I/O performance tuning parameter, `chunksize`. The actual value used by the NetCDF library might not correspond to the input value you specify.

This function corresponds to the `nc_open` and `nc__open` functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example NetCDF file included with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NOWRITE');
netcdf.close(ncid);
```

See Also

`netcdf.close` | `netcdf` | `netcdf.getConstant`

netcdf.putAtt

Write netCDF attribute

Syntax

```
netcdf.putAtt(ncid,varid,attrname,attrvalue)
```

Description

`netcdf.putAtt(ncid,varid,attrname,attrvalue)` writes the attribute named `attrname` with value `attrvalue` to the netCDF variable specified by `varid`. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`.

`ncid` is a netCDF file identifier returned by `netCDF.create` or `netCDF.open`.

Note: You cannot use `netcdf.putAtt` to set the `'_FillValue'` attribute of NetCDF4 files. Use the `netcdf.defVarFill` function to set the fill value for a variable.

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new netCDF file, defines a dimension and a variable, adds data to the variable, and then creates an attribute associated with the variable. To run this example, you must have writer permission in your current directory.

```
% Create a variable in the workspace.
my_vardata = linspace(0,50,50);

% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');
```

```
% Define a dimension in the file.
dimid = netcdf.defDim(ncid,'my_dim',50);

% Define a new variable in the file.
varid = netcdf.defVar(ncid,'my_var','double',dimid);

% Leave define mode and enter data mode to write data.
netcdf.endDef(ncid);

% Write data to variable.
netcdf.putVar(ncid,varid,my_vardata);

% Re-enter define mode.
netcdf.reDef(ncid);

% Create an attribute associated with the variable.
netcdf.putAtt(ncid,0,'my_att',10);

% Verify that the attribute was created.
[xtype xlen] = netcdf.inqAtt(ncid,0,'my_att')

xtype =

     6


xlen =

     1
```

This example creates a new netCDF file, specifies a global attribute, and assigns a value to the attribute.

```
ncid = netcdf.create('myfile.nc','CLOBBER');
varid = netcdf.getConstant('GLOBAL');
netcdf.putAtt(ncid,varid,'creation_date',datestr(now));
netcdf.close(ncid);
```

See Also

`netcdf.getAtt` | `netcdf.defVarFill` | `netcdf.getConstant`

netcdf.putVar

Write data to netCDF variable

Syntax

```
netcdf.putVar(ncid,varid,data)
netcdf.putVar(ncid,varid,start,data)
netcdf.putVar(ncid,varid,start,count,data)
netcdf.putVar(ncid,varid,start,count,stride,data)
```

Description

`netcdf.putVar(ncid,varid,data)` writes `data` to a netCDF variable identified by `varid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.putVar(ncid,varid,start,data)` writes a single data value into the variable at the index specified by `start`.

`netcdf.putVar(ncid,varid,start,count,data)` writes a section of values into the netCDF variable at the index specified by the vector `start` to the extent specified by the vector `count`, along each dimension of the specified variable.

`netcdf.putVar(ncid,varid,start,count,stride,data)` writes the subsection specified by sampling interval, `stride`, of the values in the section of the variable beginning at the index `start` and to the extent specified by `count`.

This function corresponds to several variable I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

Write Variable to New netCDF File

Create a new netCDF file and write a variable to the file.

Create a 50 element vector for a variable.

```
my_vardata = linspace(0,50,50);
```

Open the netCDF file.

```
ncid = netcdf.create('foo.nc','NOCLOBBER');
```

Define the dimensions of the variable.

```
dimid = netcdf.defDim(ncid,'my_dim',50);
```

Define a new variable in the file.

```
my_varID = netcdf.defVar(ncid,'my_var','double',dimid);
```

Leave define mode and enter data mode to write data.

```
netcdf.endDef(ncid);
```

Write data to variable.

```
netcdf.putVar(ncid,my_varID,my_vardata);
```

Verify that the variable was created.

```
[varname xtype dimid natts ] = netcdf.inqVar(ncid,0)
```

```
varname =

my_var


xtype =

     6


dimid =

     0


natts =

     0
```

Close the file.

```
netcdf.close(ncid)
```

Write Elements of Variable

Write to the first ten elements of the example `temperature` variable.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.nc');
copyfile(srcFile,'myfile.nc');
fileattrib('myfile.nc','+w');
ncid = netcdf.open('myfile.nc','WRITE');
varid = netcdf.inqVarID(ncid,'temperature');
data = [100:109];
netcdf.putVar(ncid,varid,0,10,data);
netcdf.close(ncid);
```

See Also

`netcdf.getVar`

netcdf.reDef

Put open netCDF file into define mode

Syntax

```
netcdf.reDef(ncid)
```

Description

`netcdf.reDef(ncid)` puts an open netCDF file into define mode so that dimensions, variables, and attributes can be added or renamed. Attributes can also be deleted in define mode. `ncid` is a valid NetCDF file ID, returned from a previous call to `netcdf.open` or `netcdf.create`.

This function corresponds to the `nc_redef` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Try to define a dimension.
dimid = netcdf.defdim(ncid, 'lat', 50); % should fail.
??? Error using ==> netcdflib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);

% Put file in define mode.
netcdf.reDef(ncid);
```

```
% Try to define a dimension again. Should succeed.
dimid = netcdf.defDim(ncid, 'lat', 50);
```

See Also

netcdf.create | netcdf.endDef | netcdf.open

netcdf.renameAtt

Change name of attribute

Syntax

```
netcdf.renameAtt(ncid,varid,oldName,newName)
```

Description

`netcdf.renameAtt(ncid,varid,oldName,newName)` changes the name of the attribute specified by the character string `oldName`.

`newName` is a character string that specifies the new name.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable to which the attribute is associated. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`.

This function corresponds to the `nc_rename_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Get the ID of a variable the attribute is associated with.
varID = netcdf.inqVarID(ncid,'avagadros_number')

% Rename the attribute.
netcdf.renameAtt(ncid,varID,'description','Description');
```

```
% Verify that the name changed.
attname = netcdf.inqAttName(ncid,varID,0)

attname =

Description
```

See Also

netcdf.inqAttName

netcdf.renameDim

Change name of netCDF dimension

Syntax

```
netcdf.renameDim(ncid,dimid,newName)
```

Description

`netcdf.renameDim(ncid,dimid,newName)` renames the dimension identified by the dimension identifier, `dimid`.

`newName` is a character string specifying the new name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`

This function corresponds to the `nc_rename_dim` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This examples modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file is define mode.
netcdf.reDef(ncid)

% Get the identifer of a dimension to rename.
dimid = netcdf.inqDimID(ncid,'x');

% Rename the dimension.
netcdf.renameDim(ncid,dimid,'Xdim')
```

```
% Verify that the name changed.
data = netcdf.inqDim(ncid,dimid)

data =

Xdim
```

See Also

netcdf.defDim

netcdf.renameVar

Change name of netCDF variable

Syntax

```
netcdf.renameVar(ncid,varid,newName)
```

Description

`netcdf.renameVar(ncid,varid,newName)` renames the variable identified by `varid` in the netCDF file identified by `ncid`.`newName` is a character string specifying the new name.

This function corresponds to the `nc_rename_var` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file in define mode.
netcdf.redef(ncid)

% Get name of first variable
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname

varname =

avagadros_number
```

```
% Rename the variable, using a capital letter to start the name.
netcdf.renameVar(ncid,0,'Avagadros_number')

% Verify that the name of the variable changed.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname

varname =

Avagadros_number
```

See Also

netCDF.defVar | netCDF.inqVar | netCDF.putVar

netcdf.setChunkCache

Set default chunk cache settings for NetCDF library

Syntax

```
netcdf.setChunkCache(csize,nelems,premp)
```

Description

`netcdf.setChunkCache(csize,nelems,premp)` sets the default chunk cache settings used by the NetCDF library.

Settings apply for subsequent file open or create operations, for the remainder of the MATLAB session or until you issue a `clear mex` call. This function does not change the chunk cache settings of files already open.

Input Arguments

`csize`

Scalar double specifying the total size of the raw data chunk cache in bytes.

Default:

`nelems`

Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.

Default:

`premp`

Scalar double, between `0` and `1`, inclusive, that specifies how the library handles preempting fully read chunks in the chunk cache. A value of `0` means fully read chunks are treated no differently than other chunks, that is, preemption occurs solely based

on the Least Recently Used (LRU) algorithm. A value of 1 means fully read chunks are always preempted before other chunks.

Default:

Examples

This example sets the cache chunk size used by the NetCDF library.

```
netcdf.setChunkCache(32000000, 2003, .75)
```

References

This function corresponds to the `nc_set_chunk_cache` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.getChunkCache`

netcdf.setDefaultFormat

Change default netCDF file format

Syntax

```
oldFormat = netcdf.setDefaultFormat(newFormat)
```

Description

`oldFormat = netcdf.setDefaultFormat(newFormat)` changes the default format used by `netCDF.create` when creating new netCDF files, and returns the value of the old format. You can use this function to change the format used by a netCDF file without having to change the creation mode flag used in each call to `netCDF.create`.

`newFormat` can be either of the following values.

Value	Description
`'NC_FORMAT_CLASSIC'`	Original netCDF file format
`'NC_FORMAT_64BIT'`	64-bit offset format; relaxes limitations on creating very large files

You can also specify the numeric equivalent of these values, as retrieved by `netcdf.getConstant`.

This function corresponds to the `nc_set_default_format` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

```
oldFormat = netcdf.setDefaultFormat('NC_FORMAT_64BIT');
```

See Also

`netcdf.create`

netcdf.setFill

Set netCDF fill mode

Syntax

```
old_mode = netcdf.setFill(ncid,new_mode)
```

Description

`old_mode = netcdf.setFill(ncid,new_mode)` sets the fill mode for a netCDF file identified by `ncid`.

`new_mode` can be either `'FILL'` or `'NOFILL'` or their numeric equivalents, as retrieved by `netcdf.getConstant`. The default mode is `'FILL'`. netCDF pre-fills data with fill values. Specifying `'NOFILL'` can be used to enhance performance, because it avoids the duplicate writes that occur when the netCDF writes fill values that are later overwritten with data.

This function corresponds to the `nc_set_fill` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new file and specifies the fill mode used by netCDF with the file.

```
ncid = netcdf.open('foo.nc','NC_WRITE');

% Set filling behavior
old_mode = netcdf.setFill(ncid,'NC_NOFILL');
```

See Also

`netcdf.getConstant`

netcdf.sync

Synchronize netCDF file to disk

Syntax

```
netcdf.sync(ncid)
```

Description

`netcdf.sync(ncid)` synchronizes the state of a netCDF file to disk. The netCDF library normally buffers accesses to the underlying netCDF file, unless you specify the `NC_SHARE` mode when you opened the file with `netcdf.open` or `netcdf.create`. To call `netcdf.sync`, the netCDF file must be in data mode.

This function corresponds to the `nc_sync` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new netCDF file for write access, performs an operation on the file, takes the file out of define mode, and then synchronizes the file to disk.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Perform an operation.
dimid = netcdf.defDim(ncid,'Xdim',50);

% Take file out of define mode.
netcdf.endDef(ncid);

% Synchronize the file to disk.
netcdf.sync(ncid)
```

See Also

`netcdf.close` | `netcdf.create` | `netcdf.open` | `netcdf.endDef`

newplot

Determine where to draw graphics objects

Syntax

```
newplot
h = newplot
h = newplot(hsave)
```

Description

`newplot` prepares a figure and axes for subsequent graphics commands.

`h = newplot` prepares a figure and axes for subsequent graphics commands and returns a handle to the current axes.

`h = newplot(hsave)` prepares and returns an axes, but does not delete any objects whose handles you have assigned to the `hsave` argument, which can be a vector of handles. If `hsave` is not empty, the figure and axes containing `hsave` are prepared for plotting instead of the current axes of the current figure. If `hsave` is empty, `newplot` behaves as if it were called without any inputs.

More About

Tips

To create a simple 2-D plot, use the `plot` function instead.

Use `newplot` at the beginning of high-level graphics code to determine which figure and axes to target for graphics output. Calling `newplot` can change the current figure and current axes. Basically, there are three options when you are drawing graphics in existing figures and axes:

- Add the new graphics without changing any properties or deleting any objects.
- Delete all existing objects whose handles are not hidden before drawing the new objects.

- Delete all existing objects regardless of whether or not their handles are hidden, and reset most properties to their defaults before drawing the new objects (refer to the following table for specific information).

The figure and axes `NextPlot` properties determine how `newplot` behaves. The following two tables describe this behavior with various property values.

First, `newplot` reads the current figure's `NextPlot` property and acts accordingly.

NextPlot	What Happens
`new`	Create a new figure and use it as the current figure.
`add`	Draw to the current figure without clearing any graphics objects already present.
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on` and reset figure `NextPlot` property to `add`. This clears the current figure and is equivalent to issuing the `clf` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset figure properties to their defaults, except `NextPlot` is reset to `add` regardless of user-defined defaults. • `Position`, `Units`, `PaperPosition`, and `PaperUnits` are not reset. This clears and resets the current figure and is equivalent to issuing the `clf reset` command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

NextPlot	Description
`add`	Draw into the current axes, retaining all graphics objects already present.

NextPlot	Description
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on`, but do not reset axes properties. This clears the current axes like the `cla` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset axes properties to their defaults, except `Position` and `Units`. This clears and resets the current axes like the `cla reset` command.

- "Control Graph Display"

See Also

`plot` | `axes` | `cla` | `clf` | `figure` | `hold` | `ishold` | `reset`

Introduced before R2006a

nextDirectory

Class: Tiff

Make next IFD current IFD

Syntax

```
nextDirectory(tiffobj)
```

Description

`nextDirectory(tiffobj)` makes the next image file directory (IFD) in the file the current IFD. `Tiff` object methods operate on the current IFD. Use this method to navigate among IFDs in a TIFF file containing multiple images.

Examples

Make Next Directory the Current Directory

Open a `Tiff` object and change the current IFD to the next IFD in the file.

```
t = Tiff('example.tif','r');
nextDirectory(t);
close(t);
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFReadDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.setDirectory`

nextpow2

Exponent of next higher power of 2

Syntax

```
P = nextpow2(A)
```

Description

`P = nextpow2(A)` returns the exponents for the smallest powers of two that satisfy

$$2^p \geq |A|$$

for each element in `A`.

You can use `nextpow2` to pad the signal you pass to `fft`. Doing so can speed up the computation of the FFT when the signal length is not an exact power of 2.

Examples

Next Power of 2 of Double Integer Values

Define a vector of `double` integer values and calculate the exponents for the next power of 2 higher than those values.

```
a = [1 -2 3 -4 5 9 519];
p = nextpow2(a)
```

```
p =

     0     1     2     2     3     4    10
```

Calculate the positive next powers of 2.

```
np2 = 2.^p
```

```
np2 =

     1     2     4     4     8    16  1024
```

Preserve the sign of the original input values.

```
np2.*sign(a)

ans =

     1    -2     4    -4     8    16  1024
```

Next Power of 2 of Unsigned Integer Values

Define a vector of unsigned integers and calculate the exponents for the next power of 2 higher than those values.

```
a = uint32([1020 4000 32700]);
p = nextpow2(a)

p =

      10      12      15
```

Calculate the next powers of 2 higher than the values in `a`.

```
2.^p

ans =

      1024      4096     32768
```

Optimize FFT with Padding

Use `nextpow2` to increase the performance of `fft` when the length of your signal is not a power of 2.

Create a 1-D vector containing 8191 sample values.

```
x = gallery('uniformdata',[1,8191],0);
```

Calculate the next power of 2 higher than 8191.

```
p = nextpow2(8191);
n = 2^p
```

```
n =

        8192
```

Pass the signal and the next power of 2 to the `fft` function.

```
y = fft(x,n);
```

Input Arguments

A — Input values

scalar, vector, or array of real numbers

Input values, specified as a scalar, vector, or array of real numbers of any numeric type.

Example: `15`

Example: `[-15.123 32.456 63.111]`

Example: `int16([-15 32 63])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`fft` | `log2` | `pow2`

Introduced before R2006a

nnz

Number of nonzero matrix elements

Syntax

```
n = nnz(X)
```

Description

`n = nnz(X)` returns the number of nonzero elements in matrix `X`.

The density of a sparse matrix is `nnz(X)/prod(size(X))`.

Examples

The matrix

```
w = sparse(wilkinson(21));
```

is a tridiagonal matrix with `20` nonzeros on each of three diagonals, so `nnz(w) = 60`.

See Also

`find` | `isa` | `nonzeros` | `nzmax` | `size` | `whos`

Introduced before R2006a

noanimate

Change `EraseMode` of all objects to `normal`

Note: `noanimate` has been removed.

Syntax

```
noanimate(state,fig_handle)
noanimate(state)
```

Description

`noanimate(state,fig_handle)` sets the `EraseMode` of all image, line, patch, surface, and text graphics objects in the specified figure to `normal`. `state` can be the following strings:

- `'save'` — Set the values of the `EraseMode` properties to `normal` for all the appropriate objects in the designated figure.
- `'restore'` — Restore the `EraseMode` properties to the previous values (i.e., the values before calling `noanimate` with the `'save'` argument).

`noanimate(state)` operates on the current figure.

`noanimate` is useful if you want to print the figure to a TIFF or JPEG format.

See Also

`print`

Introduced before R2006a

nonzeros

Nonzero matrix elements

Syntax

```
s = nonzeros(A)
```

Description

`s = nonzeros(A)` returns a full column vector of the nonzero elements in `A`, ordered by columns.

This gives the `s`, but not the `i` and `j`, from `[i,j,s] = find(A)`. Generally,

```
length(s) = nnz(A) <= nzmax(A) <= prod(size(A))
```

See Also

`find` | `isa` | `nnz` | `nzmax` | `size` | `whos`

Introduced before R2006a

norm

Vector and matrix norms

Syntax

```
n = norm(v)
n = norm(v,p)

n = norm(X)
n = norm(X,p)
n = norm(X,'fro')
```

Description

`n = norm(v)` returns the 2-norm or Euclidean norm of vector `v`.

`n = norm(v,p)` returns the vector norm defined by `sum(abs(v)^p)^(1/p)`, where `p` is any positive real value, `Inf`, or `-Inf`.

- If `p` is `Inf`, then `n = max(abs(v))`.
- If `p` is `-Inf`, then `n = min(abs(v))`.

`n = norm(X)` returns the 2-norm or maximum singular value of matrix `X`.

`n = norm(X,p)` returns the *p*-norm of matrix `X`, where `p` is `1`, `2`, or `Inf`.

`n = norm(X,'fro')` returns the Frobenius norm, `sqrt(sum(diag(X'*X)))`.

Examples

1- and 2- Norm of Vector

Calculate the 2-norm of a vector corresponding to the point (-2,3,-1) in 3-D space. The 2-norm is equal to the Euclidean length of the vector.

```
X = [-2 3 -1];
n = norm(X)
```

```
n =

    3.7417
```

Calculate the 1-norm of the vector, which is the sum of the element magnitudes.

```
n = norm(X,1)
```

```
n =

    6
```

2-Norm of Matrix

Calculate the 2-norm of a matrix, which is the largest singular value.

```
X = [2 0 1;-1 1 0;-3 3 0];
n = norm(X)
```

```
n =

    4.7234
```

Frobenius Norm of Sparse Matrix

Use `'fro'` to calculate the Frobenius norm of a sparse matrix, which calculates the 2-norm of the column vector, `S(:)`.

```
S = sparse(1:25,1:25,1);
n = norm(S,'fro')
```

```
n =

     5
```

Input Arguments

v — Input vector

vector

Input vector.

Data Types: `single` | `double`
Complex Number Support: Yes

X — Input matrix

matrix

Input matrix. Use `norm(X,'fro')` when `X` is sparse.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Norm type

2 (default) | positive integer scalar | `Inf` | `-Inf`

Norm type, specified as `2` (default), a different positive integer scalar, `Inf`, or `-Inf`. The valid values of `p` and what they return depend on whether the first input to `norm` is a matrix or vector, as shown in the table.

Note: This table does not reflect the actual algorithms used in calculations.

p	Matrix	Vector
1	`max(sum(abs(X)))`	`sum(abs(X))`
2	`max(svd(X))`	`sum(abs(X).^2)^(1/2)`
Positive, real-valued numeric `p`	—	`sum(abs(X).^p)^(1/p)`
`Inf`	`max(sum(abs(X')))`	`max(abs(X))`
`-Inf`	—	`min(abs(X))`

Output Arguments

n — Matrix or vector norm

scalar

Matrix or vector norm, returned as a scalar. The norm gives a measure of the magnitude of the elements. By convention, `norm` returns `NaN` if the input contains `NaN` values.

See Also

`cond` | `condest` | `hypot` | `normest` | `rcond`

Introduced before R2006a

normest

2-norm estimate

Syntax

```
nrm = normest(S)
nrm = normest(S,tol)
[nrm,count] = normest(...)
```

Description

This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.

`nrm = normest(S)` returns an estimate of the 2-norm of the matrix `S`.

`nrm = normest(S,tol)` uses relative error `tol` instead of the default tolerance `1.e-6`. The value of `tol` determines when the estimate is considered acceptable.

`[nrm,count] = normest(...)` returns an estimate of the 2-norm and also gives the number of power iterations used.

More About

Algorithms

The power iteration involves repeated multiplication by the matrix `S` and its transpose, `S'`. The iteration is carried out until two successive estimates agree to within the specified relative tolerance.

See Also

`cond` | `condest` | `norm` | `rcond` | `svd`

Introduced before R2006a

not, ~

Find logical NOT

Syntax

```
~A
not(A)
```

Description

~A performs a logical NOT of input array A, and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if the input array contains a zero value element at that same array location. Otherwise, that element is set to 0.

The input of the expression can be an array or can be a scalar value. If the input is an array, then the output is an array of the same dimensions. If the input is scalar, then the output is scalar.

`not(A)` is called for the syntax ~A when A is an object.

Examples

If matrix A is

```
     0    29     0    36     0
    23    34    35     0    39
     0    24    31    27     0
     0    29     0     0    34
```

then

```
~A
ans =
     1     0     1     0     1
     0     0     0     1     0
```

```
     1     0     0     0     1
     1     0     1     1     0
```

More About

- “Truth Table for Logical Operations”

See Also

`all` | `and` | `any` | `bitcmp` | `or` | `xor`

Introduced before R2006a

notebook

Open MATLAB Notebook in Microsoft Word software (on Microsoft Windows platforms)

Syntax

```
notebook
notebook('filename')
notebook('-setup')
```

Description

`notebook` starts Microsoft Word software and creates a new MATLAB Notebook titled `Document 1`.

`notebook('filename')` starts Microsoft Word and opens the notebook `filename`, where `filename` is either in the MATLAB current folder or is a full path. If `filename` does not exist, MATLAB creates a new notebook titled `filename`. If the file name extension is not specified, MATLAB assumes `.doc`.

`notebook('-setup')` runs an interactive setup function for MATLAB Notebook. It copies the notebook template, `m-book.dot`, to the Microsoft Word template folder, whose location MATLAB automatically determines from the Windows system registry. Upon completion, MATLAB displays a message indicating whether or not the setup was successful.

More About

- "Create a MATLAB Notebook with Microsoft Word"
- "Publishing MATLAB Code"

Introduced before R2006a

now

Current date and time as serial date number

Syntax

```
t = now
```

Description

`t = now` returns the current date and time as a serial date number. A serial date number represents the whole and fractional number of days from a fixed, preset date (January 0, 0000).

`floor(now)` returns the current date as a serial date number, and `rem(now,1)` returns the current time as a serial date number. `datestr(now)` returns the current date and time as a string.

Examples

```
t1 = now, t2 = rem(now,1)

t1 =

   7.2908e+05


t2 =

    0.4013
```

More About

Tips

- To return a datetime scalar representing the current date and time, type:

```
t = datetime('now')
```

See Also

clock | date | datenum | datestr | datetime

Introduced before R2006a

nthroot

Real nth root of real numbers

Syntax

```
Y = nthroot(X,N)
```

Description

`Y = nthroot(X,N)` returns the real nth root of the elements of `X`. Both `X` and `N` must be real scalars or arrays of the same size. If an element in `X` is negative, then the corresponding element in `N` must be an odd integer.

Examples

Calculate Real Root of Negative Number

Find the real cube root of `-27`.

```
nthroot(-27, 3)
```

```
ans =

    -3
```

For comparison, also calculate `(-27)^(1/3)`.

```
(-27)^(1/3)
```

```
ans =

   1.5000 + 2.5981i
```

The result is the complex cube root of `-27`.

Calculate Several Real Roots of Scalar

Create a vector of roots to calculate, `N`.

```
N = [5 3 -1];
```

Use `nthroot` to calculate several real roots of `-8`.

```
Y = nthroot(-8,N)
```

```
Y =

   -1.5157   -2.0000   -0.1250
```

The result is a vector of the same size as `N`.

Elementwise Roots of Matrix

Create a matrix of bases, `X`, and a matrix of nth roots, `N`.

```
X = [-2 -2 -2; 4 -3 -5]
N = [1 -1 3; 1/2 5 3]
```

```
X =

    -2    -2    -2
     4    -3    -5


N =

    1.0000   -1.0000    3.0000
    0.5000    5.0000    3.0000
```

Each element in `X` corresponds to an element in `N`.

Calculate the real nth roots of the elements in `X`.

```
Y = nthroot(X,N)
```

```
Y =

   -2.0000   -0.5000   -1.2599
   16.0000   -1.2457   -1.7100
```

Except for the signs (which are treated separately), the result is comparable to `abs(X).^(1./N)`. By contrast, you can calculate the complex roots using `X.^(1./N)`.

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `X` can be either a scalar or an array of the same size as `N`. The elements of `X` must be real.

Data Types: `single` | `double`

N — Roots to calculate

scalar | array of same size as `X`

Roots to calculate, specified as a scalar or array of the same size as `X`. The elements of `N` must be real. If an element in `X` is negative, the corresponding element in `N` must be an odd integer.

Data Types: `single` | `double`

More About

Tips

- While `power` is a more efficient function for computing the roots of numbers, in cases where both real and complex roots exist, `power` returns only the complex roots. In these cases, use `nthroot` to obtain the real roots.

See Also

`power` | `sqrt`

Introduced before R2006a

null

Null space

Syntax

```
Z = null(A)
Z = null(A,'r')
```

Description

`Z = null(A)` is an orthonormal basis for the null space of `A` obtained from the singular value decomposition. That is, `A*Z` has negligible elements, `size(Z,2)` is the nullity of `A`, and `Z'*Z = I`.

`Z = null(A,'r')` is a "rational" basis for the null space obtained from the reduced row echelon form. `A*Z` is zero, `size(Z,2)` is an estimate for the nullity of `A`, and, if `A` is a small matrix with integer elements, the elements of the reduced row echelon form (as computed using `rref`) are ratios of small integers.

The orthonormal basis is preferable numerically, while the rational basis may be preferable pedagogically.

Examples

Example 1

Compute the orthonormal basis for the null space of a matrix A.

```
A = [1     2     3
     1     2     3
     1     2     3];

Z = null(A);
A*Z

ans =
```

```
  1.0e-015 *
     0.2220    0.2220
     0.2220    0.2220
     0.2220    0.2220

Z'*Z

ans =
    1.0000   -0.0000
   -0.0000    1.0000
```

Example 2

Compute the 1-norm of the matrix A*Z and determine that it is within a small tolerance.

```
norm(A*Z,1) < 1e-12
ans =
    1
```

Example 3

Compute the rational basis for the null space of the same matrix A.

```
ZR = null(A,'r')

ZR =
    -2    -3
     1     0
     0     1

A*ZR

ans =

     0     0
     0     0
     0     0
```

See Also

orth | rank | rref | svd

Introduced before R2006a

num2cell

Convert array to cell array with consistently sized cells

Syntax

```
C = num2cell(A)
C = num2cell(A,dim)
```

Description

`C = num2cell(A)` converts array A into cell array C by placing each element of A into a separate cell in C. Array A need not be numeric.

`C = num2cell(A,dim)` splits the contents of A into separate cells of C, where dim specifies which dimensions of A to include in each cell. dim can be a scalar or a vector of dimensions. For example, if A has 2 rows and 3 columns, then:

- `num2cell(A,1)` creates a 1-by-3 cell array `C`, where each cell contains a 2-by-1 column of `A`.
- `num2cell(A,2)` creates a 2-by-1 cell array `C`, where each cell contains a 1-by-3 row of `A`.
- `num2cell(A,[1 2])` creates a 1-by-1 cell array `C`, where the cell contains the entire array `A`.

Examples

Convert Arrays to Cell Array

Place all elements of a numeric array into separate cells.

```
a = magic(3)
c = num2cell(a)
```

```
a =
```

```
     8     1     6
     3     5     7
     4     9     2


c =

    [8]    [1]    [6]
    [3]    [5]    [7]
    [4]    [9]    [2]
```

Place individual letters of a word into separate cells of an array.

```
a = ['four';'five';'nine']
c = num2cell(a)


a =

four
five
nine


c =

    'f'    'o'    'u'    'r'
    'f'    'i'    'v'    'e'
    'n'    'i'    'n'    'e'
```

Create Cell Array of Numeric Arrays

Generate a 4-by-3-by-2 numeric array, and then create a 1-by-3-by-2 cell array of 4-by-1 column vectors.

```
A = reshape(1:12,4,3);
A(:,:,2) = A*10
C = num2cell(A,1)


A(:,:,1) =
```

```
     1     5     9
     2     6    10
     3     7    11
     4     8    12


A(:,:,2) =

    10    50    90
    20    60   100
    30    70   110
    40    80   120


C(:,:,1) =

    [4x1 double]    [4x1 double]    [4x1 double]


C(:,:,2) =

    [4x1 double]    [4x1 double]    [4x1 double]
```

Each 4-by-1 vector contains elements from along the *first* dimension of `A`:

```
C{1}
```

```
ans =

     1
     2
     3
     4
```

Create a 4-by-1-by-2 cell array of 1-by-3 numeric arrays.

```
C = num2cell(A,2)
```

```
C(:,:,1) =

    [1x3 double]
```

```
    [1x3 double]
    [1x3 double]
    [1x3 double]


C(:,:,2) =

    [1x3 double]
    [1x3 double]
    [1x3 double]
    [1x3 double]
```

Each 1-by-3 row vector contains elements from along the *second* dimension of A:

```
C{1}


ans =

     1     5     9
```

Finally, create a 4-by-3 cell array of 1-by-1-by-2 numeric arrays.

```
C = num2cell(A,3)


C =

    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
```

Each 1-by-1-by-2 vector contains elements from along the *third* dimension of A:

```
C{1}


ans(:,:,1) =

     1
```

```
ans(:,:,2) =

    10
```

Combine Across Multiple Dimensions

Create a cell array by combining elements into numeric arrays along several dimensions.

```
A = reshape(1:12,4,3);
A(:,:,2) = A*10
c = num2cell(A,[1 3])
```

```
A(:,:,1) =

     1     5     9
     2     6    10
     3     7    11
     4     8    12


A(:,:,2) =

    10    50    90
    20    60   100
    30    70   110
    40    80   120


c =

    [4x1x2 double]    [4x1x2 double]    [4x1x2 double]
```

Each 4-by-1-by-2 array contains elements from along the first and third dimension of `A`:

```
c{1}
```

```
ans(:,:,1) =

     1
     2
```

```
     3
     4


ans(:,:,2) =

    10
    20
    30
    40


c = num2cell(A,[2 3])


c =

    [1x3x2 double]
    [1x3x2 double]
    [1x3x2 double]
    [1x3x2 double]
```

Input Arguments

A — Input

any type of multidimensional array

Input, specified as any type of multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `cell` | `function_handle`

dim — Dimension of A

positive integer | positive vector of integers

Dimension of A, specified as a positive integer or a vector of positive integers. dim must be between 1 and `ndims`(A).

Elements need not be in numeric order. However, `num2cell` permutes the dimensions of the arrays in each cell of C to match the order of the specified dimensions.

Data Types: `double`

Output Arguments

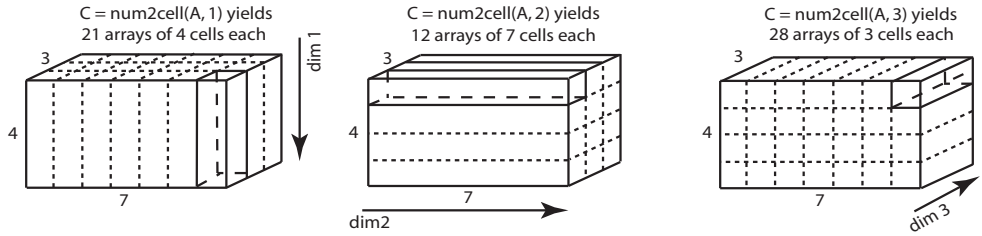
C — Resulting array

cell array

Resulting array, returned as a cell array. The size of C depends on the size of A and the values of dim.

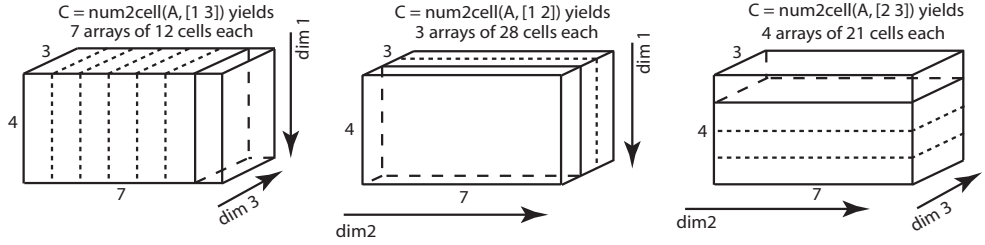
- If dim is not specified, then C is the same size as A.
- If dim is a scalar, then C contains `numel(A)/size(A,dim)` cells. If dim is 1 or 2, then each cell contains a column or row vector, respectively. If dim > 2, then each cell contains an array whose dimth dimensional length is `size(A,dim)`, and whose other dimensions are all singletons.

  For example, given a 4-by-7-by-3 array, `A`, this figure shows how `num2cell` creates cells corresponding to dim values of `1`, `2`, and `3`.



- If dim is a vector containing `N` values, then C has `numel(A)/prod([size(A,dim(1)),...,size(A,vdim(N))])` cells. Each cell contains an array whose `dim(i)`th dimension has a length of `size(A,dim(i))` and whose other dimensions are singletons.

  For example, given a 4-by-7-by-3 array, you can specify dim as an positive integer vector to create cell arrays of different dimensions.



See Also

`cat` | `cell2mat` | `mat2cell`

Introduced before R2006a

num2hex

Convert singles and doubles to IEEE hexadecimal strings

Syntax

```
num2hex(X)
```

Description

If `X` is a single or double precision array with n elements, `num2hex(X)` is an n-by-8 or n-by-16 char array of the hexadecimal floating-point representation. The same representation is printed with format `hex`.

Examples

```
num2hex([1 0 0.1 -pi Inf NaN])
```

returns

```
ans =

3ff0000000000000
0000000000000000
3fb999999999999a
c00921fb54442d18
7ff0000000000000
fff8000000000000
num2hex(single([1 0 0.1 -pi Inf NaN]))
```

returns

```
ans =

3f800000
00000000
3dcccccd
c0490fdb
```

```
7f800000
ffc00000
```

See Also

`hex2num` | `format` | `dec2hex`

Introduced before R2006a

num2str

Convert number to string

Syntax

```
s = num2str(A)
s = num2str(A,precision)
s = num2str(A,formatSpec)
```

Description

`s = num2str(A)` converts a numeric array into a string representation. The output format depends on the magnitudes of the original values. `num2str` is useful for labeling and titling plots with numeric values.

`s = num2str(A,precision)` returns a string representation with the maximum number of significant digits specified by `precision`.

`s = num2str(A,formatSpec)` applies a format specified by `formatSpec` to all elements of `A`.

Examples

Default Conversions of Floating-Point Values

Convert the floating-point values returned by `pi` and `eps` to strings.

```
s = num2str(pi)
```

```
s =

3.1416
```

```
s = num2str(eps)
```

```
s =

2.2204e-16
```

Specify Precision

Specify the maximum number of significant digits for floating-point values.

```
A = gallery('normaldata',[2,2],0);
s = num2str(A,3)
```

```
s =

-0.433      0.125
 -1.67      0.288
```

Specify Formatting

Specify the width, precision, and other formatting for an array of floating-point values.

```
A = gallery('uniformdata',[2,3],0) * 9999;
s = num2str(A,'%10.5e\n')
```

```
s =

9.50034e+03
6.06782e+03
8.91210e+03
2.31115e+03
4.85934e+03
7.62021e+03
```

The format `'%10.5e'` prints each value in exponential format with five decimal places, and `'\n'` prints a new line character.

Input Arguments

A — Input array

numeric array

Input array, specified as a numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`precision` — Maximum number of significant digits

positive integer

Maximum number of significant digits in the output string, specified as a positive integer.

Note: If you specify `precision` to exceed the precision of the input floating-point data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

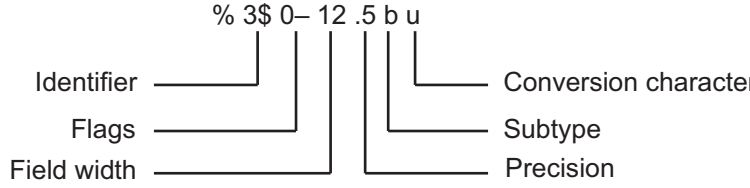
`formatSpec` — Format of output fields

string containing formatting operators

Format of the output fields, specified as a string containing formatting operators. `formatSpec` also can include ordinary text and special characters.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).



Conversion Character

This table shows conversion characters to format numeric and character data as strings.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a`–`f`
	`%X`	Same as `%x`, uppercase letters `A`–`F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters	`%c`	Single character
	`%s`	String of characters

Optional Operators

The optional identifier, flags, field width, precision, and subtype operators further define the format of the output string.

- **Identifier**

Order for processing values from the input list. Use the syntax *n*$, where *n* represents the position of the value in the input list.

Example: `'%3$s %2$s %1$s %2$s'` prints inputs `'A'`, `'B'`, `'C'` as follows: `C B A B`.

- **Flags**

`'–'`	Left-justify. **Example:** `%-5.2f`
`'+'`	Always print a sign character (+ or –) for any value. **Example:** `%+5.2f`
`' '`	Insert a space before the value. **Example:** `% 5.2f`
`'0'`	Pad to field width with zeros before the value. **Example:** `%05.2f`
`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. **Example:** `%#5.0f`

- **Field Width**

Minimum number of characters to print. The field width operator can be a number, or an asterisk (`*`) to refer to an argument in the input list.

Example: The input list `('%12d',intmax)` is equivalent to `('%*d', 12, intmax)`.

The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`

For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `' 3.142'`

The precision operator can be a number, or an asterisk (`*`) to refer to an argument in the input list.

Example: The input list `('%6.4f', pi)` is equivalent to `('%*.*f', 6, 4, pi)`.

Note: If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

- **Subtypes**

Certain conversion characters can support a subtype. The subtype operator immediately precedes the conversion character. This table shows the conversions that can use subtypes.

Input Value Type	Subtype and Conversion Character	Output Value Type

Floating-point number	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Integer	`%ld` or `%li`
	`%lo`
	`%lu`
	`%lx` or `%lX`
		64-bit value

Integer	`%hd` or `%hi`
	`%ho`
	`%hu`
		16-bit value

Input Value Type	Subtype and Conversion Character	Output Value Type
	`%hx` or `%hX`	

Text Before or After Formatting Operators

`formatSpec` can also include additional text before a percent sign, `%`, or after a conversion character. The text can be:

- Ordinary text to print.
- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in `formatSpec`.

Special Character	Representation
Single quotation mark	`''`
Percent character	`%%`
Backslash	`\\`
Alarm	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Character whose ASCII code is the hexadecimal number, `N`	`\xN`
Character whose ASCII code is the octal number, `N`	`\N`

Notable Behavior of Conversions with Formatting Operators

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a string conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.

  Example: `'%s'` converts `pi` to `3.141593e+00`.

- If you apply a string conversion (`%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters.

  Example: `'%s'` converts `[65 66 67]` to `ABC`.

Output Arguments

s — String representation of input array

character array

String representation of the input array, returned as a character array.

More About

Algorithms

`num2str` trims any leading spaces from a string, even when formatSpec includes a space character flag. For example, `num2str(42.67,'% 10.2f')` returns a 1-by-5 character array `'42.67'`.

See Also

`cast` | `int2str` | `mat2str` | `sprintf` | `str2num`

Introduced before R2006a

numberOfStrips

Class: Tiff

Total number of strips in image

Syntax

```
numStrips = numberOfStrips(tiffobj)
```

Description

`numStrips = numberOfStrips(tiffobj)` returns the total number of strips in the image.

Examples

Determine Number of Strips in Image

Determine the number of strips in the second image of a file.

Create a `Tiff` object associated with the example file, `example.tif`.

```
t = Tiff('example.tif','r');
```

When the `Tiff` object is created, the first image in the file is the current image file directory.

Make the second image the current directory.

```
nextDirectory(t)
```

Get the number of strips in the image if the image has a stripped organization.

```
if ~isTiled(t)
 numStrips = numberOfStrips(t)
end
```

```
numStrips =
```

```
     7
```

The image has 7 strips.

Close the `Tiff` object.

```
close(t)
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

References

This method corresponds to the `TIFFNumberOfStrips` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.numberOfTiles` | `Tiff.isTiled`

numberOfTiles

Class: Tiff

Total number of tiles in image

Syntax

```
numTiles = numberOfTiles(tiffobj)
```

Description

`numTiles = numberOfTiles(tiffobj)` returns the total number of tiles in the image.

Examples

Determine Number of Tiles in Image

Create a `Tiff` object associated with the example file, `example.tif`. Get the number of tiles in the image if the image has a tiled organization.

```
t = Tiff('example.tif','r');
if isTiled(t)
 nTiles = numberOfTiles(t)
end
```

```
nTiles =

   110
```

The image has 110 tiles.

Close the `Tiff` object.

```
close(t)
```

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

References

This method corresponds to the `TIFFNumberOfTiles` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.numberOfStrips` | `Tiff.isTiled`

numel

Number of array elements

Syntax

```
n = numel(A)
```

Description

`n = numel(A)` returns the number of elements, `n`, in array `A`, equivalent to `prod(size(A))`.

Examples

Number of Elements in 3-D Matrix

Create a 4-by-4-by-2 matrix.

```
A = magic(4);
A(:,:,2) = A'
```

```
A(:,:,1) =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

A(:,:,2) =
    16     5     9     4
     2    11     7    14
     3    10     6    15
    13     8    12     1
```

`numel` counts 32 elements in the matrix.

```
n = numel(A)
```

```
n =

    32
```

Number of Elements in Cell Array of Strings

Create a cell array of strings.

```
A = {'dog','cat','fish','horse'};
```

`numel` counts 4 string elements in the array.

```
n = numel(A)
```

```
n =

     4
```

Number of Elements in Table

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

A = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
A =

                Age    Height    Weight     BloodPressure
                ---    ------    ------    ---------------
    Smith       38     71        176       124          93
    Johnson     43     69        163       109          77
    Williams    38     64        131       125          83
    Jones       40     67        133       117          75
    Brown       49     64        119       122          80
```

Find the number of elements in the table.

```
n = numel(A)
```

```
n =

    20
```

`numel` returns a value equivalent to `prod(size(A))` corresponding to the 5 rows and 4 variables.

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. This includes numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, calendarDuration arrays, tables, structure arrays, cell arrays, and object arrays.

Limitations

- If `A` is a table, `numel` returns the number of elements in the table, `A`, equivalent to `prod(size(A))`. Variables in a table can have multiple columns, but `numel(A)` only accounts for the number of rows and number of variables.

More About

- "Overloading numel, subsref, and subsasgn"

See Also

`prod` | `size` | `subsref`

Introduced before R2006a

nzmax

Amount of storage allocated for nonzero matrix elements

Syntax

```
n = nzmax(S)
```

Description

`n = nzmax(S)` returns the amount of storage allocated for nonzero elements.

If `S` is a sparse matrix...	`nzmax(S)` is the number of storage locations allocated for the nonzero elements in `S`.
If `S` is a full matrix...	`nzmax(S) = prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i,j,s,m,n,nzmax)` or its simpler form, `spalloc(m,n,nzmax)`, can set `nzmax` in anticipation of later fill-in.

See Also

`find` | `isa` | `nnz` | `nonzeros` | `size` | `whos`

Introduced before R2006a

matlab.lang.ObjectUpdateFailure class

Package: matlab.lang

Class representing objects that cannot be updated to new class definition

Description

MATLAB converts objects to instances of `matlab.lang.ObjectUpdateFailure` when a class definition changes, but existing objects of that class cannot be updated to conform to the new definition. When objects cannot be updated, the automatic update process replaces existing objects of the class with `matlab.lang.ObjectUpdateFailure` objects.

Typical cases where objects are converted to `matlab.lang.ObjectUpdateFailure` objects include cases where a class:

- Is made abstract
- Is converted to an enumeration class
- Changes its heterogeneous root
- Inherits subsref, subsasgn, cat, vertcat, or horzcat methods

Attributes

`Sealed`	`true`
`HandleCompatible`	`true`

To learn about attributes of classes, see Class Attributes in the MATLAB Object-Oriented Programming documentation.

More About

- "Automatic Updates for Modified Classes"
- Class Attributes

ode15i

Solve fully implicit differential equations, variable order method

Syntax

```
[T,Y] = ode15i(odefun,tspan,y0,yp0)
[T,Y] = ode15i(odefun,tspan,y0,yp0,options)
[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)
sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)
```

Arguments

The following table lists the input arguments for `ode15i`.

`odefun`	A function handle that evaluates the left side of the differential equations, which are of the form $f(t,y,y') = 0$.
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0, yp0`	Vectors of initial conditions for y and y' respectively.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for `ode15i`.

`T`	Column vector of time points
`Y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

Description

`[T,Y] = ode15i(odefun,tspan,y0,yp0)` with `tspan = [t0 tf]` integrates the system of differential equations $f(t,y,y') = 0$ from time `t0` to `tf` with initial conditions `y0` and `yp0`. `odefun` is a function handle. Function `ode15i` solves ODEs and DAEs of index 1. The initial conditions must be consistent, meaning that $f(t0,y0,yp0) = 0$. You

can use the function `decic` to compute consistent initial conditions close to guessed values. Function `odefun(t,y,yp)`, for a scalar `t` and column vectors `y` and `yp`, must return a column vector corresponding to $f(t,y,y')$. Each row in the solution array `Y` corresponds to a time returned in the column vector `T`. To obtain solutions at specific times `t0,t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, if necessary.

`[T,Y] = ode15i(odefun,tspan,y0,yp0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used options include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components `1e-6` by default). See `odeset` for details.

`[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)` with the `'Events'` property in `options` set to a function `events`, solves as above while also finding where functions of (t,y,y'), called event functions, are zero. The function `events` is of the form `[value,isterminal,direction] = events(t,y,yp)` and includes the necessary event functions. Code the function `events` so that the `i`th element of each output vector corresponds to the `i`th event. For the `i`th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Output `TE` is a column vector of times at which events occur. Rows of `YE` are the corresponding solutions, and indices in vector `IE` specify which event occurred. See "Integrator Options" in the MATLAB Mathematics documentation for more information.

`sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)` returns a structure that can be used with `deval` to evaluate the solution at any point between `t0` and `tfinal`. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver. If you specify the `Events` option and a terminal event is detected, `sol.x(end)` contains the end of the step at which the event occurred.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Options

`ode15i` accepts the following parameters in `options`. For more information, see `odeset` and Changing ODE Integration Properties in the MATLAB Mathematics documentation.

Error control	`RelTol`, `AbsTol`, `NormControl`
Solver output	`OutputFcn`, `OutputSel`, `Refine`, `Stats`
Event location	`Events`
Step size	`MaxStep`, `InitialStep`
Jacobian matrix	`Jacobian`, `JPattern`, `Vectorized`

Solver Output

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

Jacobian Matrices

The Jacobian matrices $\partial f/\partial y$ and $\partial f/\partial y'$ are critical to reliability and efficiency. You can provide these matrices as one of the following:

- Function of the form `[dfdy,dfdyp] = FJAC(t,y,yp)` that computes the Jacobian matrices. If `FJAC` returns an empty matrix `[]` for either `dfdy` or `dfdyp`, then `ode15i` approximates that matrix by finite differences.
- Cell array of two constant matrices `{dfdy,dfdyp}`, either of which could be empty.

Use `odeset` to set the `Jacobian` option to the function or cell array. If you do not set the `Jacobian` option, `ode15i` approximates both Jacobian matrices by finite differences.

For `ode15i`, `Vectorized` is a two-element cell array. Set the first element to `'on'` if `odefun(t,[y1,y2,...],yp)` returns `[odefun(t,y1,yp),odefun(t,y2,yp),...]`. Set the second element to `'on'` if `odefun(t,y,[yp1,yp2,...])` returns `[odefun(t,y,yp1),odefun(t,y,yp2),...]`. The default value of `Vectorized` is `{'off','off'}`.

For `ode15i`, `JPattern` is also a two-element sparse matrix cell array. If $\partial f/\partial y$ or $\partial f/\partial y'$ is a sparse matrix, set `JPattern` to the sparsity patterns, `{SPDY,SPDYP}`. A sparsity pattern of $\partial f/\partial y$ is a sparse matrix `SPDY` with `SPDY(i,j) = 1` if component `i` of `f(t,y,yp)` depends on component `j` of `y`, and `0` otherwise. Use `SPDY = []` to indicate that $\partial f/\partial y$ is a full matrix. Similarly for $\partial f/\partial y'$ and `SPDYP`. The default value of `JPattern` is `{[],[]}`.

Examples

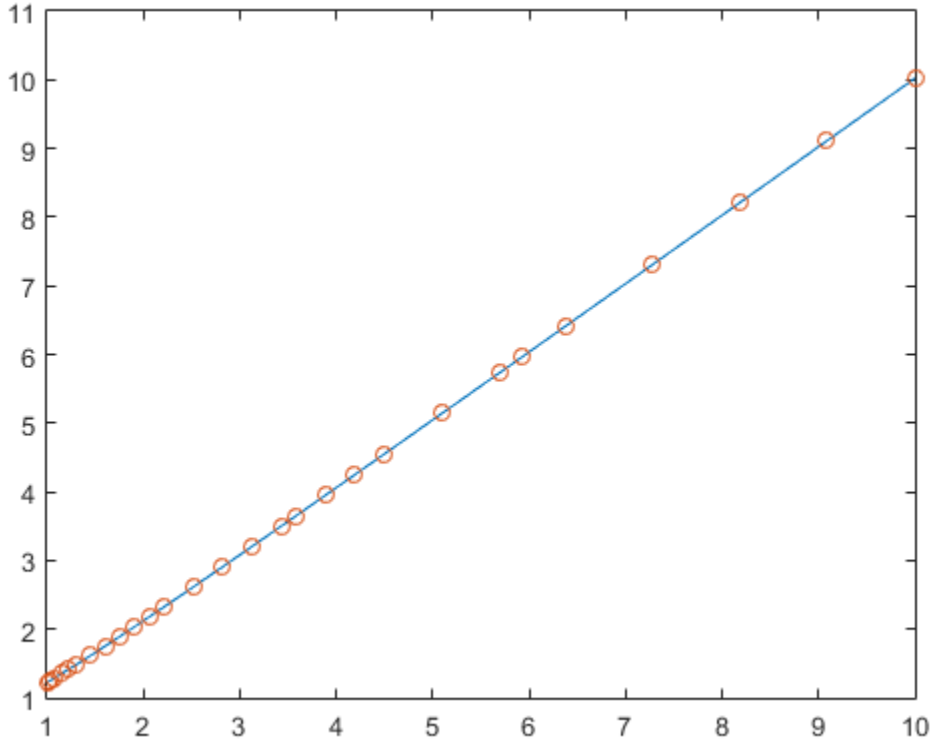
Solve Weissinger Implicit ODE

This example uses a helper function, `decic`, to hold fixed the initial value for *y(t0)* and compute a consistent initial value for *y'(t0)* for the Weissinger implicit ODE. The Weissinger function evaluates the residual of the implicit ODE.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

Use `ode15i` to solve the ODE, and then plot the numerical solution, `y`, against the analytical solution, `ytrue`.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,t,ytrue,'o')
```



Other Examples

The files, `ihb1dae.m` and `iburgersode.m`, are examples of implicit ODEs.

See Also

`decic` | `deval` | `odeget` | `odeset` | `function_handle` | `ode45` | `ode23` | `ode113` | `ode15s` | `ode23s` | `ode23t` | `ode23tb`

Introduced before R2006a

ode15s

Solve stiff differential equations and DAEs; variable order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.
`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first

input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.

`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T, [Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2),...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.
- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-5544 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

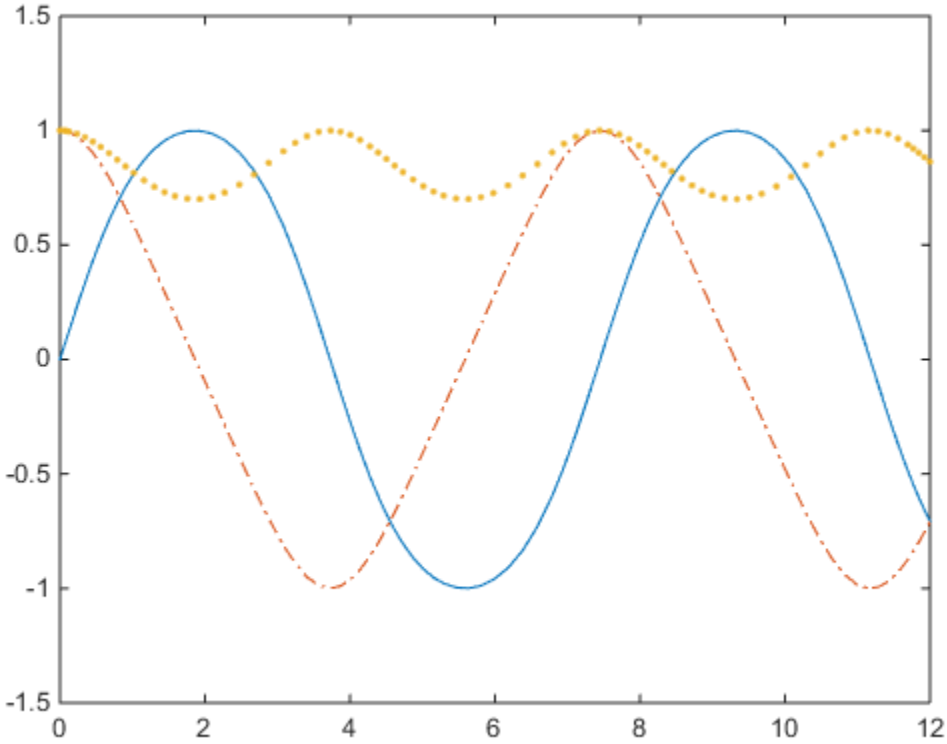
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array Y versus T shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

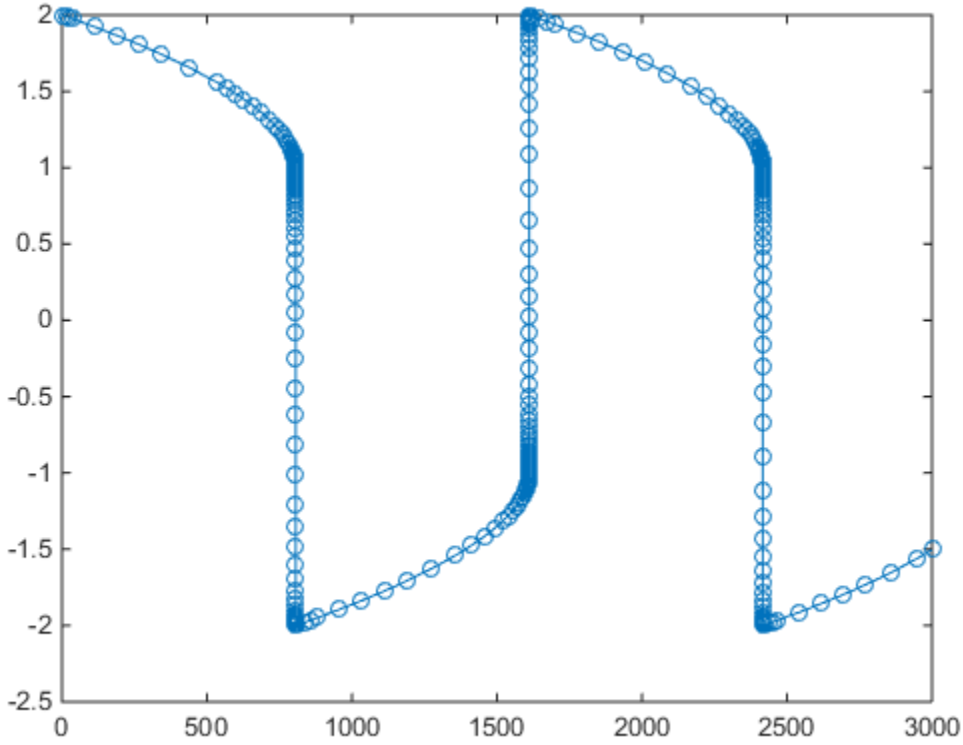
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

`y'(t) + f(t)y(t) = g(t)`
The initial condition is `y(1) = 1`, where the function `f(t)` is defined by the `n`-by-`1` vector `f` evaluated at times `ft`, and the function `g(t)` is defined by the `m`-by-`1` vector `g` evaluated at times `gt`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

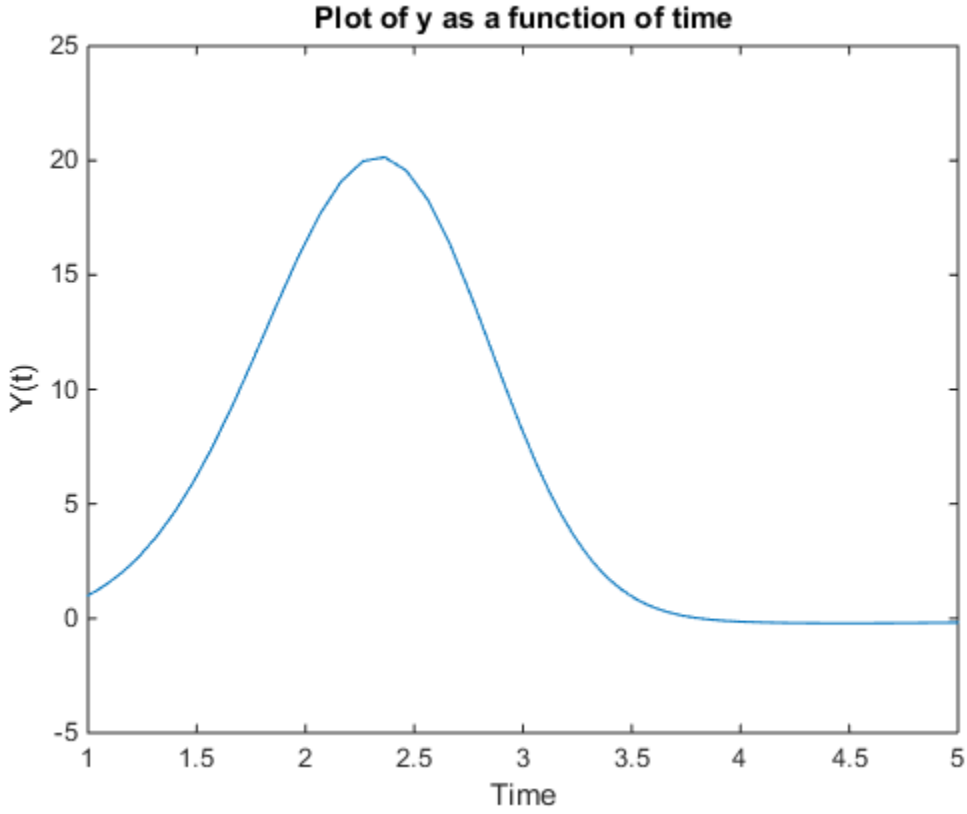
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=1) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time');
ylabel('Y(t)');
```



More About

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, “Transient Simulation of Silicon Devices and Circuits,” *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

ode23

Solve nonstiff differential equations; low order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.
`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first

input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.

`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T, [Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2),...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.
- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-5558 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

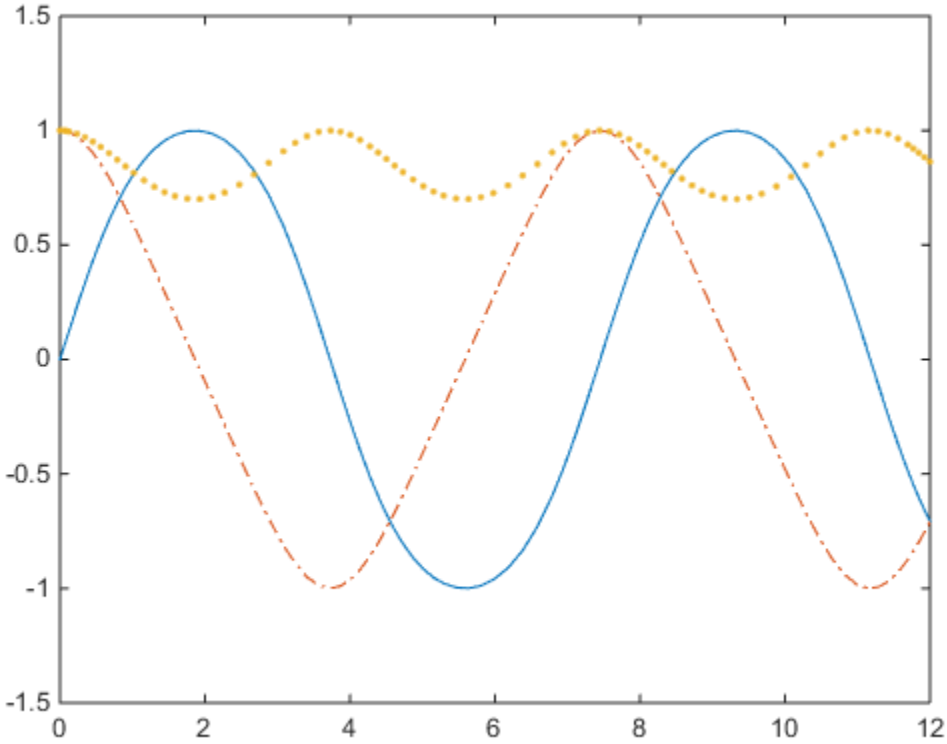
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array Y versus T shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

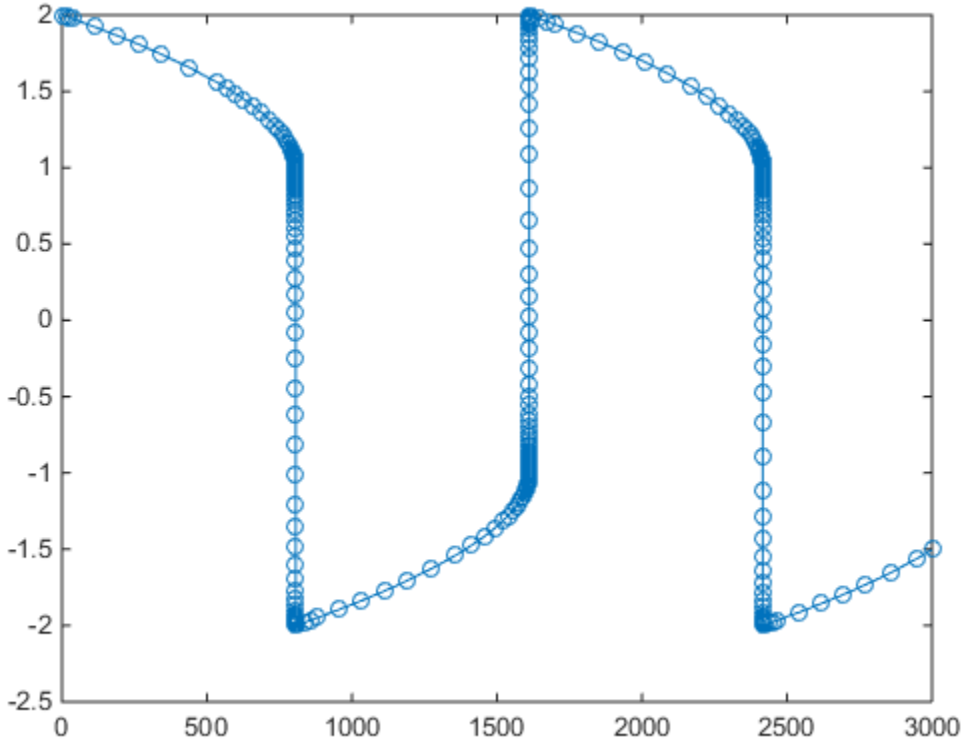
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

`y'(t) + f(t)y(t) = g(t)`
The initial condition is `y(1) = 1`, where the function `f(t)` is defined by the `n`-by-`1` vector `f` evaluated at times `ft`, and the function `g(t)` is defined by the `m`-by-`1` vector `g` evaluated at times `gt`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

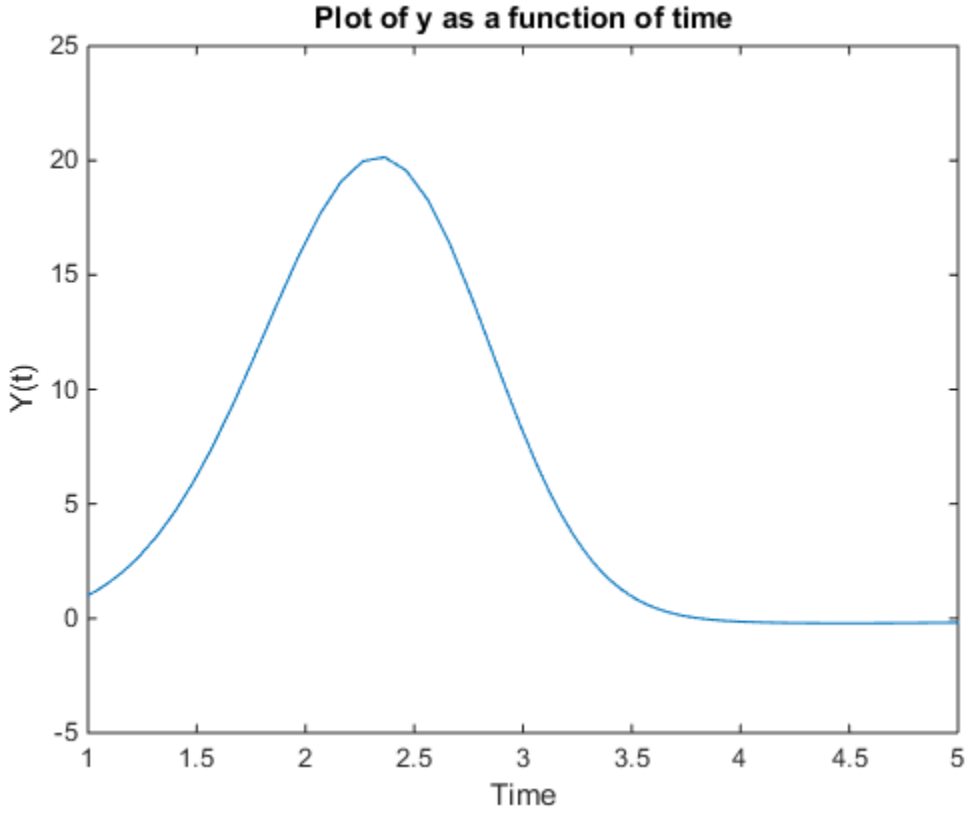
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=1) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time');
ylabel('Y(t)');
```



More About

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(t`$_n$`)`, it needs only the solution at the immediately preceding time point, `y(t`$_{n-1}$`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, “Transient Simulation of Silicon Devices and Circuits,” *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

ode23s

Solve stiff differential equations; low order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.
`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first

input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.

`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T, [Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2),...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.
- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-5572 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

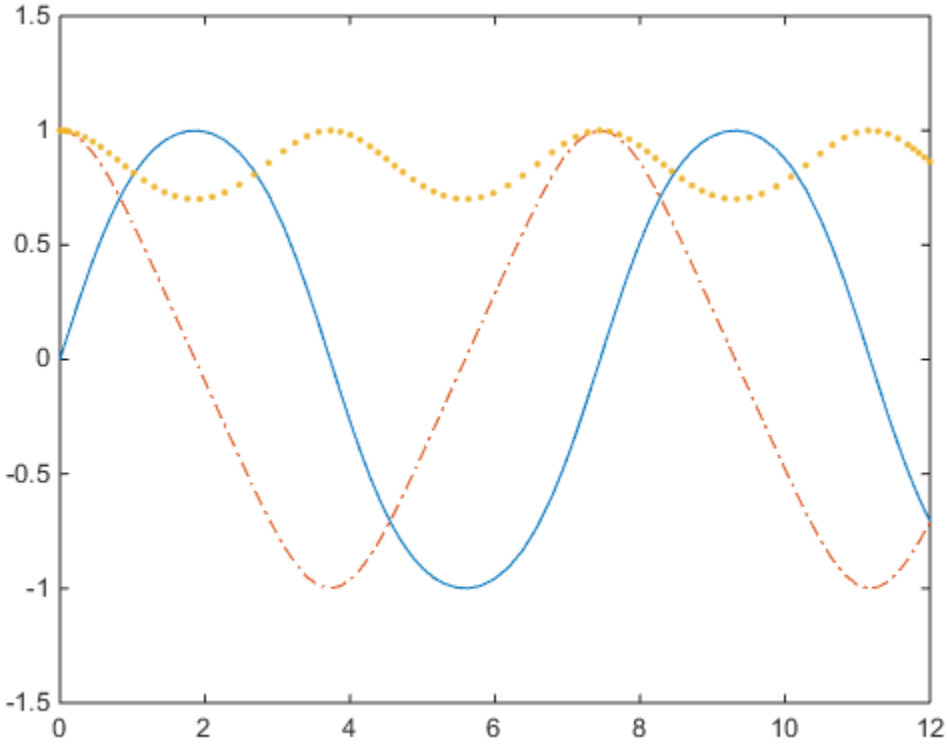
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array Y versus T shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

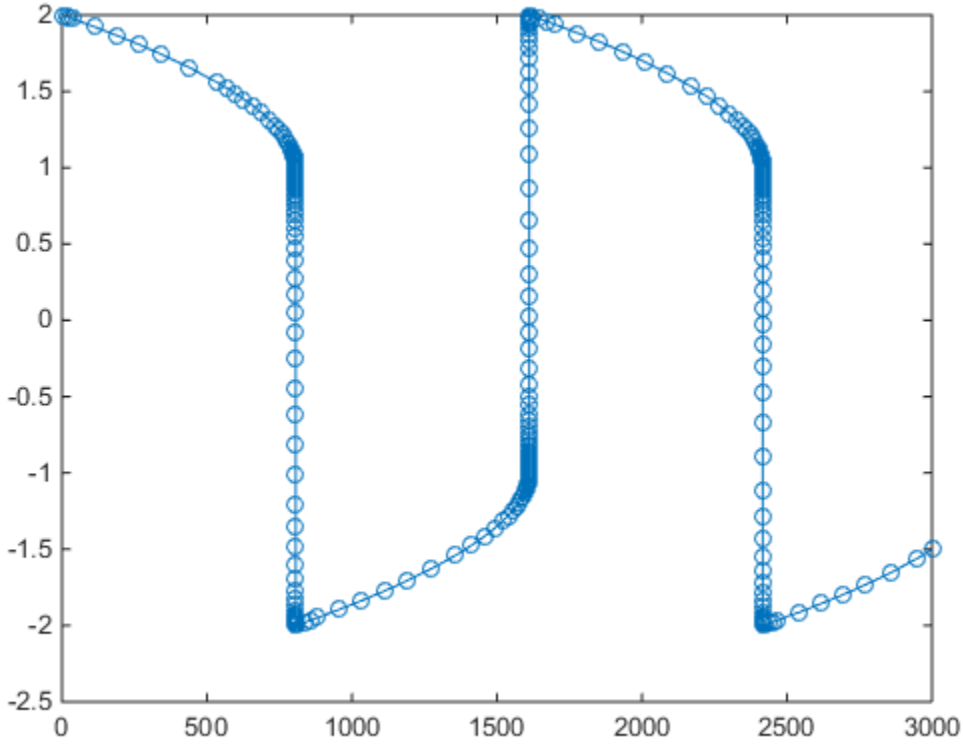
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

`y'(t) + f(t)y(t) = g(t)`
The initial condition is `y(1) = 1`, where the function `f(t)` is defined by the `n`-by-`1` vector `f` evaluated at times `ft`, and the function `g(t)` is defined by the `m`-by-`1` vector `g` evaluated at times `gt`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

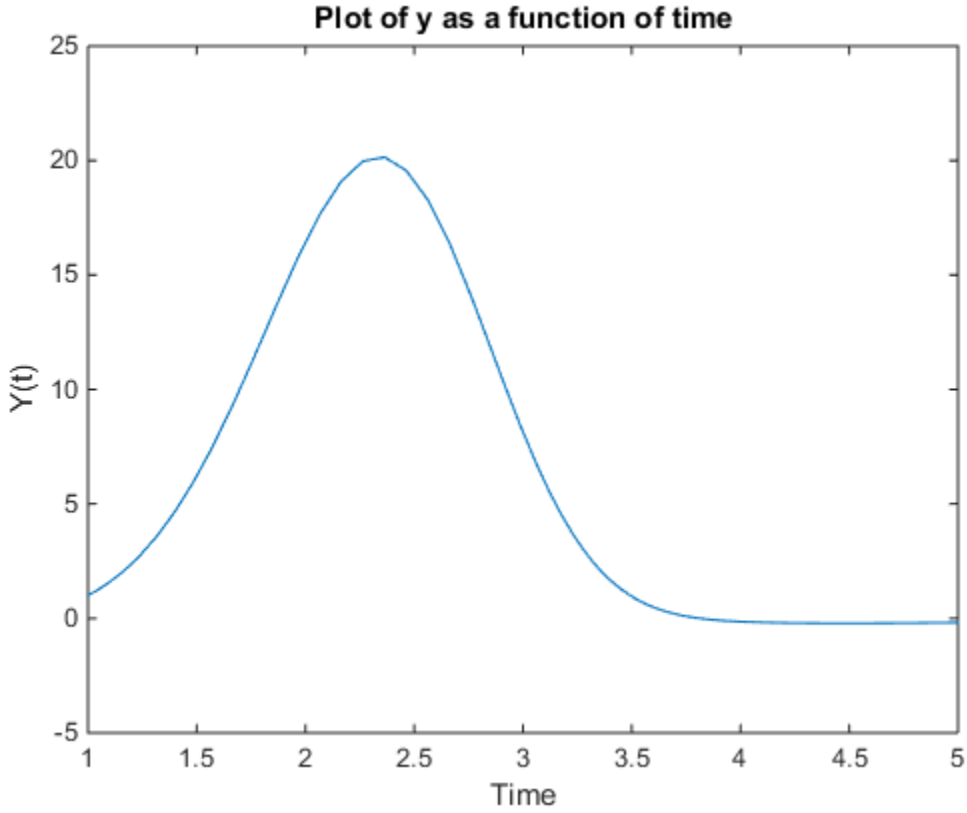
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=1) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time');
ylabel('Y(t)');
```



More About

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(`t_n`)`, it needs only the solution at the immediately preceding time point, `y(`t_{n-1}`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, “Transient Simulation of Silicon Devices and Circuits,” *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

ode23t

Solve moderately stiff ODEs and DAEs; trapezoidal rule

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.
`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first

input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.

`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T, [Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2),...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.
- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-5586 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

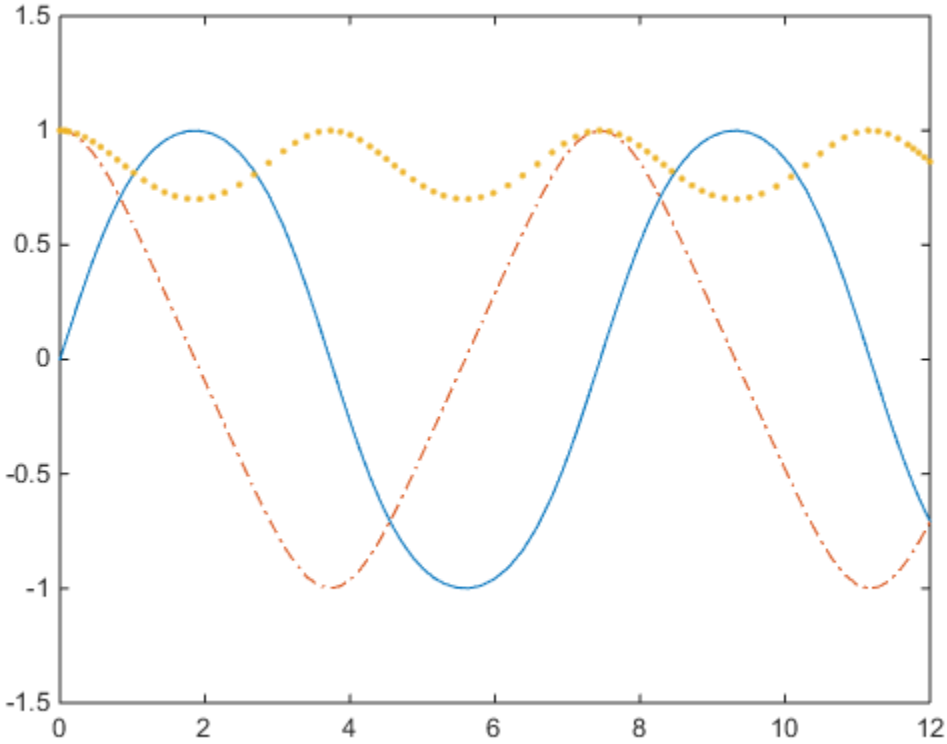
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array Y versus T shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

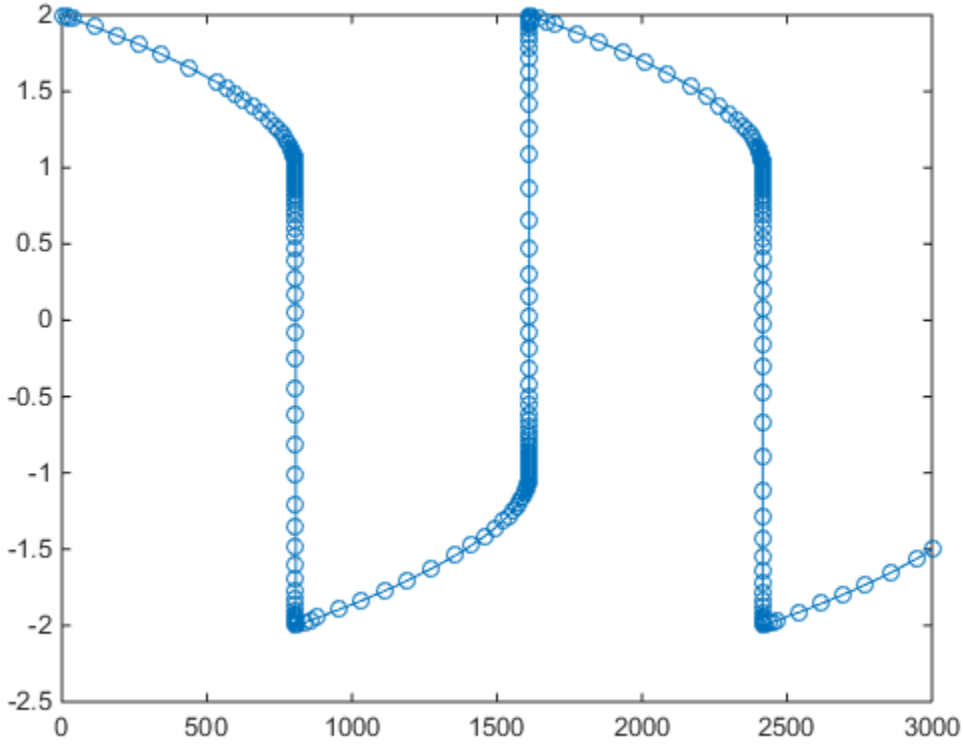
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

`y'(t) + f(t)y(t) = g(t)`
The initial condition is `y(1) = 1`, where the function `f(t)` is defined by the `n`-by-`1` vector `f` evaluated at times `ft`, and the function `g(t)` is defined by the `m`-by-`1` vector `g` evaluated at times `gt`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

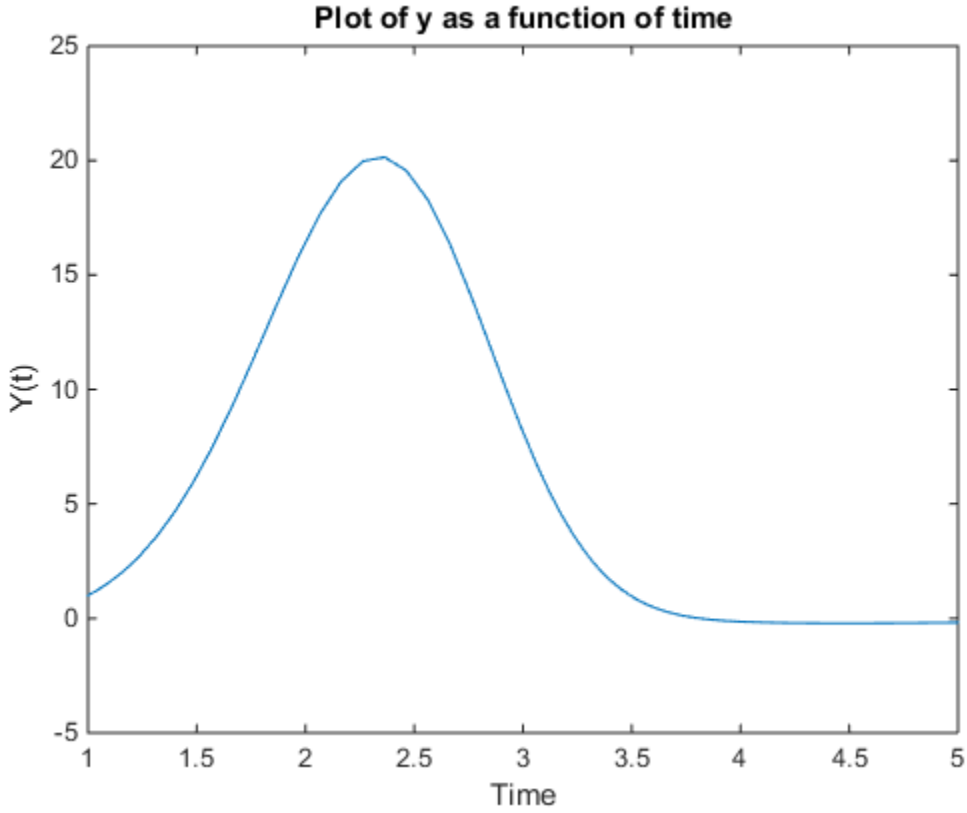
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=1) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time');
ylabel('Y(t)');
```



More About

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, “Transient Simulation of Silicon Devices and Circuits,” *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

ode23tb

Solve stiff differential equations; low order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.
`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first

input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.

`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2),...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.
- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-5600 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

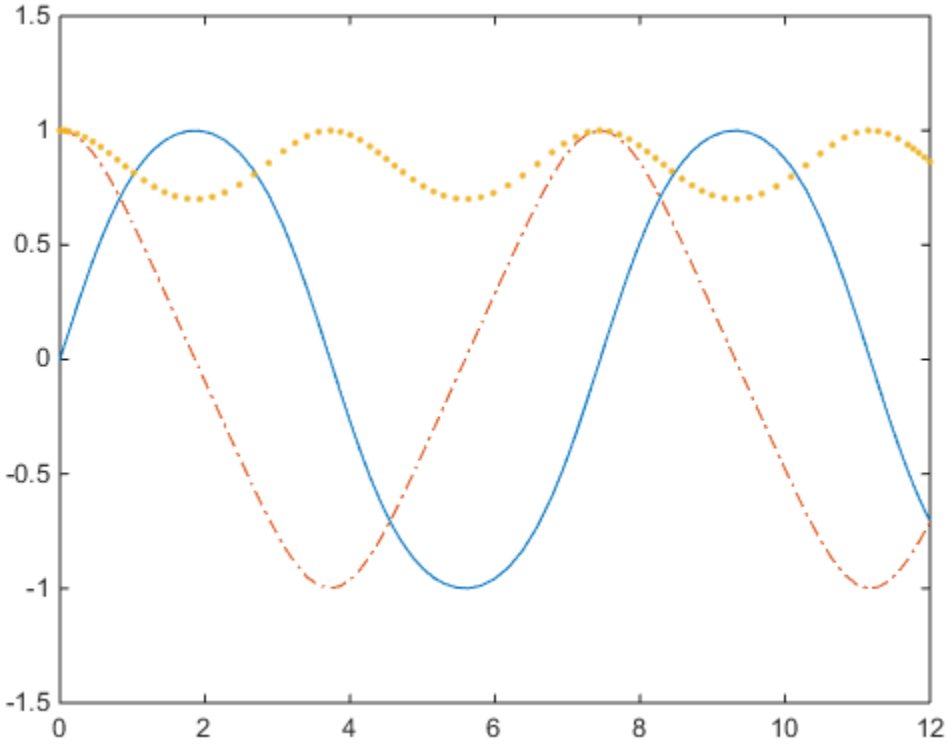
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

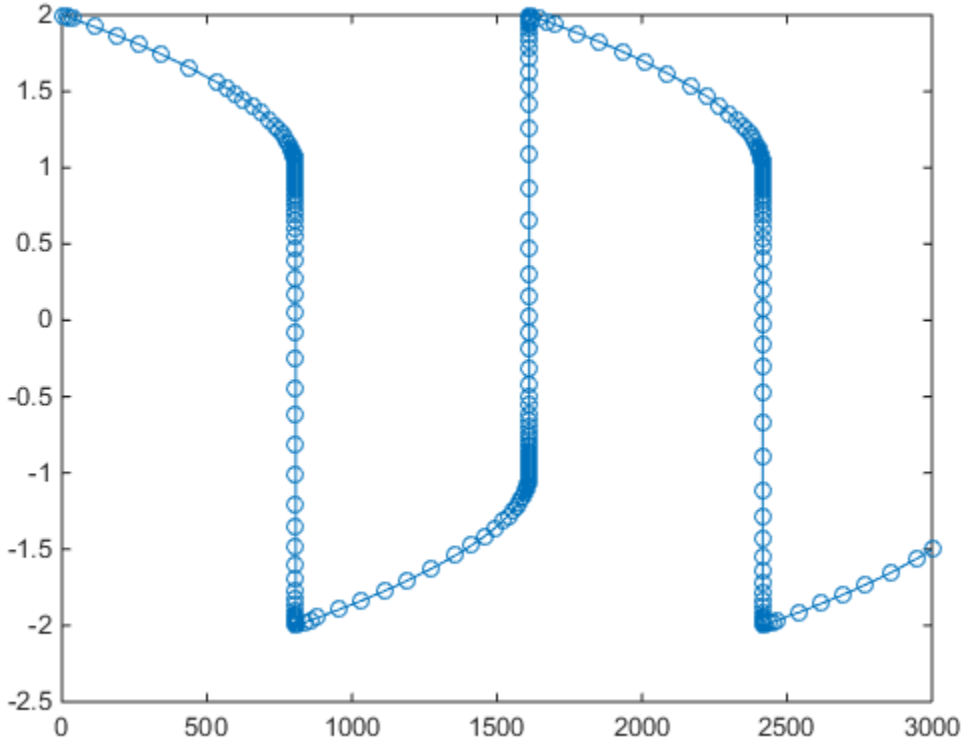
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

`y'(t) + f(t)y(t) = g(t)`
The initial condition is `y(1) = 1`, where the function `f(t)` is defined by the `n`-by-`1` vector `f` evaluated at times `ft`, and the function `g(t)` is defined by the `m`-by-`1` vector `g` evaluated at times `gt`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

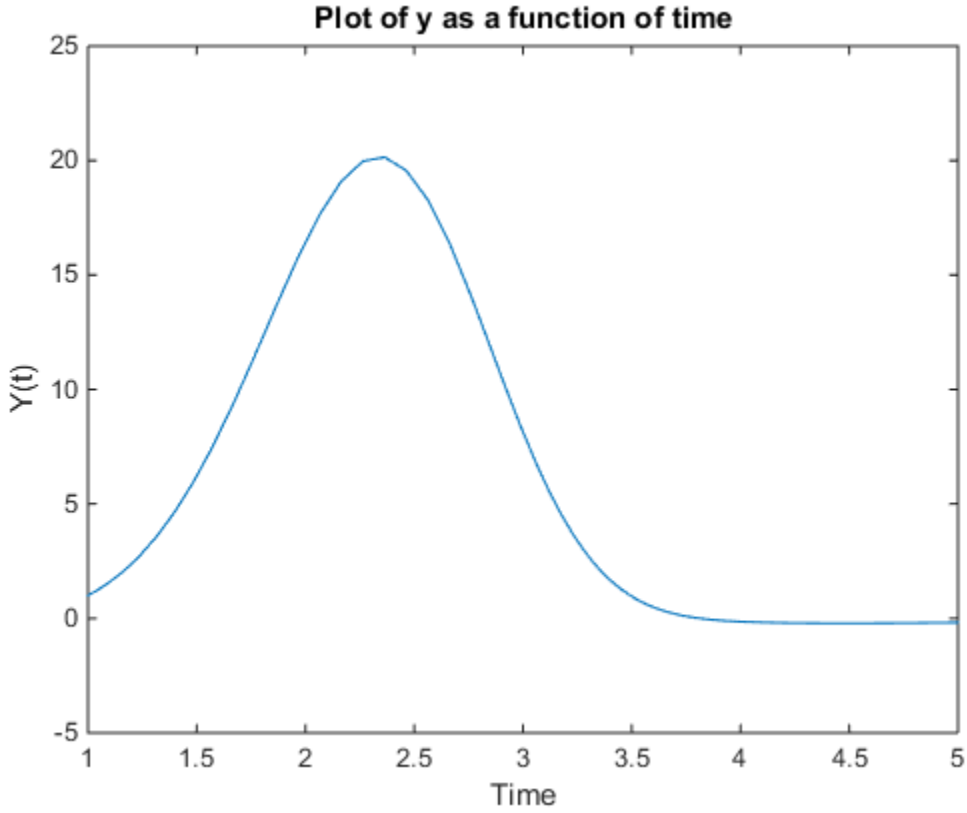
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=1) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time');
ylabel('Y(t)');
```



More About

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, “Transient Simulation of Silicon Devices and Circuits,” *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

ode45

Solve nonstiff differential equations; medium order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.
`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first

input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.

`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T, [Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2),...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.
- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-5614 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

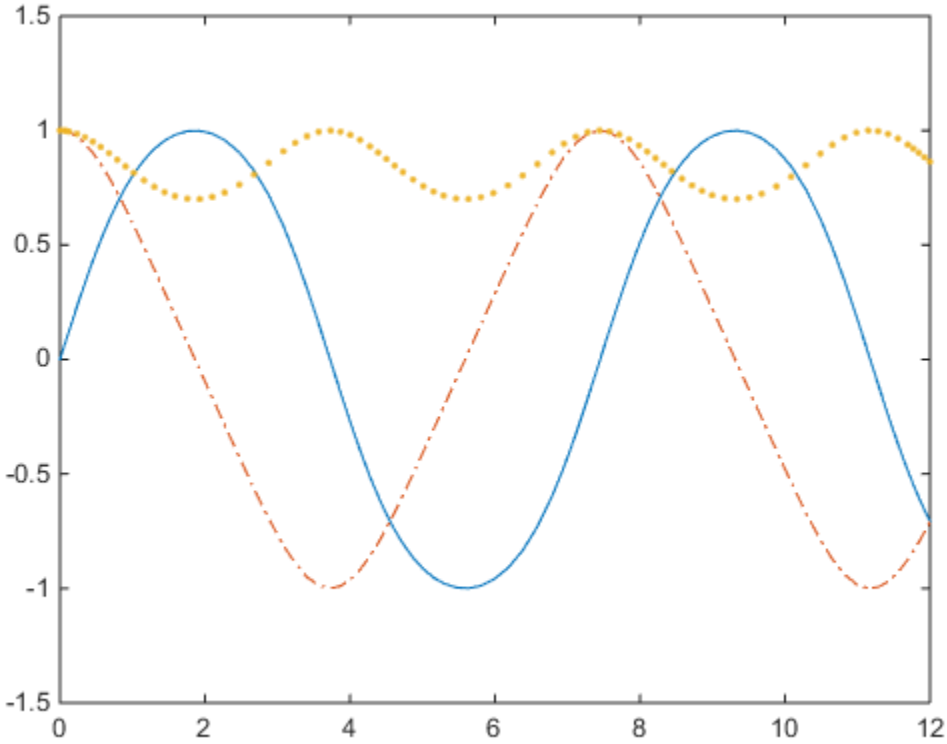
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array Y versus T shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

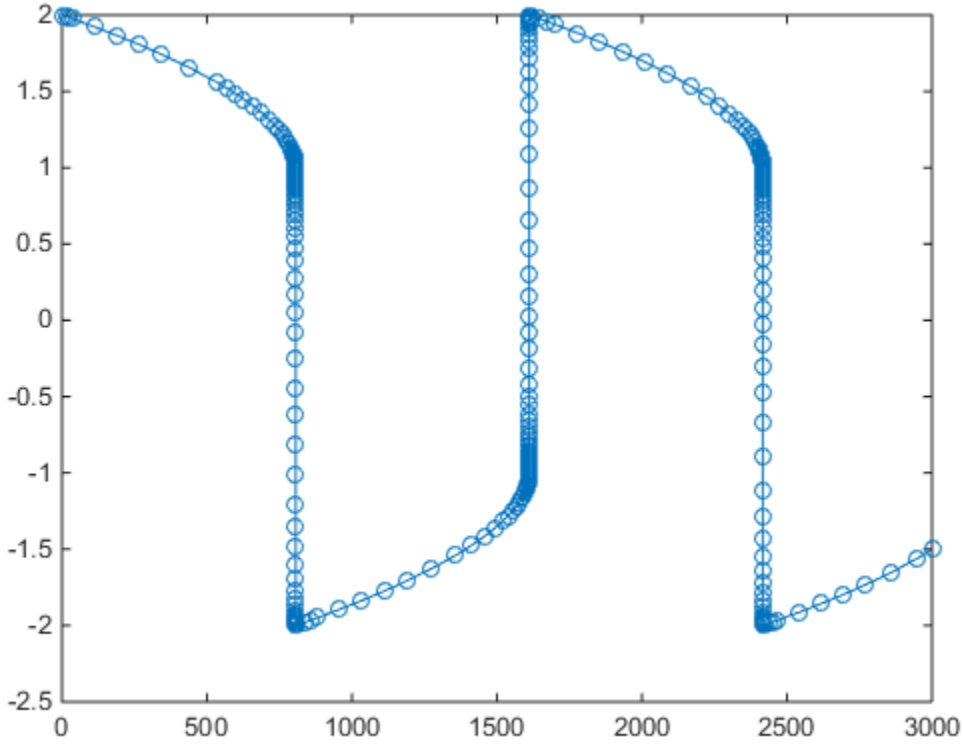
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

`y'(t) + f(t)y(t) = g(t)`
The initial condition is `y(1) = 1`, where the function `f(t)` is defined by the `n`-by-`1` vector `f` evaluated at times `ft`, and the function `g(t)` is defined by the `m`-by-`1` vector `g` evaluated at times `gt`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

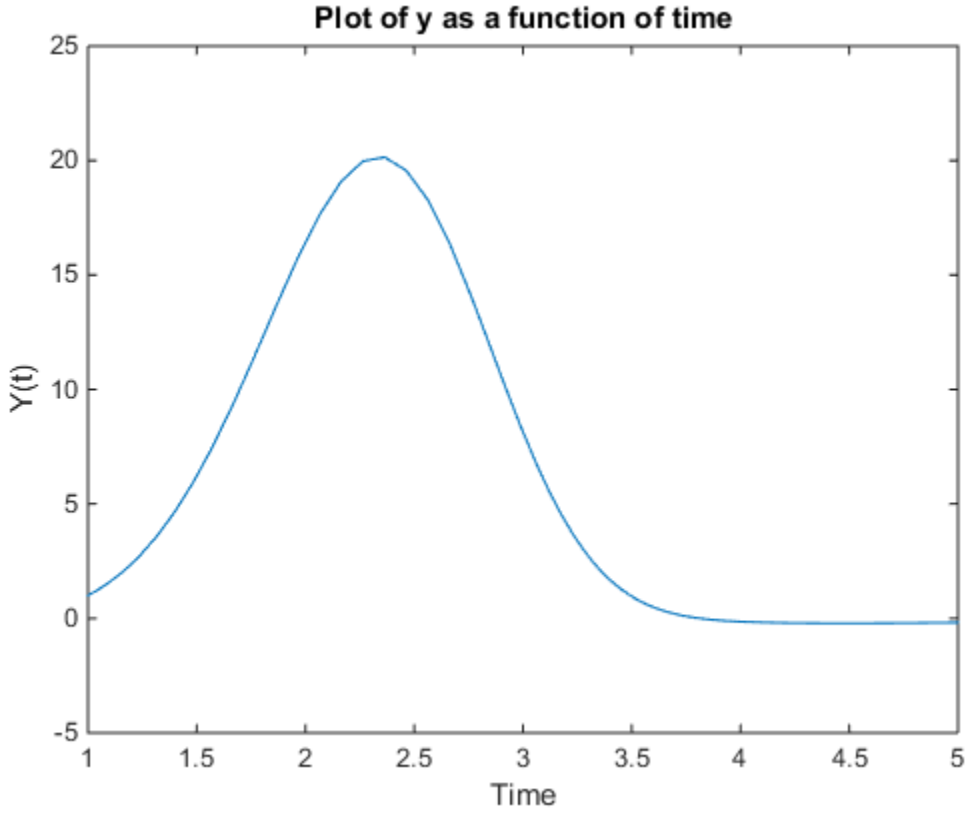
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=1) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time');
ylabel('Y(t)');
```



More About

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, “Transient Simulation of Silicon Devices and Circuits,” *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

ode113

Solve nonstiff differential equations; variable order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.
`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first

input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.

`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T, [Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2),...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.
- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-5628 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

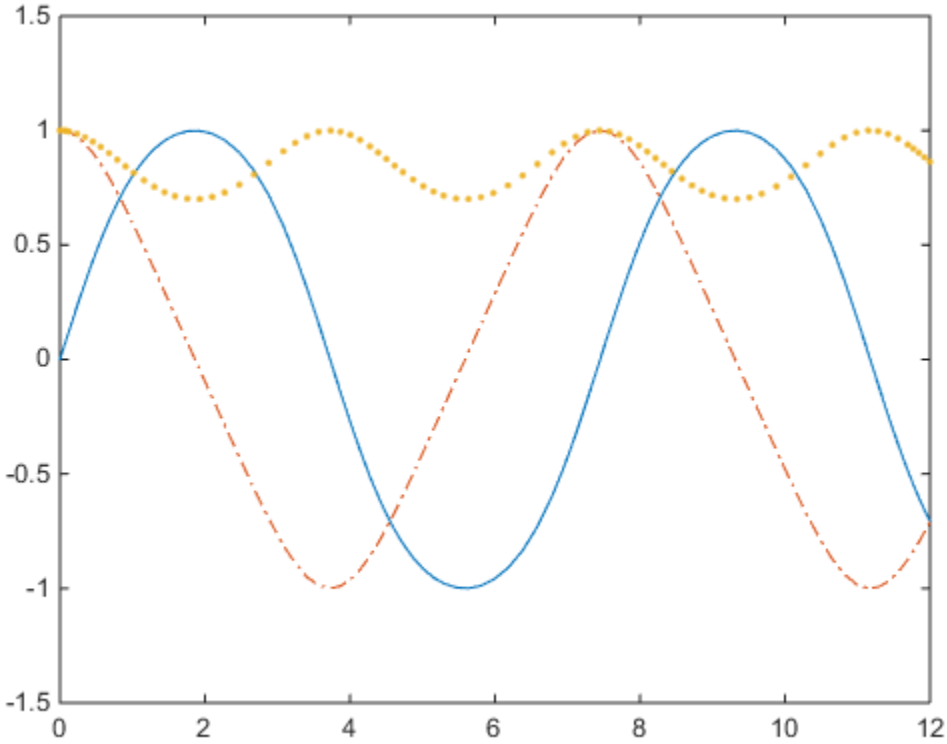
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array Y versus T shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

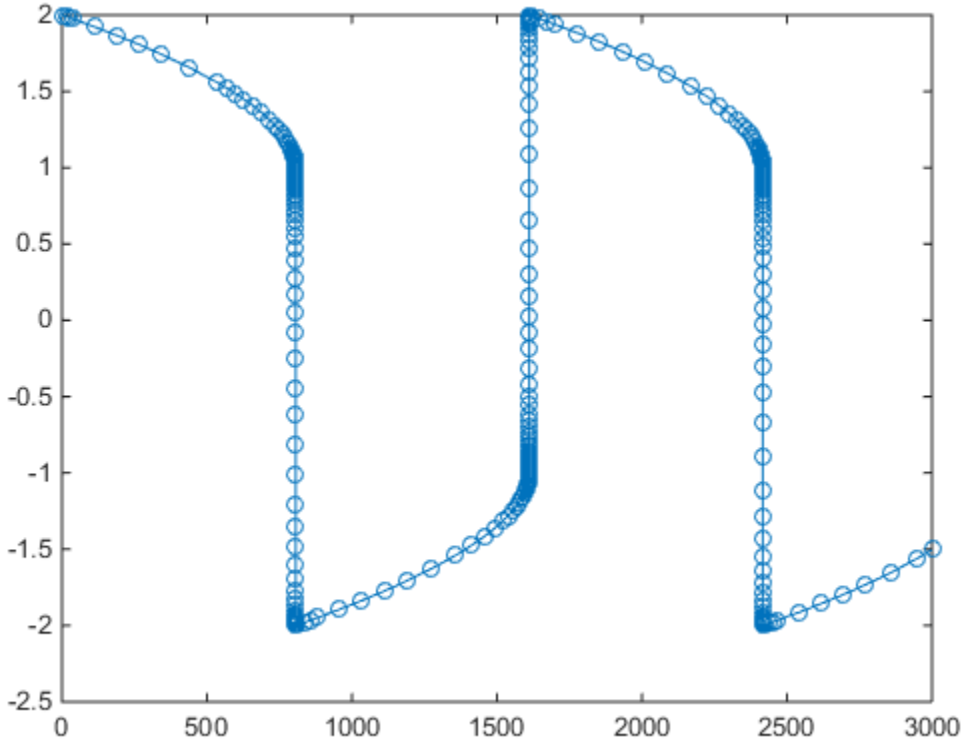
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

`y'(t) + f(t)y(t) = g(t)`
The initial condition is `y(1) = 1`, where the function `f(t)` is defined by the `n`-by-`1` vector `f` evaluated at times `ft`, and the function `g(t)` is defined by the `m`-by-`1` vector `g` evaluated at times `gt`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

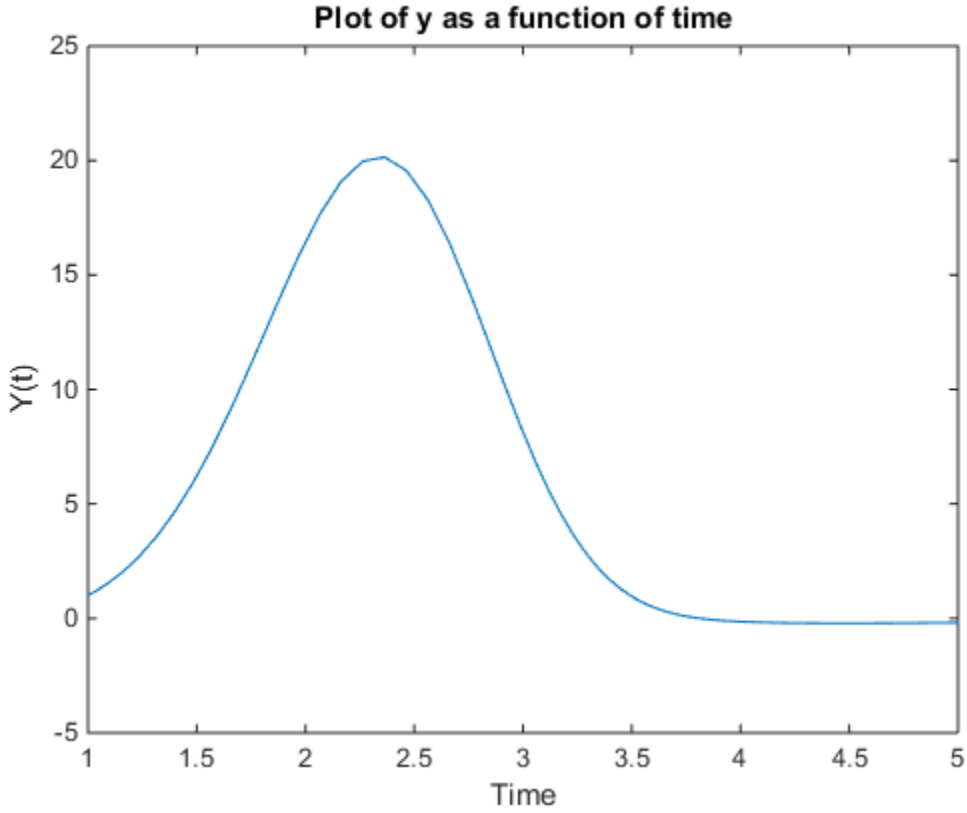
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=1) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time');
ylabel('Y(t)');
```



More About

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, “Transient Simulation of Silicon Devices and Circuits,” *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Introduced before R2006a

odeget

Ordinary differential equation options parameters

Syntax

```
o = odeget(options,'name')
o = odeget(options,'name',default)
```

Description

`o = odeget(options,'name')` extracts the value of the property specified by string `'name'` from integrator options structure `options`, returning an empty matrix if the property value is not specified in `options`. It is only necessary to type the leading characters that uniquely identify the property name. Case is ignored for property names.

`o = odeget(options,'name',default)` returns `o = default` if the named property is not specified in `options`.

The empty matrix `[]` is a valid `options` argument, and the syntax `odeget([],'name',default)` always returns `default`.

Examples

Having constructed an ODE options structure,

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-3 2e-3 3e-3]);
```

you can view these property settings with `odeget`.

```
odeget(options,'RelTol')
ans =

       1.0000e-04

odeget(options,'AbsTol')
ans =
```

```
    0.0010    0.0020    0.0030

odeget([],'RelTol',1e-3)
ans =

   1.0000e-03
```

See Also

odeset

Introduced before R2006a

odeset

Create or alter options structure for ordinary differential equation solvers

Syntax

```
options = odeset('name1',value1,'name2',value2,...)
options = odeset(oldopts,'name1',value1,...)
options = odeset(oldopts,newopts)
odeset
```

Description

The `odeset` function lets you adjust the integration parameters of the following ODE solvers.

For solving fully implicit differential equations:
`ode15i`

For solving initial value problems:
`ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb`

See below for information about the integration parameters.

`options = odeset('name1',value1,'name2',value2,...)` creates an options structure that you can pass as an argument to any of the ODE solvers. In the resulting structure, `options`, the named properties have the specified values. For example, `'name1'` has the value `value1`. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify a property name. Case is ignored for property names.

`options = odeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This sets `options` equal to the existing structure `oldopts`, overwrites any values in `oldopts` that are respecified using name/value pairs, and adds any new pairs to the structure. The modified structure is returned as an output argument.

`options = odeset(oldopts,newopts)` alters an existing options structure `oldopts` by combining it with a new options structure `newopts`. Any new options not equal to the empty matrix overwrite corresponding options in `oldopts`.

`odeset` with no input arguments displays all property names as well as their possible and default values.

ODE Properties

The following sections describe the properties that you can set using `odeset`. The available properties depend on the ODE solver you are using. There are several categories of properties:

Note This reference page describes the ODE properties for MATLAB, Version 7. The Version 5 properties are supported only for backward compatibility. For information on the Version 5 properties, type at the MATLAB command line: `more on, type odeset, more off`.

Error Control Properties

At each step, the solver estimates the local error `e` in the `i`th component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.
`|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))`

For routine problems, the ODE solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require

tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For the absolute error tolerance, the scaling of the solution components is important: if `|y|` is somewhat smaller than `AbsTol`, the solver is not constrained to obtain any correct digits in `y`. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.

The following table describes the error control properties. Further information on each property is given following the table.

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	Relative error tolerance that applies to all components of the solution vector `y`.
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the individual components of the solution vector.
`NormControl`	`on` | {`off`}	Control error relative to norm of solution.

Description of Error Control Properties

`RelTol` — This tolerance is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components, except those smaller than thresholds `AbsTol(i)`.

The default, `1e-3`, corresponds to 0.1% accuracy.

`AbsTol` — `AbsTol(i)` is a threshold below which the value of the `i`th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero.

If `AbsTol` is a vector, the length of `AbsTol` must be the same as the length of the solution vector `y`. If `AbsTol` is a scalar, the value applies to all components of `y`.

`NormControl` — Set this property `on` to request that the solvers control the error in each integration step with `norm(e) <= max(RelTol*norm(y),AbsTol)`. By default the solvers use a more stringent componentwise error control.

Solver Output Properties

The following table lists the solver output properties that control the output that the solvers generate. Further information on each property is given following the table.

Property	Value	Description
`NonNegative`	Vector of integers	Specifies which components of the solution vector must be nonnegative. The default value is `[]`.
`OutputFcn`	Function handle	A function for the solver to call after every successful integration step.
`OutputSel`	Vector of indices	Specifies which components of the solution vector are to be passed to the output function.
`Refine`	Positive integer	Increases the number of output points by a factor of `Refine`.
`Stats`	`on` | {`off`}	Determines whether the solver should display statistics about its computations. By default, `Stats` is `off`.

Description of Solver Output Properties

`NonNegative` — The `NonNegative` property is not available in `ode23s`, `ode15i`. In `ode15s`, `ode23t`, and `ode23tb`, `NonNegative` is not available for problems where there is a mass matrix.

`OutputFcn` — To specify an output function, set `'OutputFcn'` to a function handle. For example,

```
options = odeset('OutputFcn',@myfun)
```

sets `'OutputFcn'` to `@myfun`, a handle to the function `myfun`. See the `function_handle` reference page for more information.

The output function must be of the form

```
status = myfun(t,y,flag)
```

“Parameterizing Functions” explains how to provide additional parameters to `myfun`, if necessary.

The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:

Flag	Description
`init`	The solver calls `myfun(tspan,y0,'init')` before beginning the integration to allow the output function to initialize. `tspan` and `y0` are the input arguments to the ODE solver.
`{[]}`	The solver calls `status = myfun(t,y,[])` after each integration step on which output is requested. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points in `t`. If `t` is a vector, the `i`th column of `y` corresponds to the `i`th element of `t`. When `length(tspan) > 2` the output is produced at every point in `tspan`. When `length(tspan) = 2` the output is produced according to the `Refine` option. `myfun` must return a `status` output value of `0` or `1`. If `status = 1`, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.
`done`	The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores.

You can use these general purpose output functions or you can edit them to create your own. Type `help function` at the command line for more information.

- `odeplot` — Time series plotting (default when you call the solver with no output arguments and you have not specified an output function)
- `odephas2` — Two-dimensional phase plane plotting
- `odephas3` — Three-dimensional phase plane plotting
- `odeprint` — Print solution as it is computed

Note If you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history.

`OutputSel` — Use `OutputSel` to specify which components of the solution vector you want passed to the output function. For example, if you want to use the `odeplot` output

function, but you want to plot only the first and third components of the solution, you can do this using

```
options = ...
odeset('OutputFcn',@odeplot,'OutputSel',[1 3]);
```

By default, the solver passes all components of the solution to the output function.

`Refine` — If `Refine` is `1`, the solver returns solutions only at the end of each time step. If `Refine` is `n` >`1`, the solver subdivides each time step into `n` smaller intervals and returns solutions at each time point. `Refine` does not apply when `length(tspan)>2` or the ODE solver returns the solution as a structure.

Note In all the solvers, the default value of `Refine` is `1`. Within `ode45`, however, the default is `4` to compensate for the solver's large step sizes. To override this and see only the time steps chosen by `ode45`, set `Refine` to `1`.

The extra values produced for `Refine` are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

`Stats` — By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays

- Number of successful steps
- Number of failed attempts
- Number of times the ODE function was called to evaluate $f(t,y)$

Solvers based on implicit methods, including `ode23s`, `ode23t`, `ode15s`, and `ode15i`, also display

- Number of times that the partial derivatives matrix $\partial f/\partial x$ was formed
- Number of LU decompositions
- Number of solutions of linear systems

Step-Size Properties

The step-size properties specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step-size properties. Further information on each property is given following the table.

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size.
`MaxStep`	Positive scalar {`0.1*abs(t0-tf)`}	Upper bound on solver step size.

Description of Step-Size Properties

`InitialStep` — `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the initial step size is based on the slope of the solution at the initial time `tspan(1)`, and if the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.

`MaxStep` — If the differential equation has periodic coefficients or solutions, it might be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep` for any of the following purposes:

- To produce more output points. This can significantly slow down solution time. Instead, use `Refine` to compute additional outputs by continuous extension at very low cost.
- When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. See “Error Control Properties” on page 1-5634 for a description of the error tolerance properties.
- To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs,

break the simulation interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth. While solving a problem, the ODE solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property. Further information on each property is given following the table.

ODE Events Property

String	Value	Description
`Events`	Function handle	Handle to a function that includes one or more event functions.

Description of Event Location Properties

`Events` — The function is of the form

```
[value,isterminal,direction] = events(t,y)
```

`value`, `isterminal`, and `direction` are vectors for which the `i`th element corresponds to the `i`th event function:

- `value(i)` is the value of the `i`th event function.
- `isterminal(i)` = `1` if the integration is to terminate at a zero of this event function, otherwise, `0`.
- `direction(i) = 0` if all zeros are to be located (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.

If you specify an events function and events are detected, the solver returns three additional outputs:

- `A` column vector of times at which events occur

- Solution values corresponding to these times
- Indices into the vector returned by the events function. The values indicate which event the solver detected.

If you call the solver as

```
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `TE`, `YE`, and `IE` respectively. If you call the solver as

```
sol = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `sol.xe`, `sol.ye`, and `sol.ie`, respectively.

For examples that use an event function, see "Event Location" and "Advanced Event Location" in the MATLAB Mathematics documentation.

Jacobian Matrix Properties

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix $\partial f/\partial y$, a matrix of partial derivatives of the function that defines the differential equations.

$$\frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots \\ \vdots & \vdots & \\ & & \end{bmatrix}$$

The Jacobian matrix properties pertain only to those solvers for stiff problems (`ode15s`, `ode23s`, `ode23t`, `ode23tb`, and `ode15i`) for which the Jacobian matrix $\partial f/\partial y$ can be critical to reliability and efficiency. If you do not provide a function to calculate the Jacobian, these solvers approximate the Jacobian numerically using finite differences. In this case, you might want to use the `Vectorized` or `JPattern` properties.

The following table describes the Jacobian matrix properties for all implicit solvers except `ode15i`. Further information on each property is given following the table. See Jacobian Properties for ode15i for `ode15i`-specific information.

Jacobian Properties for All Implicit Solvers Except ode15i

Property	Value	Description
`Jacobian`	Function handle | constant matrix	Matrix or function that evaluates the Jacobian.
`JPattern`	Sparse matrix of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Allows the solver to reduce the number of function evaluations required.

Description of Jacobian Properties

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function `FJac`, where `FJac(t,y)` computes $\partial f/\partial y$, or to the constant value of $\partial f/\partial y$.

The Jacobian for the "van der Pol Equation (Stiff)", described in the MATLAB Mathematics documentation, can be coded as

```
function J = vdp1000jac(t,y)
J = [  0                    1
     (-2000*y(1)*y(2)-1)  (1000*(1-y(1)^2)) ];
```

`JPattern` — `JPattern` is a sparsity pattern with `1`s where there might be nonzero entries in the Jacobian.

Note If you specify `Jacobian`, the solver ignores any setting for `JPattern`.

Set this property to a sparse matrix S with $S(i,j) = 1$ if component i of $f(t,y)$ depends on component j of y, and 0 otherwise. The solver uses this sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large and sparse, this can greatly accelerate execution. For an example using the `JPattern` property, see Example: Large, Stiff, Sparse Problem in the MATLAB Mathematics documentation.

`Vectorized` — The `Vectorized` property allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Set `on` to inform the solver that you have coded the ODE function `F` so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`. This allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Note If you specify `Jacobian`, the solver ignores a setting of `'on'` for `'Vectorized'`.

With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. For example, you can vectorize the "van der Pol Equation (Stiff)", described in the MATLAB Mathematics documentation, by introducing colon notation into the subscripts and by using the array power (`.^`) and array multiplication (`.*`) operators.

```
function dydt = vdp1000(t,y)
dydt = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)];
```

Note Vectorization of the ODE function used by the ODE solvers differs from the vectorization used by the boundary value problem (BVP) solver, `bvp4c`. For the ODE solvers, the ODE function is vectorized only with respect to the second argument, while `bvp4c` requires vectorization with respect to the first and second arguments.

The following table describes the Jacobian matrix properties for `ode15i`.

Jacobian Properties for ode15i

Property	Value	Description
`Jacobian`	Function handle | Cell array of constant values	Function that evaluates the Jacobian or a cell array of constant values.
`JPattern`	Sparse matrices of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Vectorized ODE function

Description of Jacobian Properties for ode15i

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function

```
[dFdy,dFdp] = Fjac(t,y,yp)
```

or to a cell array of constant values [$\partial F/\partial y$,($\partial F/\partial y$)′].

`JPattern` — `JPattern` is a sparsity pattern with `1`'s where there might be nonzero entries in the Jacobian.

Set this property to `{dFdyPattern,dFdypPattern}`, the sparsity patterns of $\partial F/\partial y$ and $\partial F/\partial y'$, respectively.

`Vectorized` —

Set this property to `{yVect, ypVect}`. Setting `yVect` to `'on'` indicates that

```
F(t, [y1 y2 ...], yp)
```

returns

```
[F(t,y1,yp), F(t,y2,yp) ...]
```

Setting `ypVect` to `'on'` indicates that

```
F(t,y,[yp1 yp2 ...])
```

returns

```
[F(t,y,yp1) F(t,y,yp2) ...]
```

Mass Matrix and DAE Properties

This section describes mass matrix and differential-algebraic equation (DAE) properties, which apply to all the solvers except `ode15i`. These properties are not applicable to `ode15i` and their settings do not affect its behavior.

The solvers of the ODE suite can solve ODEs of the form

$$M(t,y)y' = f(t,y)$$

with a mass matrix $M(t,y)$ that can be sparse.

When $M(t,y)$ is nonsingular, the equation above is equivalent to $y' = M^{-1} f(t,y)$ and the ODE has a solution for any initial values y_0 at t_0. The more general form (Equation 1-3)

is convenient when you express a model naturally in terms of a mass matrix. For large, sparse $M(t,y)$, solving Equation 1-3 directly reduces the storage and run-time needed to solve the problem.

When $M(t,y)$ is singular, then $M(t,y)$ times $M(t,y)y' = f(t,y)$ is a DAE. A DAE has a solution only when y_0 is consistent; that is, there exists an initial slope yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. If y_0 and yp_0 are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The `ode15s` and `ode23t` solvers can solve DAEs of index 1. For examples of DAE problems, see Example: Differential-Algebraic Problem, in the MATLAB Mathematics documentation, and the examples amp1dae and hb1dae.

The following table describes the mass matrix and DAE properties. Further information on each property is given following the table.

Mass Matrix and DAE Properties (Solvers Other Than ode15i)

Property	Value	Description
`Mass`	Matrix | function handle	Mass matrix or a function that evaluates the mass matrix $M(t,y)$.
`MStateDependence`	`none | {weak} | strong`	Dependence of the mass matrix on y.
`MvPattern`	Sparse matrix	$\partial(M(t,y)v)/\partial y$ sparsity pattern.
`MassSingular`	`yes | no | {maybe}`	Indicates whether the mass matrix is singular.
`InitialSlope`	Vector {zero vector}	Vector representing the consistent initial slope yp_0.

Description of Mass Matrix and DAE Properties

`Mass` — For problems of the form $M(t)y' = f(t,y)$, set `'Mass'` to a mass matrix M.

For problems of the form $M(t)y' = f(t,y)$, set `'Mass'` to a function handle `@Mfun`, where `Mfun(t,y)` evaluates the mass matrix $M(t,y)$. The `ode23s` solver can only solve problems with a constant mass matrix M. When solving DAEs, using `ode15s` or `ode23t`, it is advantageous to formulate the problem so that M is a diagonal matrix (a semiexplicit DAE).

For example problems, see "Finite Element Discretization" in the MATLAB Mathematics documentation, or the examples `fem2ode` or `batonode`.

`MStateDependence` — Set this property to `none` for problems $M(t)y' = f(t, y)$. Both `weak` and `strong` indicate $M(t,y)$, but `weak` results in implicit solvers using approximations when solving algebraic equations.

`MvPattern` — Set this property to a sparse matrix S with $S(i,j) = 1$ if, for any k, the (i,k) component of $M(t,y)$ depends on component j of y, and 0 otherwise. For use with the `ode15s`, `ode23t`, and `ode23tb` solvers when `MStateDependence` is `strong`. See burgersode as an example.

`MassSingular` — Set this property to `no` if the mass matrix is not singular and you are using either the `ode15s` or `ode23t` solver. The default value of `maybe` causes the solver to test whether the problem is a DAE, by testing whether $M(t_0,y_0)$ is singular.

`InitialSlope` — Vector representing the consistent initial slope yp_0, where yp_0 satisfies $M(t_0, y_0) \cdot yp_0 = f(t_0, y_0)$. The default is the zero vector.

This property is for use with the `ode15s` and `ode23t` solvers when solving DAEs.

ode15s and ode15i-Specific Properties

`ode15s` is a variable-order solver for stiff problems. It is based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear's methods. The `ode15s` properties let you choose among these formulas, as well as specifying the maximum order for the formula used.

`ode15i` solves fully implicit differential equations of the form

$$f(t, y, y') = 0$$

using the variable order BDF method.

The following table describes the `ode15s` and `ode15i`-specific properties. Further information on each property is given following the table. Use `odeset` to set these properties.

ode15s and ode15i-Specific Properties

Property	Value	Description
`MaxOrder`	`1 | 2 | 3 | 4 | {5}`	Maximum order formula used to compute the solution.
`BDF` (`ode15s` only)	`on | {off}`	Specifies whether you want to use the BDFs instead of the default NDFs.

Description of ode15s and ode15i-Specific Properties

`MaxOrder` — Maximum order formula used to compute the solution.

`BDF` (`ode15s` only) — Set `BDF` `on` to have `ode15s` use the BDFs.

For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently if `MaxOrder` is reduced (for example to `2`) so that only the most stable formulas are used.

See Also

`deval` | `odeget` | `ode45` | `ode23` | `ode23t` | `ode23tb` | `ode113` | `ode15s` | `ode23s` | `function_handle`

Introduced before R2006a

odextend

Extend solution of initial value problem for ordinary differential equation

Syntax

```
solext = odextend(sol, odefun, tfinal)
solext = odextend(sol,[],tfinal)
solext = odextend(sol, odefun, tfinal, yinit)
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
solext = odextend(sol, odefun, tfinal, yinit, options)
```

Description

`solext = odextend(sol, odefun, tfinal)` extends the solution stored in `sol` to an interval with upper bound `tfinal` for the independent variable. Specify `odefun` as a function handle. Specify `sol` as an ODE solution structure created using an ODE solver. The lower bound for the independent variable in `solext` is the same as in `sol`. If you created `sol` with an ODE solver other than `ode15i`, the function `odefun` computes the right-hand side of the ODE equation, which is of the form $y' = f(t,y)$. If you created `sol` using `ode15i`, the function `odefun` computes the left-hand side of the ODE equation, which is of the form $f(t,y,y') = 0$.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, if necessary.

`odextend` extends the solution by integrating `odefun` from the upper bound for the independent variable in `sol` to `tfinal`, using the same ODE solver that created `sol`. By default, `odextend` uses

- The initial conditions `y = sol.y(:,end)` for the subsequent integration
- The same integration properties and additional input arguments the ODE solver originally used to compute `sol`. This information is stored as part of the solution structure `sol` and is subsequently passed to `solext`. Unless you want to change these values, you do not need to pass them to `odextend`.

`solext = odextend(sol,[],tfinal)` uses the same ODE function that the ODE solver uses to compute `sol` to extend the solution. It is not necessary to pass in `odefun` explicitly unless it differs from the original ODE function.

`solext = odextend(sol, odefun, tfinal, yinit)` uses the column vector `yinit` as new initial conditions for the subsequent integration, instead of the vector `sol.y(end)`.

Note `solext = odextend(sol, odefun, tfinal, [yinit, ypinit])` extends solutions obtained with `ode15i` by using the column vector `ypinit` to specify the initial derivative of the solution.

`solext = odextend(sol, odefun, tfinal, yinit, options)` uses the integration properties specified in `options` instead of the options the ODE solver originally used to compute `sol`. The new options are then stored within the structure `solext`. See `odeset` for details on setting `options` properties. Set `yinit = []` as a placeholder to specify the default initial conditions.

Examples

The following command

```
sol=ode45(@vdp1,[0 10],[2 0]);
```

uses `ode45` to solve the system `y' = vdp1(t,y)`, where `vdp1` is an example of an ODE function provided with MATLAB software, on the interval `[0 10]`. Then, the commands

```
sol=odextend(sol,@vdp1,20);
plot(sol.x,sol.y(1,:));
```

extend the solution to the interval `[0 20]` and plot the first component of the solution on `[0 20]`.

See Also

`deval` | `ode23` | `ode45` | `ode113` | `ode15s` | `ode23s` | `ode23t` | `ode23tb` | `ode15i` | `odeset` | `odeget` | `deval` | `function_handle`

Introduced before R2006a

onCleanup

Cleanup tasks upon function completion

Syntax

```
cleanupObj = onCleanup(cleanupFun)
```

Description

`cleanupObj = onCleanup(cleanupFun)` creates an object that, when destroyed, executes the function cleanupFun. MATLAB implicitly clears all local variables at the termination of a function, whether by normal completion, or a forced exit, such as an error, or **Ctrl+C**.

If you reference or pass `cleanupObj` outside your function, then cleanupFun does not run when that function terminates. Instead, it runs whenever MATLAB destroys the object.

Examples

Close a Figure After Executing Function

Save the following code in `action.m` and type `action` in the Command Window.

```
function [] = action()
f = figure;
finishup = onCleanup(@() myCleanupFun(f));
disp('Display Figure')
end

function myCleanupFun(f)
close(f)
disp('Close Figure')
end
```

```
Display Figure
```

```
Close Figure
```

Switch Directories After Executing Function

Pass your own script to the `onCleanup` object so that it executes when MATLAB destroys the cleanup object.

Save the following code in `cleanup.m`.

```
cd(tempdir)
disp('You are now in the temporary folder')
```

Save the following code in `youraction.m` and type `youraction` in the Command Window.

```
function [] = youraction
 changeup = onCleanup(@cleanup);
 disp('Execute Code')
end
```

```
Execute Code
You are now in the temporary folder
```

Input Arguments

`cleanupFun` — Clean-up task

function handle

Clean-up task, specified as a handle to a function.

You can declare any number of `onCleanup` objects in a program file. However, if the clean-up tasks depend on the order of execution, then you should define only one object that calls a script or function, containing the relevant clean-up commands.

You should use an anonymous function handle to call your clean-up task. This allows you to pass arguments to your clean-up function.

Example: `@() fclose('file.m')`

Example: `@() user_script`

Example: `@() function(input)`

Data Types: `function_handle`

More About

Tips

- Avoid using nested functions during cleanup. MATLAB can clear variables used in nested functions before the clean-up function tries to read from them.
- If your program contains multiple cleanup objects, MATLAB does not guarantee the order that it destroys these objects.
- If the order of your cleanup functions matters, define one `onCleanup` object for all the tasks.

- "Clean Up When Functions Complete"
- "Object Lifecycle"
- "Function Handles"
- "What Are Anonymous Functions?"

See Also

`clear` | `clearvars` | `function_handle`

ones

Create array of all ones

Syntax

```
X = ones
X = ones(n)
X = ones(sz1,...,szN)
X = ones(sz)

X = ones(classname)
X = ones(n,classname)
X = ones(sz1,...,szN,classname)
X = ones(sz,classname)

X = ones('like',p)
X = ones(n,'like',p)
X = ones(sz1,...,szN,'like',p)
X = ones(sz,'like',p)
```

Description

`X = ones` returns the scalar `1`.

`X = ones(n)` returns an `n`-by-`n` matrix of ones.

`X = ones(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of ones where `sz1,...,szN` indicates the size of each dimension. For example, `ones(2,3)` returns a 2-by-3 array of ones.

`X = ones(sz)` returns an array of ones where the size vector, `sz`, defines `size(X)`. For example, `ones([2,3])` returns a 2-by-3 array of ones.

`X = ones(classname)` returns a scalar `1` where the string, `classname`, specifies the data type. For example, `ones('int8')` returns a scalar, 8-bit integer `1`.

`X = ones(n,classname)` returns an `n`-by-`n` array of ones of data type `classname`.

`X = ones(sz1,...,szN,classname)` returns an `sz1`-by-...-by-`szN` array of ones of data type `classname`.

`X = ones(sz,classname)` returns an array of ones where the size vector, `sz`, defines `size(X)` and `classname` defines `class(X)`.

`X = ones('like',p)` returns a scalar `1` with the same data type, sparsity, and complexity (real or complex) as the numeric variable, `p`.

`X = ones(n,'like',p)` returns an `n`-by-`n` array of ones like `p`.

`X = ones(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of ones like `p`.

`X = ones(sz,'like',p)` returns an array of ones like `p` where the size vector, `sz`, defines `size(X)`.

Examples

Square Array of Ones

Create a 4-by-4 array of ones.

```
X = ones(4)
```

```
X =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

3-D Array of Ones

Create a 2-by-3-by-4 array of ones.

```
X = ones(2,3,4);
```

```
size(X)
```

```
ans =
```

```
     2     3     4
```

Size Defined by Existing Array

Define a 3-by-2 array A.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)
```

```
sz =

     3     2
```

Create an array of ones that is the same size as A

```
X = ones(sz)
```

```
X =

     1     1
     1     1
     1     1
```

Nondefault Numeric Data Type

Create a 1-by-3 vector of ones whose elements are 16-bit unsigned integers.

```
X = ones(1,3,'uint16'),
class(X)
```

```
X =

      1      1      1


ans =

uint16
```

Complex One

Create a scalar 1 that is not real valued, but instead is complex like an existing array.

Define a complex vector.

```
p = [1+2i 3i];
```

Create a scalar 1 that is complex like p.

```
X = ones('like',p)
```

```
X =

   1.0000 + 0.0000i
```

Size and Numeric Data Type of Defined by Existing Array

Define a 2-by-3 array of 8-bit unsigned integers.

```
p = uint8([1 3 5 ; 2 4 6]);
```

Create an array of ones that is the same size and data type as p.

```
X = ones(size(p),'like',p),
class(X)
```

```
X =

     1     1     1
     1     1     1


ans =

uint8
```

Input Arguments

n — Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If n is 0, then X is an empty matrix.
- If n is negative, then it is treated as 0.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values, defines `X` as a sz1-by...-by-szN array.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `X`, does not include those dimensions.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `X`, does not include those dimensions.

Example: `sz = [2,3,4]` defines `X` as a 2-by-3-by-4 array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`classname` — Output class

`'double'` (default) | `'single'` | `'int8'` | `'uint8'` | ...

Output class, specified as `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Data Types: `char`

`p` — Prototype

numeric variable

Prototype, specified as a numeric variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

More About

- “Class Support for Array-Creation Functions”
- “ Preallocating Arrays”

See Also

`complex` | `eye` | `false` | `rand` | `randn` | `size` | `zeros`

Introduced before R2006a

open

Open file in appropriate application

Syntax

```
open(name)
output = open(name)
```

Description

`open(name)` opens the specified file or variable in the appropriate application.

`output = open(name)` returns an empty `output` (`[]`) for most cases. If opening a MAT-file, `output` is a structure that contains the variables in the file. If opening a figure, `output` is a handle to that figure.

Input Arguments

`name`

Name of file or variable to open. If `name` does not include an extension, the `open` function:

1. Searches for a variable named `name`. If the variable exists, `open` opens it in the Variables editor.
2. Searches the MATLAB path for `name.mdl`, `name.slx`, or `name.m`. If `name.mdl` or `name.slx` exists, then `open` opens the model in Simulink. If only `name.m` exists, `open` opens the file in the MATLAB Editor.

If more than one file named `name` exists on the MATLAB path, the `open` function opens the file returned by `which(name)`.

The `open` function performs the following actions based on the file extension:

`.m`	Open in MATLAB Editor.

`.mat`	Return variables in structure *`st`* when called with the syntax: `st = open(name)`
`.fig`	Open figure in Handle Graphics.
`.mdl` or `.slx`	Open model in Simulink.
`.prj`	Open project in the MATLAB Compiler Deployment Tool.
`.doc*`	Open document in Microsoft Word.
`.exe`	Run executable file (only on Windows systems).
`.pdf`	Open document in Adobe® Acrobat®.
`.ppt*`	Open document in Microsoft PowerPoint.
`.xls*`	Start MATLAB Import Wizard.
`.htm` or `.html`	Open document in MATLAB browser.
`.url`	Open file in your default Web browser.

Examples

Open `Contents.m` in the MATLAB Editor by typing:

```
open Contents.m
```

Generally, MATLAB opens *`matlabroot`*`\toolbox\matlab\general\Contents.m`. However, if you have a file called `Contents.m` in a directory that is before `toolbox\matlab\general` on the MATLAB path, then `open` opens that file instead.

Open a file not on the MATLAB path by including the complete file specification:

```
open('D:\temp\data.mat')
```

If the file does not exist, MATLAB displays an error message.

Create a function called `opentxt` to handle files with extension `.txt`:

```
function opentxt(filename)

   fprintf('You have requested file: %s\n', filename);

   wh = which(filename);
```

```
    if exist(filename, 'file') == 2
      fprintf('Opening in MATLAB Editor: %s\n', filename);
      edit(filename);
    elseif ~isempty(wh)
      fprintf('Opening in MATLAB Editor: %s\n', wh);
      edit(wh);
    else
      warning('MATLAB:fileNotFound', ...
              'File was not found: %s', filename);
    end

end
```

Open the file `ngc6543a.txt` (a description of `ngc6543a.jpg`, located in *`matlabroot`*`\toolbox\matlab\demos`):

```
photo_text = 'ngc6543a.txt';
open(photo_text)
```

`open` calls your function with the following syntax:

```
opentxt(photo_text)
```

More About

Tips

The `open` function opens files based on their extension. You can extend the functionality of `open` by defining your own file handling function of the form `open`*`xxx`*, where *`xxx`* is a file extension. For example, if you create a function `openlog`, the `open` function calls `openlog` to process any files with the `.log` extension. The `open` function returns any single *`output`* defined by your function.

See Also

`edit` | `load` | `openfig` | `openvar` | `path` | `uiopen` | `which` | `winopen`

Introduced before R2006a

open

Class: VideoWriter

Open file for writing video data

Syntax

```
open(writerObj)
```

Description

`open(writerObj)` opens the file associated with writerObj for writing. When you open the file, all properties of the object become read only. `open` discards any existing contents of the file.

Input Arguments

`writerObj`

`VideoWriter` object created by the `VideoWriter` function.

Examples

Open a new AVI file:

```
myObj = VideoWriter('newfile.avi');
open(myObj);
```

See Also

`close` | `writeVideo` | `VideoWriter`

openfig

Open figure saved in FIG-file

Syntax

```
openfig(filename)
openfig(filename,copies)
openfig(___,visibility)

fig = openfig(___)
```

Description

`openfig(filename)` opens the figure saved in the MATLAB figure file (FIG-file) called `filename`.

`openfig(filename,copies)` specifies whether to open a new copy of the figure in the case that a copy is already open. If you do not want to create a new copy, set `copies` to `'reuse'`. The `'reuse'` option brings the existing figure to the front of the screen. To open a new copy of the figure regardless of whether a copy is already open, set `copies` to `'new'` . The `'new'` option is the default behavior.

`openfig(___,visibility)` specifies whether to open the figure in a visible or invisible state. To display the figure, set `visibility` to `'visible'`. If you do not want to display the figure, use the `'invisible'` setting. You can use this option with any of the input argument combinations in the previous syntaxes.

`fig = openfig(___)` returns the figure object. Set properties of the figure object to modify its appearance or behavior. For a list of properties, see Figure Properties.

Examples

Open Figure Saved in MATLAB Figure File

Create a surface plot and save the figure as a MATLAB figure file. Then, close the figure.

```
surf(peaks)
savefig('MySavedPlot.fig')
close(gcf)
```

Open the saved figure.

```
openfig('MySavedPlot.fig')
```

Open Invisible Figure in Visible State

Create a surface plot and make the figure invisible. Then, save the figure as a MATLAB figure file. Close the invisible figure.

```
surf(peaks)
set(gcf,'Visible','off')
savefig('MySavedPlot.fig')
close(gcf)
```

Open the saved figure and make it visible on the screen.

```
openfig('MySavedPlot.fig','visible')
```

Input Arguments

`filename` — File name of saved figure

string

File name of saved figure, specified as a string. You do not have to specify the full file path, as long as it is on your MATLAB path. Including `.fig` in the file name is optional.

Example: `openfig('MySavedFigure.fig')`

`copies` — Control for opening multiple copies of figure

`'new'` (default) | `'reuse'`

Control for opening multiple copies of the figure, specified as one of these values:

- `'new'` — Open a new copy of the figure, even if a copy already exists on the screen.
- `'reuse'` — Open a new copy of the figure only if one does not exist. If a copy exists, then bring the existing copy to the front of the screen. If the figure is off the screen, then `'reuse'` repositions the figure so that it is completely on the screen. This option helps provide compatibility with different screen sizes and resolutions by ensuring that the figure displays on screen.

Example: `openfig('MySavedFigure.fig','reuse')`

`visibility` — Figure visibility
`'visible' | 'invisible'`

Figure visibility, specified as one of these values:

- `'visible'` — Open the saved figure in a visible state. If the MATLAB figure file contains an invisible figure, then you can use this option to make the figure visible when it opens.
- `'invisible'` — Open the saved figure in an invisible state.

Example: `openfig('MySavedFigure.fig','invisible')`

Output Arguments

`fig` — Figure object
figure object

Figure object. Set properties of the figure to change the appearance or behavior of the opened figure. For a list of properties, see Figure Properties.

Limitations

- Do not use `openfig` to open FIG-files created with GUIDE. Use the `guide` function instead.

More About

- "Save Figure to Reopen in MATLAB Later"

See Also
`open` | `saveas` | `savefig`

Introduced before R2006a

opengl

Control OpenGL rendering

Compatibility

The `autoselect`, `neverselect`, `advise`, `quiet`, `verbose`, and `DriverBugWorkaround` inputs have been removed in R2014b. For more information about the behavior of these syntaxes in previous releases, see opengl for R2014a.

Syntax

```
opengl info
d = opengl('data')

opengl software
opengl hardware

opengl('save','software')
opengl('save','hardware')
opengl('save','none')
```

Description

`opengl info` prints information about the OpenGL library on your system, such as the version, vendor, and graphics features that it supports. The returned fields might change in future releases as graphics features change. Using this command loads the OpenGL library.

`d = opengl('data')` returns the same data provided with `opengl info`, but stores it in a structure.

`opengl software` uses the software version of OpenGL built into MATLAB to render subsequent graphics. This command works only on Windows systems. To use software OpenGL on Linux systems, start MATLAB with the `-softwareopengl` flag. Macintosh systems do not support software OpenGL.

`opengl hardware` uses the version of OpenGL on your operating system to render subsequent graphics. Use this option if you have graphics hardware that supports hardware-accelerated OpenGL. This command works only on Windows systems. To use hardware OpenGL on Linux systems, start MATLAB with the `-nosoftwareopengl` flag.

`opengl('save','software')` sets your preferences so that future sessions of MATLAB on this computer use software OpenGL. This command does not affect the current session.

`opengl('save','hardware')` sets your preferences to the version of OpenGL on your operating system. This command does not affect the current session.

`opengl('save','none')` resets your preferences to the default values. This command does not affect the current session.

Examples

Display Information About OpenGL

Display information about the OpenGL library on your system, such as the vendor, the version, and the MATLAB graphics features that it supports. Also display whether MATLAB is using a hardware-accelerated implementation or a software implementation of OpenGL.

```
opengl info
```

```
                         Version: '3.3.0'
                          Vendor: 'NVIDIA Corporation'
                        Renderer: 'Quadro 400/PCIe/SSE2'
                  MaxTextureSize: 8192
                          Visual: 'Visual 0x07, (RGBA 32 bits (8 8 8 8), ...'
                        Software: 'false'
      SupportsGraphicsSmoothing: 1
  SupportsDepthPeelTransparency: 1
     SupportsAlignVertexCenters: 1
                     Extensions: {251x1 cell}
             MaxFrameBufferSize: 8192
```

Determine Graphics Hardware

Determine your graphics hardware by checking the `Vendor` and `Renderer` fields returned by the `opengl info` command.

```
opengl info
```

```
                          Version: '3.3.0'
                          Vendor: 'NVIDIA Corporation'
                        Renderer: 'Quadro 400/PCI/SSE2'
                 MaxTextureSize: 8192
                          Visual: 'Visual 0x07, (RGBA 32 bits (8 8 8 8), Z depth 24 b
                        Software: 'false'
       SupportsGraphicsSmoothing: 1
   SupportsDepthPeelTransparency: 1
      SupportsAlignVertexCenters: 1
                     Extensions: {235x1 cell}
             MaxFrameBufferSize: 8192
```

If you are using software OpenGL, then the name listed in the `Vendor` field is not your graphics hardware vendor. The line `Software: 'true'` indicates that you are using software OpenGL.

If you are using software OpenGL on Windows systems, determine your graphics hardware by closing all figures, switching to hardware OpenGL, and reissuing the `opengl info` command. Then, switch back to software OpenGL.

```
close all
opengl hardware
opengl info
opengl software
```

If you are using software OpenGL on Linux systems, determine your graphics hardware by starting MATLAB with the `-nosoftwareopengl` flag. Then, issue the `opengl info` command.

Mac OS X systems do not support software OpenGL, so the `Vendor` field always lists the name of your graphics hardware vendor.

Use Software OpenGL for Current Session

Switch to using software OpenGL to render graphics in the current session.

```
opengl software
```

This command works only on Windows systems.

Use Software OpenGL for Future Sessions

Set your preferences so that MATLAB uses software OpenGL to render graphics in all future sessions. This command does not affect the current session.

```
opengl('save','software')
```

More About

Tips

- By default, MATLAB uses your graphics hardware if it supports hardware-accelerated OpenGL. If it is not available or if you are using a virtual machine, a Linux VNC, or an unsupported graphics driver, then MATLAB uses software OpenGL instead. For more information on graphics renderers, see the figure `Renderer` property.
- For the best results with graphics, upgrade your graphics drivers to the latest version available. For more information, see "System Requirements for Graphics".

- "System Requirements for Graphics"
- "Resolving Low-Level Graphics Issues"

See Also

`matlablinux` | `matlabmac` | `matlabwindows`

Introduced before R2006a

openvar

Open workspace variable in Variables editor or other graphical editing tool

Syntax

```
openvar(varname)
```

Description

`openvar(varname)` opens the workspace variable named by the string, `varname`, in the Variables editor for graphical editing. Changes that you make to variables in the Variables editor occur in the workspace as soon as you enter them.

In some toolboxes, `openvar` opens a tool appropriate for viewing or editing objects indicated by `varname` instead of opening the Variables editor.

MATLAB does not impose any limitation on the size of a variable that you can open in the Variables editor. However, your operating system or the amount of physical memory installed on your computer can impose such limits.

Examples

Identify Outliers in a Linked Graph

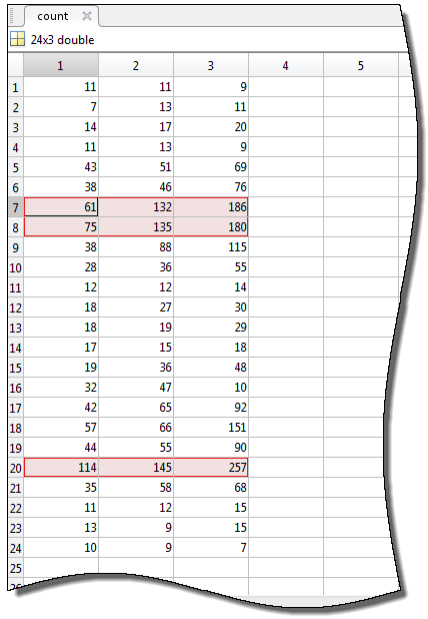
Use data brushing to identify observations in a vector or matrix that might warrant further analysis.

Make a scatter plot of data in the sample MAT-file `count.dat`, and open the variable `count` in the Variables editor.

```
load count.dat
scatter(count(:,1),count(:,2))
openvar('count')
```

Right-click a cell in the Variables editor and select **Brushing** > **Brushing On**. This turns on data brushing in the Variables editor.

Select the rows 7, 8, and 20. (Select noncontiguous rows by holding down the **Ctrl** key and clicking in each row.)

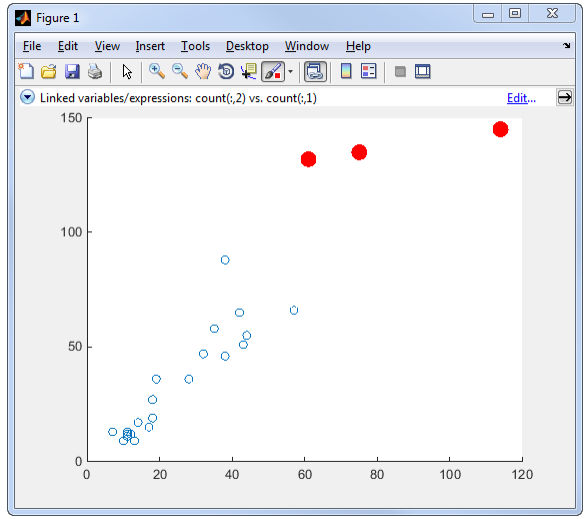

count

24x3 double

	1	2	3	4	5
1	11	11	9		
2	7	13	11		
3	14	17	20		
4	11	13	9		
5	43	51	69		
6	38	46	76		
7	61	132	186		
8	75	135	180		
9	38	88	115		
10	28	36	55		
11	12	12	14		
12	18	27	30		
13	18	19	29		
14	17	15	18		
15	19	36	48		
16	32	47	10		
17	42	65	92		
18	57	66	151		
19	44	55	90		
20	114	145	257		
21	35	58	68		
22	11	12	15		
23	13	9	15		
24	10	9	7		
25					

In the Figure window with the scatter plot, click **Brush/Select Data**  to enable data brushing, and **Link Plot**  to enable data linking.

The data observations you brushed in the Variables editor appear highlighted in the scatter plot.



As long as data linking is enabled in the figure, observations that you brush in the scatter plot are highlighted in the Variables editor. When a figure is not linked to its data sources, you can still brush its graphs and you can brush the same data in the Variables editor, but only the display that you brush responds by highlighting.

Input Arguments

`varname` — Variable name

string

Variable name, specified as a string. The named variable can be an array, character string, cell array, structure, or an object and its properties. If the named variable is a multidimensional array, then you can only view the array in the Variables editor, and not edit it.

Example: `'myVariable'`

Example: `'A'`

More About

Tips

- As an alternative to the `openvar` function, double-click a variable in the Workspace browser.

- "View, Edit, and Copy Variables"
- "Making Graphs Responsive with Data Linking"

See Also

`brush` | `linkdata` | `load` | `save` | `workspace`

Introduced before R2006a

optimget

Optimization options values

Syntax

```
val = optimget(options,'param')
val = optimget(options,'param',default)
```

Description

`val = optimget(options,'param')` returns the value of the specified parameter in the optimization options structure `options`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = optimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`. Note that this form of the function is used primarily by other optimization functions.

Examples

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options`.

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'`.

```
optnew = optimget(my_options,'Display','final');
```

See Also

`optimset` | `fminbnd` | `fminsearch` | `fzero` | `lsqnonneg`

Introduced before R2006a

optimset

Create or edit optimization options structure

Syntax

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,'param1',value1,...)
options = optimset(oldopts,newopts)
```

Description

The function `optimset` creates an `options` structure that you can pass as an input argument to the following four MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`
- `lsqnonneg`

You can use the `options` structure to change the default parameters for these functions.

Note If you have an Optimization Toolbox license, you can also use `optimset` to create an expanded `options` structure containing additional options specifically designed for the functions provided in that toolbox. For more information about these additional options, see the reference page for the enhanced Optimization Toolbox `optimset` function.

`options = optimset('param1',value1,'param2',value2,...)` creates an optimization options structure called `options`, in which the specified parameters (`param`) have specified values. Any unspecified parameters are set to `[]` (parameters with value `[]` indicate to use the default value for that parameter when `options`

is passed to the optimization function). It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`optimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all fields are set to `[]`.

`options = optimset(optimfun)` creates an options structure `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = optimset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the available options for the MATLAB optimization functions.

Option	Value	Description	Solvers
`Display`	`'off'` | `'iter'` | {`'final'`} | `'notify'`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration (not available for `lsqnonneg`); `'final'` displays just the final output; `'notify'` displays output only if the function does not converge.	`fminbnd`, `fminsearch`,`fzero`, `lsqnonneg`
`FunValChec`	{`'off'`} | `'on'`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no error.	`fminbnd`, `fminsearch`,`fzero`

Option	Value	Description	Solvers
`MaxFunEval`	positive integer	Maximum number of function evaluations allowed.	`fminbnd`, `fminsearch`
`MaxIter`	positive integer	Maximum number of iterations allowed.	`fminbnd`, `fminsearch`
`OutputFcn`	function | {[ ]}	User-defined function that an optimization function calls at each iteration. See “Output Functions”.	`fminbnd`, `fminsearch`,`fzero`
`PlotFcns`	function | {[ ]}	User-defined or built-in plot function that an optimization function calls at each iteration. Built-in functions: • `@optimplotx` plots the current point • `@optimplotfval` plots the function value • `@optimplotfunccount` plots the function count (not available for `fzero`) See “Plot Functions”.	`fminbnd`, `fminsearch`,`fzero`
`TolFun`	positive scalar	Termination tolerance on the function value. See “Tolerances and Stopping Criteria”.	`fminsearch`
`TolX`	positive scalar	Termination tolerance on *x*, the current point. See “Tolerances and Stopping Criteria”.	`fminbnd`, `fminsearch`,`fzero`, `lsqnonneg`

Examples

This statement creates an optimization options structure called `options` in which the `Display` parameter is set to `'iter'` and the `TolFun` parameter is set to `1e-8`.

```
options = optimset('Display','iter','TolFun',1e-8)
```

This statement makes a copy of the options structure called `options`, changing the value of the `TolX` parameter and storing new values in `optnew`.

```
optnew = optimset(options,'TolX',1e-4);
```

This statement returns an optimization options structure that contains all the parameter names and default values relevant to the function `fminbnd`.

```
optimset('fminbnd')
```

See Also

`optimset` | `optimget` | `fminbnd` | `fminsearch` | `fzero` | `lsqnonneg`

Introduced before R2006a

or, |

Find logical OR

Syntax

```
A | B | ...
or(A, B)
```

Description

`A | B | ...` performs a logical OR of all input arrays `A`, `B`, etc., and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if any input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input `A` is a 3-by-5 matrix and input `B` is the number 1, then `B` is treated as if it were a 3-by-5 matrix of ones.

`or(A, B)` is called for the syntax `A | B` when either `A` or `B` is an object.

Note The symbols | and || perform different operations in a MATLAB application. The element-wise OR operator described here is |. The short-circuit OR operator is ||.

Examples

If matrix `A` is

```
    0.4235    0.5798         0    0.7942         0
    0.5155         0         0         0    0.8744
```

```
         0         0         0    0.4451    0.0150
    0.4329    0.6405    0.6808         0         0
```

and matrix `B` is

```
     0     1     0     1     0
     1     1     0     0     1
     0     0     0     1     0
     0     1     0     0     1
```

then

```
A | B
ans =
     1     1     0     1     0
     1     1     0     0     1
     0     0     0     1     1
     1     1     1     0     1
```

More About

- "Truth Table for Logical Operations"

See Also

`all` | `and` | `any` | `bitor` | `Logical Operators: Short Circuit` | `not` | `xor`

Introduced before R2006a

ordeig

Eigenvalues of quasitriangular matrices

Syntax

```
E = ordeig(T)
E = ordeig(AA,BB)
```

Description

`E = ordeig(T)` takes a quasitriangular Schur matrix `T`, typically produced by `schur`, and returns the vector `E` of eigenvalues in their order of appearance down the diagonal of T.

`E = ordeig(AA,BB)` takes a quasitriangular matrix pair `AA` and `BB`, typically produced by `qz`, and returns the generalized eigenvalues in their order of appearance down the diagonal of `AA-λ*BB`.

`ordeig` is an order-preserving version of `eig` for use with `ordschur` and `ordqz`. It is also faster than `eig` for quasitriangular matrices.

Examples

Example 1

```
T=diag([1 -1 3 -5 2]);
```

`ordeig(T)` returns the eigenvalues of `T` in the same order they appear on the diagonal.

```
ordeig(T)

ans =

     1
    -1
     3
```

```
    -5
     2
```

`eig(T)`, on the other hand, returns the eigenvalues in order of increasing magnitude.

```
eig(T)

ans =

    -5
    -1
     1
     2
     3
```

Example 2

```
A = rand(10);
[U, T] = schur(A);
abs(ordeig(T))

ans =

    5.3786
    0.7564
    0.7564
    0.7802
    0.7080
    0.7080
    0.5855
    0.5855
    0.1445
    0.0812
% Move eigenvalues with magnitude < 0.5 to the
% upper-left corner of T.
[U,T] = ordschur(U,T,abs(E)<0.5);
abs(ordeig(T))

ans =

    0.1445
    0.0812
    5.3786
    0.7564
```

```
    0.7564
    0.7802
    0.7080
    0.7080
    0.5855
    0.5855
```

See Also

`schur` | `qz` | `ordschur` | `ordqz` | `eig`

Introduced before R2006a

orderfields

Order fields of structure array

Syntax

```
s = orderfields(s1)
s = orderfields(s1, s2)
s = orderfields(s1, c)
s = orderfields(s1, perm)
[s, perm] = orderfields(...)
```

Description

`s = orderfields(s1)` orders the fields in `s1` so that the new structure array `s` has field names in ASCII dictionary order.

`s = orderfields(s1, s2)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in `s2`. Structures `sl` and `s2` must have the same fields.

`s = orderfields(s1, c)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in the cell array of field name strings `c`. Structure `s1` and cell array `c` must contain the same field names.

`s = orderfields(s1, perm)` orders the fields in `s1` so that the new structure array `s` has fieldnames in the order specified by the indices in permutation vector `perm`.

If `s1` has `N` fieldnames, the elements of `perm` must be an arrangement of the numbers from `1` to `N`. This is particularly useful if you have more than one structure array that you would like to reorder in the same way.

`[s, perm] = orderfields(...)` returns a permutation vector representing the change in order performed on the fields of the structure array that results in `s`.

Examples

Create a structure `s`. Then create a new structure from `s`, but with the fields ordered alphabetically:

```
s = struct('b', 2, 'c', 3, 'a', 1)
s =
    b: 2
    c: 3
    a: 1

snew = orderfields(s)
snew =
    a: 1
    b: 2
    c: 3
```

Arrange the fields of `s` in the order specified by the second (cell array) argument of `orderfields`. Return the new structure in `snew` and the permutation vector used to create it in `perm`:

```
[snew, perm] = orderfields(s, {'b', 'a', 'c'})
snew =
    b: 2
    a: 1
    c: 3
perm =
     1
     3
     2
```

Now create a new structure, `s2`, having the same fieldnames as `s`. Reorder the fields using the permutation vector returned in the previous operation:

```
s2 = struct('b', 3, 'c', 7, 'a', 4)
s2 =
    b: 3
    c: 7
    a: 4

snew = orderfields(s2, perm)
snew =
    b: 3
    a: 4
```

```
    c: 7
```

More About

Tips

`orderfields` only orders top-level fields. It is not recursive.

See Also

`cell2struct` | `fieldnames` | `getfield` | `isfield` | `rmfield` | `setfield` | `struct` | `struct2cell`

Introduced before R2006a

ordqz

Reorder eigenvalues in QZ factorization

Syntax

```
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)
[...] = ordqz(AA,BB,Q,Z,keyword)
[...] = ordqz(AA,BB,Q,Z,clusters)
```

Description

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)` reorders the QZ factorizations `Q*A*Z = AA` and `Q*B*Z = BB` produced by the `qz` function for a matrix pair `(A,B)`. It returns the reordered pair `(AAS,BBS)` and the cumulative orthogonal transformations `QS` and `ZS` such that `QS*A*ZS = AAS` and `QS*B*ZS = BBS`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular pair `(AAS,BBS)`, and the corresponding invariant subspace is spanned by the leading columns of `ZS`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along the diagonal of `AA-λ*BB`.

Note To extract `E` from `AA` and `BB`, use `ordeig(BB)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `AA-λ*BB`.

`[...] = ordqz(AA,BB,Q,Z,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[...] = ordqz(AA,BB,Q,Z,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices commensurate with `E = ordeig(AA,BB)`, such that all eigenvalues with the same `clusters` value form one cluster, `ordqz` sorts the specified clusters in descending order along the diagonal of `(AAS,BBS)`. The cluster with highest index appears in the upper left corner.

See Also

`ordeig` | `ordschur` | `qz`

Introduced before R2006a

ordschur

Reorder eigenvalues in Schur factorization

Syntax

```
[US,TS] = ordschur(U,T,select)
[US,TS] = ordschur(U,T,keyword)
[US,TS] = ordschur(U,T,clusters)
```

Description

`[US,TS] = ordschur(U,T,select)` reorders the Schur factorization `X = U*T*U'` produced by the `schur` function and returns the reordered Schur matrix `TS` and the cumulative orthogonal transformation `US` such that `X = US*TS*US'`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular Schur matrix `TS`, and the corresponding invariant subspace is spanned by the leading columns of `US`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along `T`'s diagonal.

Note To extract `E` from `T`, use `E = ordeig(T)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `TS`.

`[US,TS] = ordschur(U,T,keyword)` sets the selected cluster to include all eigenvalues in one of the following regions:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[US,TS] = ordschur(U,T,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices, commensurate with `E = ordeig(T)`, and such that all eigenvalues with the same `clusters` value form one cluster, `ordschur` sorts the specified clusters in descending order along the diagonal of `TS`, the cluster with highest index appearing in the upper left corner.

See Also

`ordeig` | `ordqz` | `schur`

Introduced before R2006a

orient

Hardcopy paper orientation

Alternatives

Use **File —> Print Preview** on the figure window menu to directly manipulate print layout, paper size, headers, fonts and other properties when printing figures. For details, see “Print Figure from File Menu” in the MATLAB Graphics documentation.

Syntax

```
orient
orient landscape
orient portrait
orient tall
orient(fig_handle)
orient(simulink_model)
orient(fig_handle,orientation)
orient(simulink_model,orientation)
```

Description

`orient` returns a string with the current paper orientation: `portrait`, `landscape`, or `tall`.

`orient landscape` sets the paper orientation of the current figure to full-page landscape, orienting the longest page dimension horizontally. The figure is centered on the page and scaled to fit the page with a 0.25 inch border.

`orient portrait` sets the paper orientation of the current figure to portrait, orienting the longest page dimension vertically. The `portrait` option returns the page orientation to the MATLAB default. (Note that the result of using the `portrait` option is affected by changes you make to figure properties. See the "Algorithm" section for more specific information.)

`orient tall` maps the current figure to the entire page in portrait orientation, leaving a 0.25 inch border.

`orient(fig_handle)` returns the current orientation of the specified figure.

`orient(simulink_model)` returns the current orientation of the Simulink model.

`orient(fig_handle,`*`orientation`*`)` sets the orientation for the specified figure to the specified orientation (`landscape`, `portrait`, or `tall`).

`orient(simulink_model,`*`orientation`*`)` sets the orientation for the Simulink model.

More About

Algorithms

`orient` sets the `PaperOrientation`, `PaperPosition`, and `PaperUnits` properties of the current figure. Subsequent print operations use these properties. The result of using the `portrait` option can be affected by default property values as follows:

- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the current values of `PaperOrientation` and `PaperPosition` to place the figure on the page.
- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the default figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.
- If the current figure `PaperType` is different from the default figure `PaperType`, then the `orient portrait` command uses the current figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.

See Also

`print` | `printpreview` | `set`

Introduced before R2006a

orth

Orthonormal basis for range of matrix

Syntax

```
Q = orth(A)
```

Description

`Q = orth(A)` returns an orthonormal basis for the range of `A`. The columns of `Q` are vectors, which span the range of `A`. The number of columns in `Q` is equal to the rank of `A`.

Examples

Basis for Full Rank Matrix

Calculate and verify the orthonormal basis vectors for the range of a full rank matrix.

Define a matrix and find the rank.

```
A = [1 0 1;-1 -2 0; 0 1 -1];
r = rank(A)
```

```
r =

     3
```

Since `A` is a square matrix of full rank, the orthonormal basis calculated by `orth(A)` matches the matrix `U` calculated in the singular value decomposition, `[U,S] = svd(A,'econ')`. This is because the singular values of `A` are all nonzero.

Calculate the orthonormal basis for the range of `A` using `orth`.

```
Q = orth(A)
```

```
Q =
```

```
   -0.1200   -0.8097    0.5744
    0.9018    0.1531    0.4042
   -0.4153    0.5665    0.7118
```

The number of columns in `Q` is equal to `rank(A)`. Since `A` is of full rank, `Q` and `A` are the same size.

Verify that the basis, `Q`, is orthogonal and normalized within a reasonable error range.

```
E = norm(eye(r)-Q'*Q,'fro')
```

```
E =

   9.6228e-16
```

The error is on the order of `eps`.

Basis for Rank Deficient Matrix

Calculate and verify the orthonormal basis vectors for the range of a rank deficient matrix.

Define a singular matrix and find the rank.

```
A = [1 0 1; 0 1 0; 1 0 1];
r = rank(A)
```

```
r =

     2
```

Since `A` is rank deficient, the orthonormal basis calculated by `orth(A)` matches only the first `r = 2` columns of matrix `U` calculated in the singular value decomposition, `[U,S] = svd(A,'econ')`. This is because the singular values of `A` are *not* all nonzero.

Calculate the orthonormal basis for the range of `A` using `orth`.

```
Q = orth(A)
```

```
Q =

   -0.7071         0
         0    1.0000
   -0.7071         0
```

Since A is rank deficient, Q contains one fewer column than A.

Input Arguments

A — Input matrix

scalar | vector | matrix

Input matrix, specified as a scalar, vector, or matrix.

Data Types: single | double
Complex Number Support: Yes

More About

Range

The column space, or *range*, of a matrix A is the collection of all linear combinations of the columns of A. Any vector, b, that is a solution to the linear equation, A*x = b, is included in the range of A since you can also write it as a linear combination of the columns of A.

Rank

The rank of a matrix is equal to the dimension of the range.

Algorithms

orth is obtained from U in the singular value decomposition, [U,S] = svd(A,'econ'). If r = rank(A), the first r columns of U form an orthonormal basis for the range of A.

See Also

null | rank | svd

Introduced before R2006a

outerjoin

Outer join between two tables

Syntax

```
C = outerjoin(A,B)
C = outerjoin(A,B,Name,Value)
[C,ia,ib] = outerjoin(___)
```

Description

`C = outerjoin(A,B)` creates the table, `C`, as the outer join between the tables `A` and `B` by matching up rows using all the variables with the same name as key variables.

The outer join includes the rows that match between `A` and `B`, and also unmatched rows from either `A` or `B`, all with respect to the key variables. `C` contains all variables from both `A` and `B`, including the key variables.

`C = outerjoin(A,B,Name,Value)` performs the outer-join operation with additional options specified by one or more Name,Value pair arguments.

`[C,ia,ib] = outerjoin(___)` also returns index vectors, ia and ib, indicating the correspondence between rows in `C` and those in `A` and `B` respectively. You can use this syntax with any of the input arguments in the previous syntaxes.

Examples

Outer-Join Operation of Tables with One Variable in Common

Create a table, `A`.

```
A = table([5;12;23;2;15;6],...
    {'cheerios';'pizza';'salmon';'oreos';'lobster';'pizza'},...
    'VariableNames',{'Age','FavoriteFood'},...
    'RowNames',{'Amy','Bobby','Holly','Harry','Marty','Sally'})
```

```
A =

              Age    FavoriteFood
              ___    ____________

    Amy        5     'cheerios'
    Bobby     12     'pizza'
    Holly     23     'salmon'
    Harry      2     'oreos'
    Marty     15     'lobster'
    Sally      6     'pizza'
```

Create a table, `B`, with one variable in common with `A`, called `FavoriteFood`.

```
B = table({'cheerios';'oreos';'pizza';'salmon';'cake'},...
    [110;160;140;367;243],...
    {'A-';'D';'B';'B';'C-'},...
    'VariableNames',{'FavoriteFood','Calories','NutritionGrade'})
```

```
B =

    FavoriteFood    Calories    NutritionGrade
    ____________    ________    ______________

    'cheerios'      110         'A-'
    'oreos'         160         'D'
    'pizza'         140         'B'
    'salmon'        367         'B'
    'cake'          243         'C-'
```

Use the `outerjoin` function to create a new table, `C`, with data from tables `A` and `B`.

```
C = outerjoin(A,B)
```

```
C =

    Age    FavoriteFood_A    FavoriteFood_B    Calories    NutritionGrade
    ___    ______________    ______________    ________    ______________

    NaN    ''                'cake'            243         'C-'
      5    'cheerios'        'cheerios'        110         'A-'
     15    'lobster'         ''                NaN         ''
      2    'oreos'           'oreos'           160         'D'
     12    'pizza'           'pizza'           140         'B'
      6    'pizza'           'pizza'           140         'B'
```

```
    23    'salmon'          'salmon'          367          'B'
```

Table C contains a separate variable for the key variable from A, called FavoriteFood_A, and the key variable from B, called FavoriteFood_B.

Merge Key Variable Pair to Single Variable

Create a table, A.

```
A = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
    'VariableNames',{'Key1' 'Var1'})
```

```
A =

    Key1    Var1
    ____    ____

    'a'      1
    'b'      2
    'c'      3
    'e'     11
    'h'     17
```

Create a table, B, with common values in the variable Key1 between tables A and B, but also containing rows with values of Key1 not present in A.

```
B = table({'a','b','d','e'}',[4;5;6;7],...
    'VariableNames',{'Key1' 'Var2'})
```

```
B =

    Key1    Var2
    ____    ____

    'a'     4
    'b'     5
    'd'     6
    'e'     7
```

Use the outerjoin function to create a new table, C, with data from tables A and B. Merge the key values into a single variable in the output table, C.

```
C = outerjoin(A,B,'MergeKeys',true)
```

```
C =
```

```
    Key1    Var1    Var2
    ____    ____    ____

    'a'       1       4
    'b'       2       5
    'c'       3     NaN
    'd'     NaN       6
    'e'      11       7
    'h'      17     NaN
```

Variables in table C that came from A contain null values in the rows that have no match from B. Similarly, variables in C that came from B contain null values in those rows that had no match from A.

Outer-Join Operation of Tables and Indices to Values

Create a table, A.

```
A = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
    'VariableNames',{'Key1' 'Var1'})
```

```
A =

    Key1    Var1
    ____    ____

    'a'       1
    'b'       2
    'c'       3
    'e'      11
    'h'      17
```

Create a table, B, with common values in the variable Key1 between tables A and B, but also containing rows with values of Key1 not present in A.

```
B = table({'a','b','d','e'}',[4;5;6;7],...
    'VariableNames',{'Key1' 'Var2'})
```

```
B =

    Key1    Var2
    ____    ____
```

```
    'a'     4
    'b'     5
    'd'     6
    'e'     7
```

Use the `outerjoin` function to create a new table, `C`, with data from tables `A` and `B`. Match up rows with common values in the key variable, `Key1`, but also retain rows whose key values don't have a match.

Also, return index vectors, `ia` and `ib` indicating the correspondence between rows in `C` and rows in `A` and `B` respectively.

```
[C,ia,ib] = outerjoin(A,B)
```

```
C =

    Key1_A    Var1    Key1_B    Var2
    ______    ____    ______    ____

    'a'         1     'a'         4
    'b'         2     'b'         5
    'c'         3     ''        NaN
    ''        NaN     'd'         6
    'e'        11     'e'         7
    'h'        17     ''        NaN


ia =

     1
     2
     3
     0
     4
     5


ib =

     1
     2
     0
     3
     4
     0
```

The index vectors `ia` and `ib` contain zeros to indicate the rows in table `C` that do not correspond to rows in tables `A` or `B`, respectively.

Left Outer-Join Operation of Tables and Indices to Values

Create a table, `A`.

```
A = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
    'VariableNames',{'Key1' 'Var1'})
```

```
A =

    Key1    Var1
    ____    ____

    'a'      1
    'b'      2
    'c'      3
    'e'     11
    'h'     17
```

Create a table, `B`, with common values in the variable `Key1` between tables `A` and `B`, but also containing rows with values of `Key1` not present in `A`.

```
B = table({'a','b','d','e'}',[4;5;6;7],...
    'VariableNames',{'Key1' 'Var2'})
```

```
B =

    Key1    Var2
    ____    ____

    'a'     4
    'b'     5
    'd'     6
    'e'     7
```

Use the `outerjoin` function to create a new table, `C`, with data from tables `A` and `B`. Ignore rows in `B` whose key values do not match any rows in `A`.

Also, return index vectors, `ia` and `ib` indicating the correspondence between rows in `C` and rows in `A` and `B` respectively.

```
[C,ia,ib] = outerjoin(A,B,'Type','left')
```

```
C =

    Key1_A    Var1    Key1_B    Var2
    ______    ____    ______    ____

    'a'        1      'a'         4
    'b'        2      'b'         5
    'c'        3      ''        NaN
    'e'       11      'e'         7
    'h'       17      ''        NaN


ia =

     1
     2
     3
     4
     5


ib =

     1
     2
     0
     4
     0
```

All values of `ia` are nonzero indicating that all rows in `C` have corresponding rows in `A`.

Input Arguments

`A,B` — Input tables

tables

Input tables, specified as tables.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Keys',2` uses the second variable in `A` and the second variable in `B` as key variables.

`'Keys'` — Variables to use as keys

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys, specified as the comma-separated pair consisting of `'Keys'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You cannot use the `'Keys'` name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

Example: `'Keys',[1 3]` uses the first and third variables in `A` and `B` as a key variables.

`'LeftKeys'` — Variables to use as keys in `A`

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys in `A`, specified as the comma-separated pair consisting of `'LeftKeys'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You must use the `'LeftKeys'` name-value pair argument in conjunction with the `'RightKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `outerjoin` pairs key values based on their order.

Example: `'LeftKeys',1` uses only the first variable in `A` as a key variable.

`'RightKeys'` — Variables to use as keys in `B`

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables to use as keys in `B`, specified as the comma-separated pair consisting of `'RightKeys'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You must use the `'RightKeys'` name-value pair argument in conjunction with the `'LeftKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must

specify the same number of key variables. `outerjoin` pairs key values based on their order.

Example: `'RightKeys',3` uses only the third variable in `B` as a key variable.

`'MergeKeys'` — Merge keys flag

`false` (default) | `true` | `0` | `1`

Merge keys flag, specified as the comma-separated pair consisting of `'MergeKeys'` and either `false`, `true`, `0` or `1`.

`false`	`outerjoin` includes two separate variables in the output table, `C`, for each key variable pair from tables `A` and `B`. This is the default behavior.
`true`	`outerjoin` includes a single variable in the output table, `C`, for each key variable pair from tables `A` and `B`. `outerjoin` creates the single variable by merging the key values from `A` and `B`, taking values from `A` where a corresponding row exists in `A`, and taking values from `B` otherwise. If you specify, `'MergeKeys',true`, then `outerjoin` includes all key variables in the output table, `C`, and overrides the inclusion or exclusion of key variables specified via the `'LeftVariables'` and `'RightVariables'` name-value pair arguments.

`'LeftVariables'` — Variables from `A` to include in `C`

positive integer | vector of positive integers | variable name | cell array containing one or more variable names | logical vector

Variables from `A` to include in `C`, specified as the comma-separated pair consisting of `'LeftVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You can use `'LeftVariables'` to include or exclude key variables as well as nonkey variables from the output, `C`.

By default, `outerjoin` includes all variables from `A`.

`'RightVariables'` — Variables from `B` to include in `C`

positive integer | vector of positive integers | variable name | cell array containing one or more variable names | logical vector

Variables from B to include in C, specified as the comma-separated pair consisting of 'RightVariables' and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

You can use 'RightVariables' to include or exclude key variables as well as nonkey variables from the output, C.

By default, outerjoin includes all the variables from B.

'Type' — Type of outer join operation
'full' (default) | 'left' | 'right'

Type of outer-join operation, specified as the comma-separated pair consisting of 'Type' and either 'full', 'left', or 'right'.

- For a left outer join, C contains rows corresponding to key values in A that do not match any values in B, but not vice-versa.
- For a right outer join, C contains rows corresponding to key values in B that do not match any values in A, but not vice-versa.

By default, outerjoin does a full outer join and includes unmatched rows from both A and B.

Output Arguments

C — Outer join from A and B
table

Outer join from A and B, returned as a table. The output table, C, contains one row for each pair of rows in tables A and B that share the same combination of key values. If A and B contain variables with the same name, outerjoin adds a unique suffix to the corresponding variable names in C. Variables in C that came from A contain null values in those rows that had no match from B. Similarly, variables in C that came from B contain null values in those rows that had no match from A.

In general, if there are m rows in table A and n rows in table B that all contain the same combination of values in the key variables, table C contains m*n rows for that combination. C also contains rows corresponding to key value combinations in one input table that do not match any row the other input table.

C contains the horizontal concatenation of A(ia,LeftVars) and B(ib,RightVars) sorted by the values in the key variables. By default, LeftVars consists of all the variables of A, and RightVars consists of all the from B. Otherwise, LefttVars consists of the variables specified by the 'LeftVariables' name-value pair argument, and RightVars consists of the variables specified by the 'RightVariables' name-value pair argument.

You can store additional metadata such as descriptions, variable units, variable names, and row names in the table. For more information, see Table Properties.

ia — Index to A

column vector

Index to A, returned as a column vector. Each element of ia identifies the row in table A that corresponds to that row in the output table, C. The vector ia contains zeros to indicate the rows in C that do not correspond to rows in A.

ib — Index to B

column vector

Index to B, returned as a column vector. Each element of ib identifies the row in table B that corresponds to that row in the output table, C. The vector ib contains zeros to indicate the rows in C that do not correspond to rows in B.

More About

Key Variable

Variable used to match and combine data between the input tables, A and B.

See Also

innerjoin | join

Introduced in R2013b

pack

Consolidate workspace memory

Syntax

```
pack
pack filename
pack('filename')
```

Description

`pack` frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from your base and global workspaces are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, `[]`).

The MATLAB software temporarily stores your workspace data in a file called `tp######.mat` (where `######` is a numeric value) that is located in your temporary folder. (You can use the command `dir(tempdir)` to see the files in this folder).

`pack filename` frees space in memory, temporarily storing workspace data in a file specified by `filename`. This file resides in your current working folder and, unless specified otherwise, has a `.mat` file extension.

`pack('filename')` is the function form of `pack`.

Examples

Change the current folder to one that is writable, run `pack`, and return to the previous folder.

```
cwd = pwd;
cd(tempdir);
pack
cd(cwd)
```

More About

Tips

You can only run `pack` from the MATLAB command line.

If you specify a `filename` argument, that file must reside in a folder for which you have write permission.

The `pack` function does not affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory.

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the `Out of memory` message from MATLAB, the `pack` function may find you some free memory without forcing you to delete variables.

The `pack` function frees space by

- Saving all variables in the base and global workspaces to a temporary file.
- Clearing all variables and functions from memory.
- Reloading the base and global workspace variables back from the temporary file and then deleting the file.

If you use `pack` and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, you can allocate larger matrices earlier in the MATLAB session and use these system-specific tips:

- When running MATLAB on The Open Group UNIX platforms, ask your system manager to increase your swap space.
- On Microsoft Windows platforms, increase virtual memory using the Windows Control Panel.

To maintain persistent variables when you run pack, use `mlock` in the function.

See Also

`clear` | `memory`

Introduced before R2006a

padecoef

Padé approximation of time delays

Syntax

```
[num,den] = padecoef(T,N)
```

Description

`[num,den] = padecoef(T,N)` returns the `N`th-order Padé approximation of the continuous-time delay T in transfer function form. The row vectors `num` and `den` contain the numerator and denominator coefficients in descending powers of s. Both are `N`th-order polynomials.

Class support for input T:

`float: double, single`

Class Support

Input T support floating-point values of type `single` or `double`.

References

[1] Golub, G. H. and C. F. Van Loan *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore: 1996, pp. 572–574.

See Also

`pade`

pagesetupdlg

Page setup dialog box

Syntax

```
dlg = pagesetupdlg(fig)
```

Note: `pagesetupdlg` is no longer supported. Use `printpreview` instead.

Description

`dlg = pagesetupdlg(fig)` creates a dialog box from which a set of page layout properties for the figure window, `fig`, can be set.

`pagesetupdlg` implements the "Page Setup..." option in the **Figure File Menu**.

`pagesetupdlg` supports setting the layout for a single figure. `fig` must be a single figure handle, not a vector of figures or a Simulink diagram.

See Also

`printdlg` | `printpreview` | `printopt`

Introduced before R2006a

pan

Pan view of graph interactively

Syntax

```
pan on
pan xon
pan yon
pan off
pan
pan(figure_handle,...)
h = pan(figure_handle)
```

Description

`pan on` turns on mouse-based panning in the current figure.

`pan xon` turns on panning only in the *x* direction in the current figure.

`pan yon` turns on panning only in the *y* direction in the current figure.

`pan off` turns panning off in the current figure.

`pan` toggles the pan state in the current figure `on` or `off`.

`pan(figure_handle,...)` sets the pan state in the specified figure.

`h = pan(figure_handle)` returns the figure's pan *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Pan Mode Objects

Access the following properties of pan mode objects.

- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure
- *`Motion`* `'horizontal'|'vertical'|'both'` — The type of panning enabled for the figure

- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set

Pan Mode Callbacks

You can program the following callbacks for pan mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  The application can inhibit the panning operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object clicked on
  % event_obj    event data (empty in this release)
  % res [output] a logical flag to determine whether the pan
  %              operation should take place(for 'res' set to 'false')
  %              or the 'ButtonDownFcn' property of the object should
  %              take precedence (when 'res' is 'true')
  ```

- `ActionPreCallback <function_handle>` — Function to execute before panning

  Set this callback to if you need to execute code when a pan operation begins. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data
  ```

  The event data struct has the following field:

`Axes`	The handle of the axes that is being panned

- `ActionPostCallback <function_handle>` — Function to execute after panning

  Set this callback if you need to execute code when a pan operation ends. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  ```

```
% obj         handle to the figure that has been clicked on
% event_obj   object containing struct of event data
%             (same as the event data of the
%             'ActionPreCallback' callback)
```

Pan Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesPan(h,axes)` — Function querying permission to pan axes

  Calling the function `isAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a pan operation is permitted on the axes objects.
- `setAllowAxesPan(h,axes,flag)` — Function to set permission to pan axes

  Calling the function `setAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a pan operation on the axes objects.
- *`info`* `= getAxesPanMotion(h,axes)` — Function to get style of pan operations

  Calling the function `getAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, as input will return a character cell array of the same dimension as the axes handle vector, which indicates the type of pan operation for each axes. Possible values for the type of operation are `'horizontal'`, `'vertical'` or `'both'`.
- `setAxesPanMotion(h,axes,style)` — Function to set style of pan operations

  Calling the function `setAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, sets the style of panning on each axes.

Examples

Example 1 — Entering Pan Mode

Plot a graph and turn on Pan mode:

```
plot(magic(10));
pan on
```

```
% pan on the plot
```

Example 2 — Constrained Pan

Constrain pan to *x*-axis using `set`:

```
plot(magic(10));
h = pan;
h.Motion = 'horizontal';
h.Enable = 'on';
% pan on the plot in the horizontal direction.
```

Example 3 — Constrained Pan in Subplots

Create four axes as subplots and give each one a different panning behavior:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = pan;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesPan(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesPanMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesPanMotion(h,ax4,'vertical');
% pan on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for pan mode objects to trigger. Copy the following code to a new file, execute it, and observe panning behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
hLine.ButtonDownFcn = 'disp(''This executes'')';
hLine.Tag = 'DoNotIgnore';
h = pan;
h.ButtonDownFilter = @mycallback;
h.Enable = 'on';
```

```
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then
% return true.

% Indicate what the target is.
disp(['Clicked ' obj.Type ' object'])
objTag = obj.Tag;
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-ButtonDown events for pan mode objects to trigger. Copy the following code to a new file, execute it, and observe panning behavior:

```
function demo
% Listen to pan events
plot(1:10);
h = pan;
h.ActionPreCallback = @myprecallback;
h.ActionPostCallback = @mypostcallback;
h.Enable = 'on';
%
function myprecallback(obj,evd)
disp('A pan is about to occur.');
%
function mypostcallback(obj,evd)
newLim = evd.Axes.XLim;
msgbox(sprintf('The new X-Limits are [%.2f,%.2f].',newLim));
```

Example 6 — Creating a Context Menu for Pan Mode

Coding a context menu that lets the user to switch to Zoom mode by right-clicking:

```
figure
plot(magic(10));
hCM = uicontextmenu;
hMenu = uimenu('Parent',hCM,'Label','Switch to zoom',...
```

```
        'Callback','zoom(gcbf,''on'')');
hPan = pan(gcf);
hPan.UIContextMenu = hCM;
pan('on')
```

You cannot add items to the built-in pan context menu, but you can replace it with your own.

Alternatives

Use the Pan tool  on the figure toolbar to enable and disable pan mode on a plot, or select **Pan** from the figure's **Tools** menu. For details, see "Panning — Shifting Your View of the Graph".

More About

Tips

You can create a pan mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

Note: Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a UI that updates a figure's callbacks, the UI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different pan behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

See Also

`zoom` | `linkaxes` | `rotate3d`

Introduced before R2006a

matlab.unittest.parameters

Summary of classes associated with MATLAB Unit Test parameters

Description

The `matlab.unittest.parameters` package consists of the following classes used in parameterized testing.

matlab.unittest.parameters.EmptyParameter
: Empty parameter implementation

matlab.unittest.parameters.ClassSetupParameter
: Specification of Class Setup Parameter

matlab.unittest.parameters.TestParameter
: Specification of Test Parameter

matlab.unittest.parameters.MethodSetupParameter
: Specification of Method Setup Parameter

See Also

`matlab.unittest.TestSuite.selectIf`

Related Examples

- “Create Basic Parameterized Test”
- “Create Advanced Parameterized Test”

matlab.unittest.parameters.EmptyParameter class

Package: matlab.unittest.parameters

Empty parameter implementation

Description

The `matlab.unittest.parameters.EmptyParameter` class is a `Parameter` implementation that provides no parameter information. There is no need for test authors to interact with this `Parameter` directly. This class provides an empty parameter instance to the `Parameterization` property of a nonparameterized test element.

Properties

`Property`

String indicating the name of the property that defines the Empty Parameter.

`Name`

String indicating the name that uniquely identifies a particular value for a Empty Parameter.

`Value`

Value of the Empty Parameter. The `Value` property holds the data that the Test Runner uses for parameterized testing.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`matlab.unittest.TestCase`

Related Examples

- “Create Basic Parameterized Test”
- “Create Advanced Parameterized Test”

Introduced in R2014a

matlab.unittest.parameters.ClassSetupParameter class

Package: matlab.unittest.parameters

Specification of Class Setup Parameter

Description

The `matlab.unittest.parameters.ClassSetupParameter` class holds information about a single value of a Class Setup Parameter.

Properties

`Property`

String indicating the name of the property that defines the Class Setup Parameter.

`Name`

String indicating the name that uniquely identifies a particular value for a Class Setup Parameter.

`Value`

Value of the Class Setup Parameter. The `Value` property holds the data that the Test Runner uses for parameterized testing.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`matlab.unittest.TestCase`

Related Examples

- “Create Basic Parameterized Test”
- “Create Advanced Parameterized Test”

Introduced in R2014a

matlab.unittest.parameters.TestParameter class

Package: matlab.unittest.parameters

Specification of Test Parameter

Description

The `matlab.unittest.parameters.TestParameter` class holds information about a single value of a Test Parameter.

Properties

`Property`

String indicating the name of the property that defines the Test Parameter.

`Name`

String indicating the name that uniquely identifies a particular value for a Test Parameter.

`Value`

Value of the Test Parameter. The `Value` property holds the data that the Test Runner uses for parameterized testing.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`matlab.unittest.TestCase`

Related Examples

- "Create Basic Parameterized Test"

- “Create Advanced Parameterized Test”

Introduced in R2014a

matlab.unittest.parameters.MethodSetupParameter class

Package: matlab.unittest.parameters

Specification of Method Setup Parameter

Description

The `matlab.unittest.parameters.MethodSetupParameter` class holds information about a single value of a Method Setup Parameter.

Properties

`Property`

String indicating the name of the property that defines the Method Setup Parameter.

`Name`

String indicating the name that uniquely identifies a particular value for a Method Setup Parameter.

`Value`

Value of the Method Setup Parameter. The `Value` property holds the data that the Test Runner uses for parameterized testing.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`matlab.unittest.TestCase`

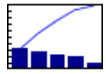
Related Examples

- “Create Basic Parameterized Test”
- “Create Advanced Parameterized Test”

Introduced in R2014a

pareto

Pareto chart



Syntax

```
pareto(Y)
pareto(Y,names)
pareto(Y,X)
H = pareto(...)
```

Description

Pareto charts display the values in the vector `Y` as bars drawn in descending order. Values in `Y` must be nonnegative and not include `NaN`s. Only the first 95% of the cumulative distribution is displayed.

`pareto(Y)` labels each bar with its element index in `Y` and also plots a line displaying the cumulative sum of `Y`.

`pareto(Y,names)` labels each bar with the associated name in the string matrix or cell array `names`.

`pareto(Y,X)` labels each bar with the associated value from `X`.

`pareto(ax,..)` plots into the axes `ax` rather than the current axes, `gca`.

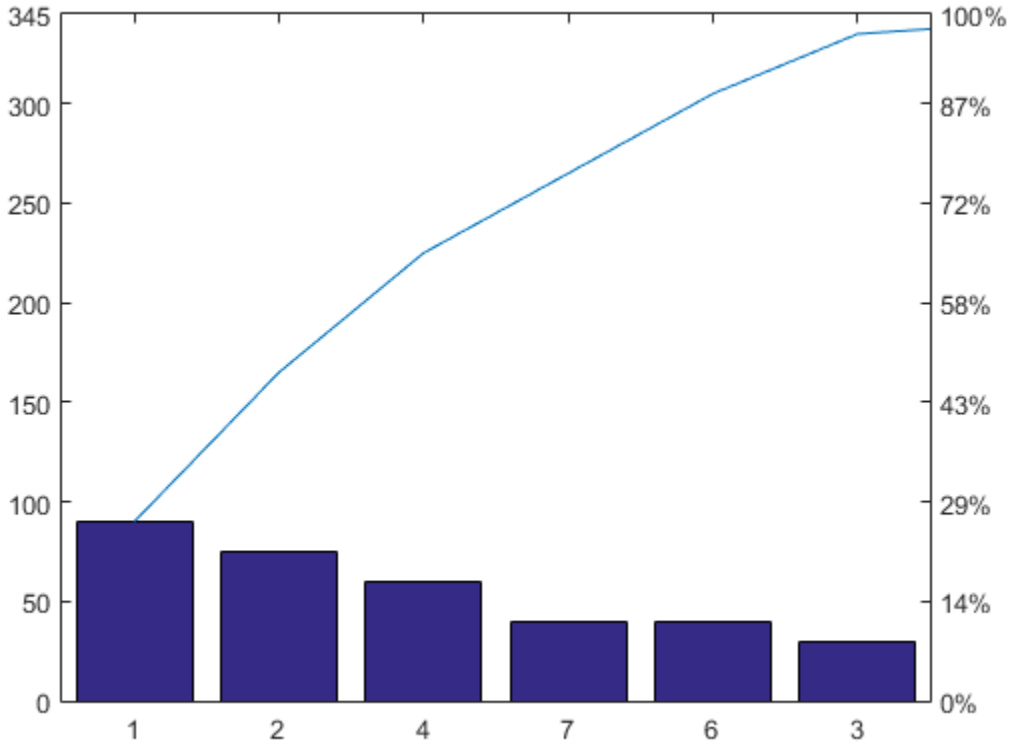
`H = pareto(...)` returns handles to the primitive line and bar series objects created.

Examples

Create Pareto Chart

Create a Pareto chart of vector `y`.

```
y = [90,75,30,60,5,40,40,5];
figure
pareto(y)
```



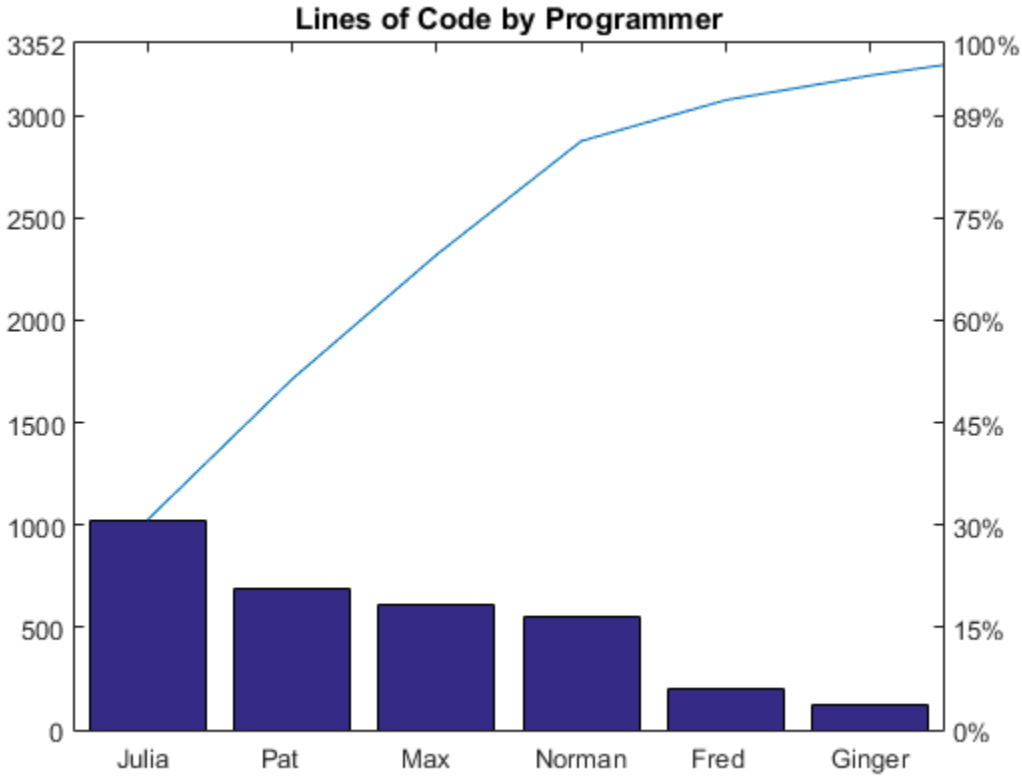
`pareto` displays the elements in `y` as bars in descending order and labels each bar with its index in `y`. Since `pareto` displays only the first 95% of the cumulative distribution, some elements in `y` are not displayed.

Label Bars in Pareto Chart

Examine the cumulative productivity of a group of programmers to see how normal its distribution is. Label each bar with the name of the programmer.

```
codelines = [200 120 555 608 1024 101 57 687];
```

```
coders = {'Fred','Ginger','Norman','Max','Julia','Wally','Heidi','Pat'};

figure
pareto(codelines, coders)
title('Lines of Code by Programmer')
```



More About

Tips

You cannot place datatips (use the Datacursor tool) on graphs created with `pareto`.

See Also

Functions
`bar` | `histogram` | `line`

Properties
Bar Series Properties | Primitive Line Properties

Introduced before R2006a

parfor

Parallel for loop

Syntax

```
parfor loopvar = initval:endval; statements; end
parfor (loopvar = initval:endval, M); statements; end
```

Description

`parfor loopvar = initval:endval; statements; end` executes a series of MATLAB statements for values of `loopvar` between `initval` and `endval`, inclusive, which specify a vector of increasing integer values. The loop runs in parallel when you have the Parallel Computing Toolbox or when you create a MEX function or standalone code with MATLAB Coder. Unlike a traditional `for`-loop, iterations are not executed in a guaranteed order.

`parfor (loopvar = initval:endval, M); statements; end` executes statements in a loop using a maximum of `M` workers or threads, where `M` is a nonnegative integer.

Examples

Perform three large eigenvalue computations using three workers or cores with Parallel Computing Toolbox software:

```
parpool(3)
parfor i=1:3, c(:,i) = eig(rand(1000)); end
```

More About

Tips

- If you have Parallel Computing Toolbox software, see the function reference pages for `parfor` and `parpool` for additional information.

- If you have MATLAB Coder software, see the `parfor` function reference page for additional information.

See Also

`for`

parse

Class: inputParser

Parse function inputs

Syntax

```
parse(p,argList)
```

Description

`parse(p,argList)` parses and validates the inputs in `arglist`.

Input Arguments

`p`

Object of class `inputParser`.

`argList`

Comma separated list of inputs to parse and validate for your custom function. The class of each input depends upon your function definition.

Examples

Input Parsing

Parse and validate required and optional function inputs.

Create a custom function with required and optional inputs in the file `findArea.m`.

```
function a = findArea(width,varargin)
   p = inputParser;
   defaultHeight = 1;
```

```
defaultUnits = 'inches';
defaultShape = 'rectangle';
expectedShapes = {'square','rectangle','parallelogram'};

addRequired(p,'width',@isnumeric);
addOptional(p,'height',defaultHeight,@isnumeric);
addParameter(p,'units',defaultUnits);
addParameter(p,'shape',defaultShape,...
             @(x) any(validatestring(x,expectedShapes)));

parse(p,width,varargin{:});
a = p.Results.width .* p.Results.height;
```

The input parser checks whether `width` and `height` are numeric, and whether the `shape` matches a string in cell array `expectedShapes`. `@` indicates a function handle, and the syntax `@(x)` creates an anonymous function with input `x`.

Call the function with inputs that do not match the scheme. For example, specify a nonnumeric value for the `width` input:

```
findArea('text')
```

```
Error using findArea (line 14)
The value of 'width' is invalid. It must satisfy the function: isnumeric.
```

Specify an unsupported value for `shape`:

```
findArea(4,'shape','circle')
```

```
Error using findArea (line 14)
The value of 'shape' is invalid. Expected input to match one of these strings:

square, rectangle, parallelogram

The input, ''circle'', did not match any of the valid strings.
```

See Also

`addOptional` | `addParameter` | `addRequired` | `inputParser`

parseSoapResponse

Convert response string from SOAP (Simple Object Access Protocol) server into MATLAB types

Compatibility

`parseSoapResponse` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
data = parseSoapResponse(response)
```

Description

`data = parseSoapResponse(response)` extracts data from SOAP server `response` and converts to MATLAB types.

Examples

Retrieve Book Information from Library Database

This example assumes the library is on a local intranet and does not use an actual endpoint; therefore, you cannot run it.

Retrieve the name of the author of a book titled “In the Fall.” The relative path of the library service is `urn:LibraryCatalog`. To get the author's name, use the `getAuthor` function, which takes the book name as the input value. The `getAuthor` parameter is `nameToLookUp`. The XML data type for title is `{http://www.w3.org/2001/XMLSchema}string`. The SOAP message style is `rpc` by default.

Create the SOAP message.

```
message = createSoapMessage(...
```

```
  'urn:LibraryCatalog',...
  'getAuthor',...
  {'In the Fall'},...
  {'nameToLookUp'},...
  {'{http://www.w3.org/2001/XMLSchema}string'})

message =

[#document: null]
```

This response does not necessarily indicate that the message is valid, although certain input problems produce error messages.

Send the message to the server for processing, and get the author's name back. The server endpoint is `http://test/soap/services/LibraryCatalog`. The server method is `urn:LibraryCatalog#getAuthor`.

```
response = callSoapService(...
  'http://test/soap/services/LibraryCatalog',...
  'urn:LibraryCatalog#getAuthor',...
  message)

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<getAuthorResponse xmlns="urn:LibraryCatalog">
<ns1:getAuthorReturn xmlns:ns1="http://latestversion.soap.test">
Kate Alvin
</ns1:getAuthorReturn>
</getAuthorResponse>
</soapenv:Body>
</soapenv:Envelope>
```

MATLAB returns the message in a single line, displayed here on separate lines for legibility.

Extract the author's name.

```
author = parseSoapResponse(response)

author = Kate Alvin
```

MATLAB automatically converted the XML string data type to `char`.

Input Arguments

`response` — Data from SOAP server

string

Data from SOAP server, specified as a string. response is the output from the `callSoapService` function.

Output Arguments

`data` — Output of SOAP service call

cell array of any valid MATLAB type

Output of SOAP service call, returned as a cell array of any valid MATLAB type. For information about data, see the documentation for the SOAP service used in the `callSoapService` function.

See Also

`callSoapService` | `createSoapMessage` | `matlab.wsdl.createWSDLClient` | `urlread` | `xmlread`

Introduced before R2006a

pascal

Pascal matrix

Syntax

```
A = pascal(n)
A = pascal(n,1)
A = pascal(n,2)
```

Description

`A = pascal(n)` returns a "Pascal's Matrix" on page 1-5740 of order `n`: a symmetric positive definite matrix with integer entries taken from Pascal's triangle. The inverse of `A` has integer entries.

`A = pascal(n,1)` returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. It is *involutary*, that is, it is its own inverse.

`A = pascal(n,2)` returns a transposed and permuted version of `pascal(n,1)`. `A` is a cube root of the identity matrix.

Examples

`pascal(4)` returns

```
1     1     1     1
1     2     3     4
1     3     6    10
1     4    10    20
```

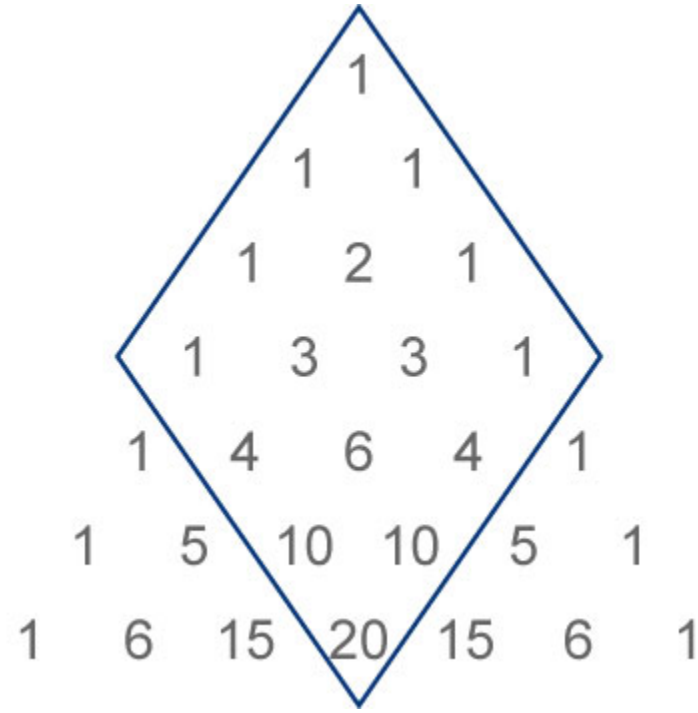
`A = pascal(3,2)` produces

```
A =
     1     1     1
    -2    -1     0
     1     0     0
```

More About

Pascal's Matrix

Pascal's triangle is a triangle formed by rows of numbers. The first row has entry `1`. Each succeeding row is formed by adding adjacent entries of the previous row, substituting a `0` where there is no adjacent entry. Pascal's matrix is generated by selecting the portion of Pascal's triangle that corresponds to the specified matrix dimensions, as outlined in the graphic. The matrix outlined corresponds to the MATLAB command `pascal(4)`.



See Also

`chol` | `gallery` | `vander`

Introduced before R2006a

patch

Create one or more filled polygons

Syntax

```
patch(X,Y,C)
patch(X,Y,Z,C)
patch(FV)
patch(X,Y,C,'PropertyName',propertyvalue...)
patch('PropertyName',propertyvalue,...)
handle = patch(...)
```

Properties

For a list of properties, see Patch Properties.

Description

`patch(X,Y,C)` adds a filled 2-D patch object to the current axes. A patch object is one or more polygons defined by the coordinates of its vertices. The elements of `X` and `Y` specify the vertices of a polygon . If `X` and `Y` are `m`-by-`n` matrices, MATLAB draws `n` polygons with `m` vertices. `C` determines the color of the patch. For more information on color input requirements, see “Coloring Patches” on page 1-5746.

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the end of the `Faces` matrix with `NaNs`. To define a patch with faces that do not close, add one or more `NaNs` to the row in the `Vertices` matrix that defines the vertex you do not want connected.

See “Introduction to Patch Objects” for more information on using patch objects.

`patch(X,Y,Z,C)` creates a patch in 3-D coordinates. If the coordinate data does not define closed polygons, `patch` closes the polygons. The data can define concave or intersecting polygons. However, if the edges of an individual patch face intersect themselves, the resulting face might be only partly filled. In that case, it is better to divide the face into smaller polygons.

`patch(FV)` creates a patch using structure `FV`, which contains the fields `vertices`, `faces`, and optionally `facevertexcdata`. These fields correspond to the `Vertices`, `Faces`, and `FaceVertexCData` patch properties. Specifying only unique vertices and their connection matrix can reduce the size of the data for patches having many faces. For an example of how to specify patches with this method, see "Specifying Patch Object Shapes" on page 1-5743.

`patch(X,Y,C,'PropertyName',propertyvalue...)` follows the `X`, `Y`, `(Z)`, and `C` arguments with property name/property value pairs to specify additional patch properties. For a description of the properties, see Patch Properties. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

`patch('PropertyName',propertyvalue,...)` specifies all properties using property name/property value pairs. This form lets you omit the color specification because MATLAB uses the default face color and edge color unless you explicitly assign a value to the `FaceColor` and `EdgeColor` properties. This form also lets you specify the patch using the `Faces` and `Vertices` properties instead of *x*-, *y*-, and *z*-coordinates. See "Specifying Patch Object Shapes" on page 1-5743 for more information.

`handle = patch(...)` returns the handle of the patch object it creates.

Unlike high-level area creation functions, such as `fill` or `area`, `patch` does not check the settings of the figure and axes `NextPlot` properties. It simply adds the patch object to the current axes.

Examples

Define a Square Patch

Create a square patch with red face color and black edges using x- and y-coordinates:

```
x = [0 1 1 0];
y = [0 0 1 1];
patch(x,y,'red')
```

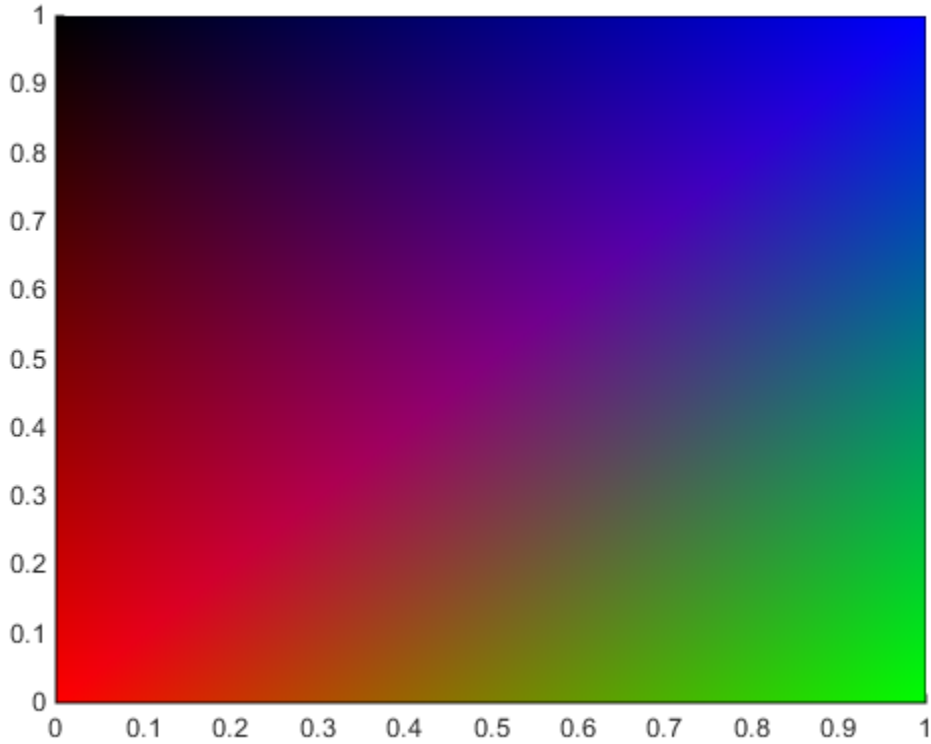
Create a square patch with red face color and black edges using vertices and faces:

```
vert = [0 0;1 0;1 1;0 1]; % x and y vertex coordinates
fac = [1 2 3 4]; % vertices to connect to make square
```

```
patch('Faces',fac,'Vertices',vert,'FaceColor','red')
```

Specify colors for each vertex and interpolate the face color:

```
fvc = [1 0 0;0 1 0;0 0 1;0 0 0];
patch('Faces',fac,'Vertices',vert,...
'FaceVertexCData',fvc,'FaceColor','interp')
```



Specifying Patch Object Shapes

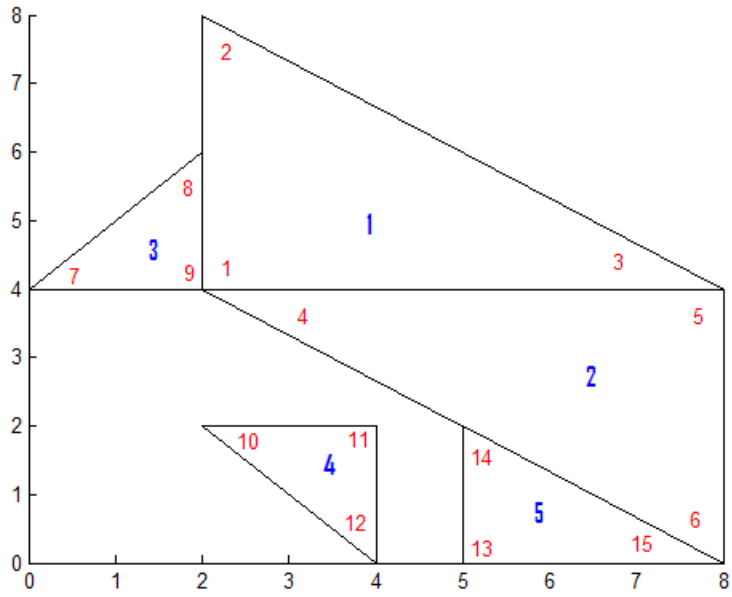
The next two examples create a patch object using two methods:

- Specifying *x*-, *y*-, and *z*-coordinates and color data (`XData`, `YData`, `ZData`, and `CData` properties)

- Specifying vertices, the connection matrix, and color data (`Vertices`, `Faces`, and `FaceVertexCData` properties)

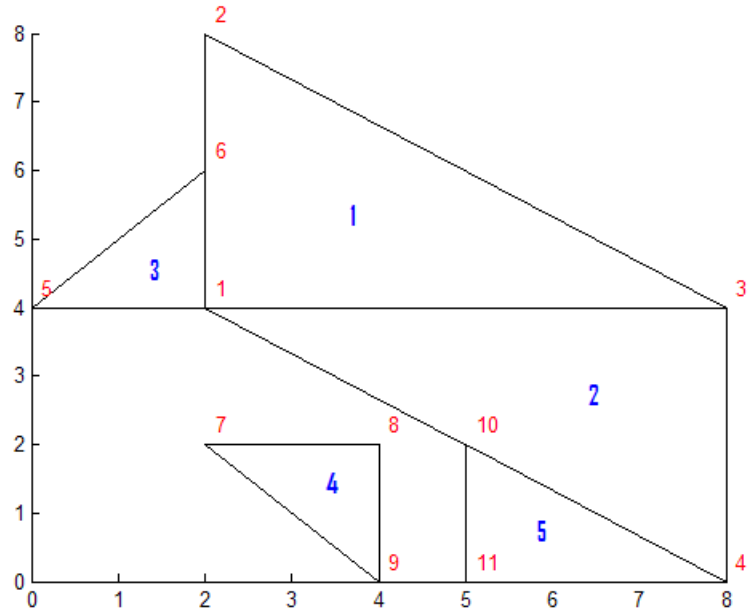
Create five triangular faces, each having three vertices, by specifying the *x*-, *y*-, and *z*-coordinates of each vertex:

```
xdata = [2 2 0 2 5;
         2 8 2 4 5;
         8 8 2 4 8];
ydata = [4 4 4 2 0;
         8 4 6 2 2;
         4 0 4 0 0];
zdata = ones(3,5);

% Red numbers denote the vertex indices.
% For this example:
% xindices = [1 4 7 10 13;
%             2 5 8 11 14;
%             3 6 9 12 15];
% Blue numbers denote the face numbers.
patch(xdata,ydata,zdata,'w')
```



Create the five triangular faces, specifying faces and vertices:

```
% The Vertices property contains the coordinates of each
% unique vertex defining the patch. The Faces property
% specifies how to connect these vertices to form each
% face of the patch. More than one face may
% use a given vertex.
% For this example, five triangles have 11 total vertices,
% instead of 15. Each row contains
% the x- and y-coordinates
% of each vertex.
verts = [2 4; ...
        2 8; ...
        8 4; ...
        8 0; ...
        0 4; ...
        2 6; ...
        2 2; ...
        4 2; ...
        4 0; ...
        5 2; ...
        5 0 ];

% There are five faces, defined by connecting the
% vertices in the order indicated.
faces = [ ...
        1  2  3; ...
        1  3  4; ...
        5  6  1; ...
        7  8  9; ...
        11 10 4 ];

% Create the patch by specifying the Faces, Vertices,
% and FaceVertexCData properties as well as the
% FaceColor property. Red numbers denote the vertex
% numbers, as defined in faces. Blue indicate face numbers.
p = patch('Faces',faces,'Vertices',verts,'FaceColor','w');
```




```
% Using the previous values for verts and faces, you can
% create the same patch object using a structure:
patchinfo.Vertices = verts;
patchinfo.Faces = faces;
patchinfo.FaceColor = 'w';

patch(patchinfo);
```

Coloring Patches

There are many ways to customize your patch objects using colors. The appropriate input depends on:

- Whether you want to change the edge colors
- How you specified the patch faces:
  - Using face/vertex values
  - Using *x*-, *y*-, and *z*-coordinates

The following sections present the various options available.

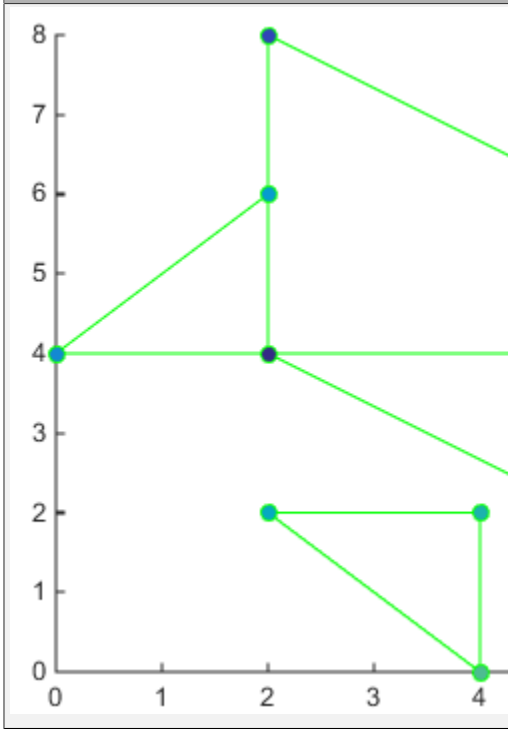
Specifying Edge Colors

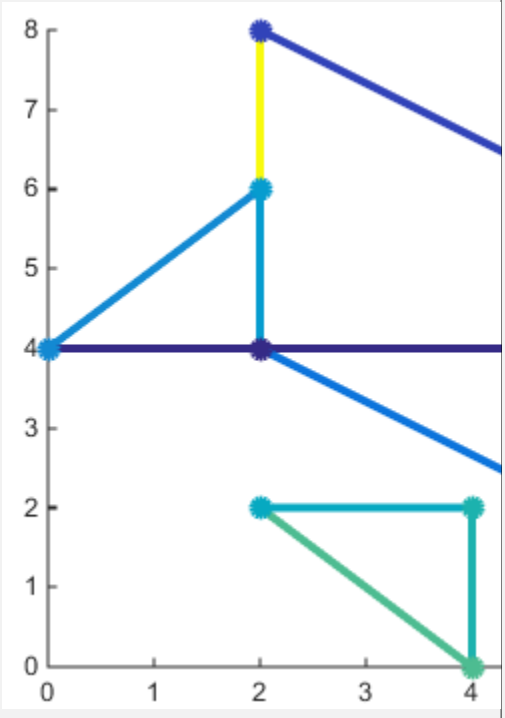
The following options apply to the edge colors of your patch object. The settings are independent of the face colors, but the colors themselves depend on the colors specified at each vertex. Markers show the color at each vertex. Specify the colors using the `EdgeColor` property. Explore the options using the Sample Code in the following table. Create a base patch object with this code:

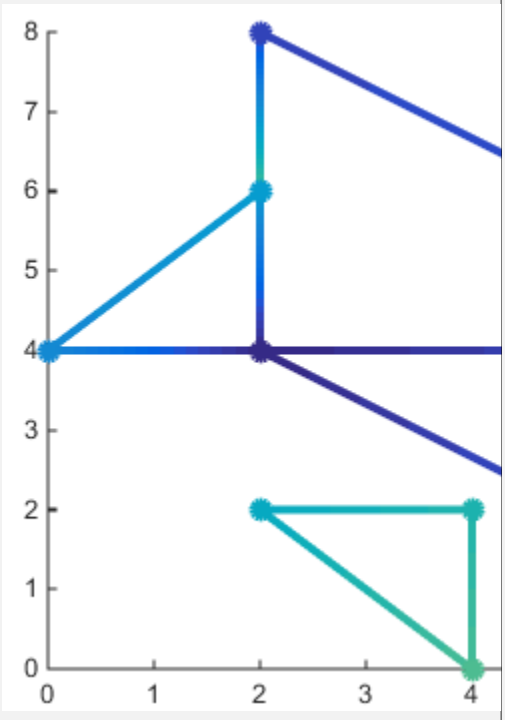
```
xdata = [2     2     0     2     5;
         2     8     2     4     5;
         8     8     2     4     8];
ydata = [4     4     4     2     0;
         8     4     6     2     2;
         4     0     4     0     0];
cdata = [15     0     4     6    10;
         1     2     5     7     9;
         2     3     0     8     3];
p = patch(xdata,ydata,cdata,'Marker','o',...
          'MarkerFaceColor','flat',...
          'FaceColor','none')
```

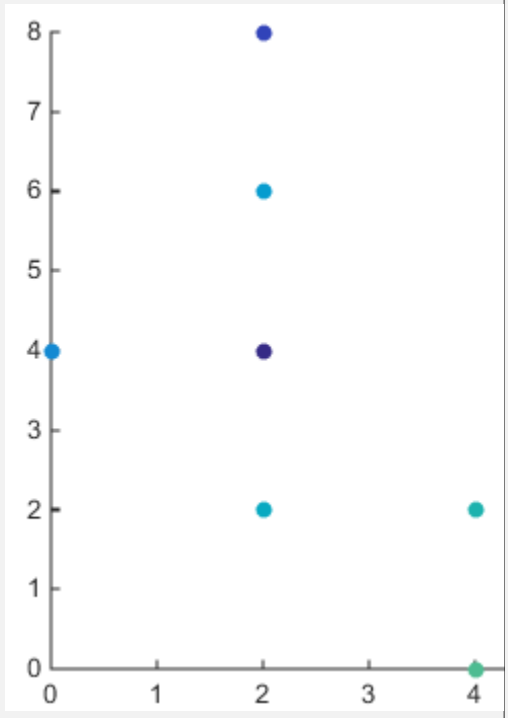
For more detailed information on how the `EdgeColor` property works, see thePatch Properties property page.

Desired Look	EdgeColor Value	Sample Code
All edges have the same color, around all faces. This option does not rely on the `FaceColor` value.	MATLAB color specifier.`EdgeColor`	`set(p,'EdgeColor','g')`

Desired Look	EdgeColor Value	Sample Code
		

Desired Look	EdgeColor Value	Sample Code
Each edge corresponds to the color of the vertex that precedes the edge, with one color per edge. This option requires that the `FaceColor` property be `flat` or `interp`. By default, if you specify `CData` when creating the patch object, its `FaceColor` property is `interp`. 	`'flat'`	`set(p,'EdgeColor','flat',...` `'LineWidth',3)`

Desired Look	EdgeColor Value	Sample Code
Each edge corresponds to the vertex colors, interpolated between vertices. This option requires that the `FaceColor` property be `flat` or `interp`. By default, if you specify `CData` when creating the patch object, its `FaceColor` property is `interp`. 	`'interp'`	`set(p,'EdgeColor','interp')`

Desired Look	EdgeColor Value	Sample Code
Edges have no color. This option does not rely on the `FaceColor` value. If set, markers retain vertex colors. 	`'none'`	`set(p,'EdgeColor','none')`

Specifying Face Colors Using Face/Vertex Input Matrices

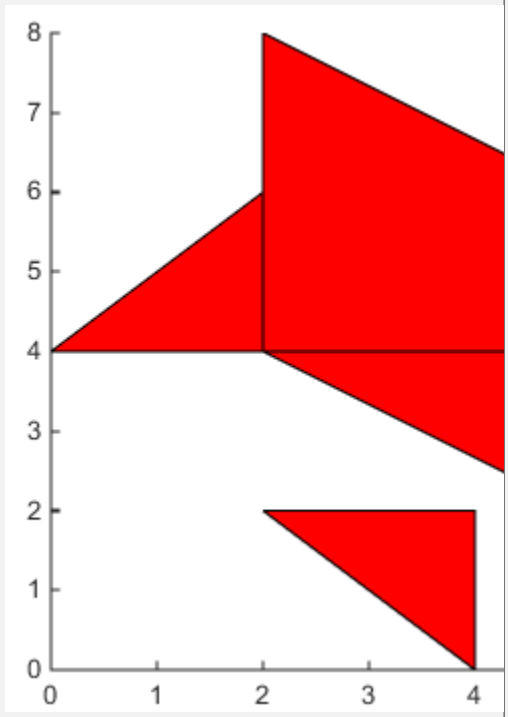
The following options apply to the face colors of your patch object when you specify the faces using face/vertex input matrices. To explore the options, start with a base patch object:

```
% For this example, there are five triangles (m = 5)
% sharing eleven unique vertices (k = 11).
verts = [2 4; ...
         2 8; ...
         8 4; ...
         8 0; ...
```

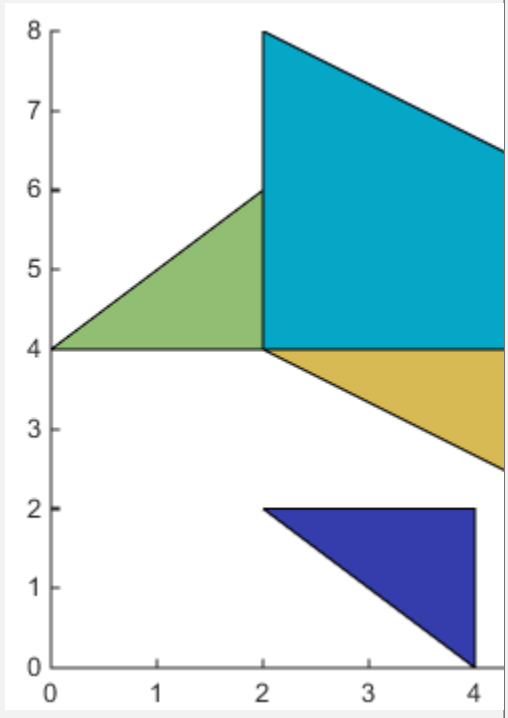
```
          0 4; ...
          2 6; ...
          2 2; ...
          4 2; ...
          4 0; ...
          5 2; ...
          5 0 ];
faces = [1  2  3; ...
         1  3  4; ...
         5  6  1; ...
         7  8  9; ...
         11 10 4];
p = patch('Faces',faces,'Vertices',verts,'FaceColor','b');
```

For more information on the relevant properties, see `FaceColor`, `FaceVertexCData`, and `CDataMapping`.

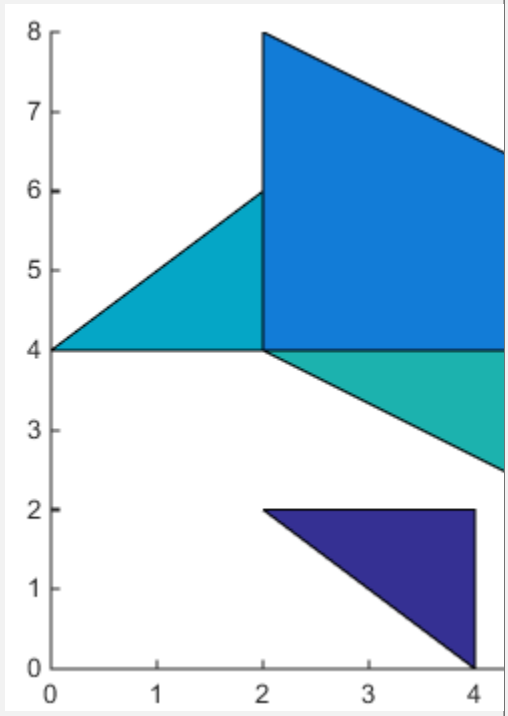
Desired Look	Parameter Values	Sample Code

All faces have the same color.
 
 	
	• `FaceColor`
	• `FaceVertexCData`: `[]` (no input)
	An empty array is the default value, and `patch` ignores any input until you set `FaceColor` to `'flat'` or `'interp'`.
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`set(p,'FaceColor','r')`
		or
		`set(p,'FaceColor',[1 0 0])`

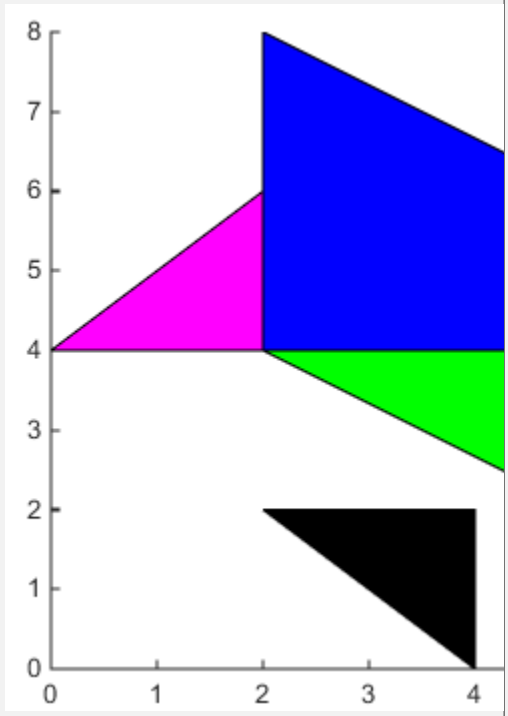
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, indexed from a selected section of the colormap.
 
 	
	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: `m`-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','scaled')`

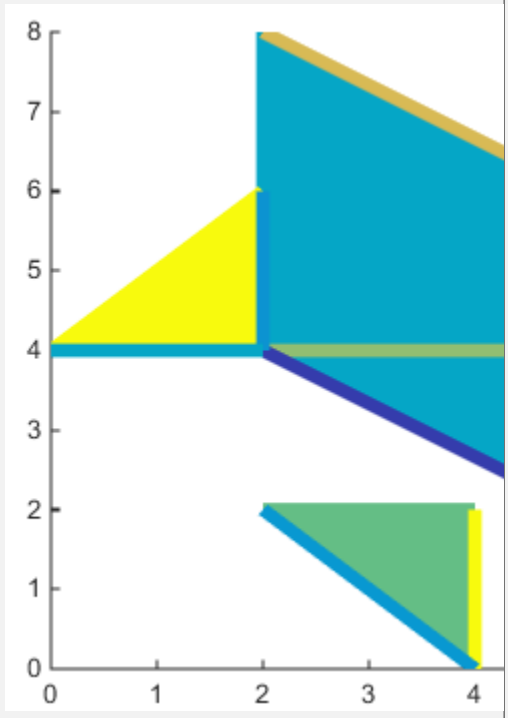
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, indexed from the whole colormap.
 
 	
	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','dire`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','direct')`

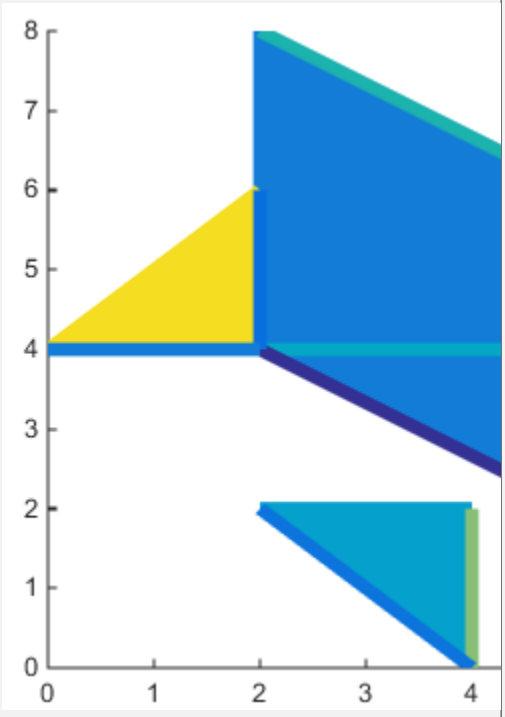
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, determined by truecolor value input.
 
 	
	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: `m`-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1 0 0.8;`
		`0 1 0 0 0.8;`
		`1 0 1 0 0.8]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata)`

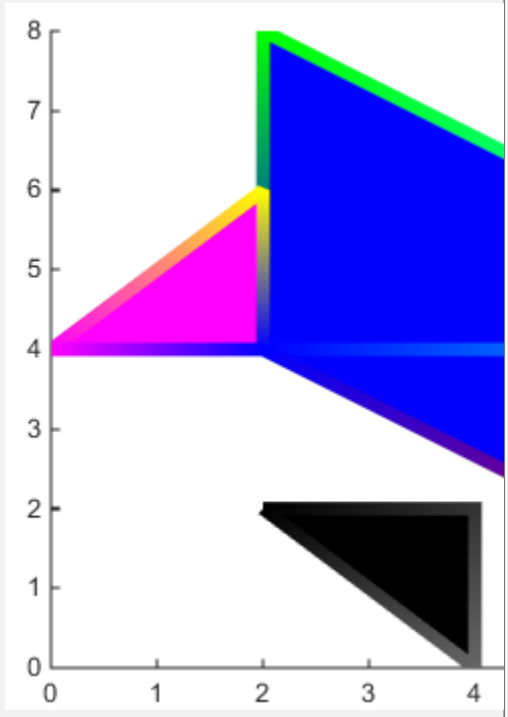
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 ...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5,...`
		`'CDataMapping','scaled')`

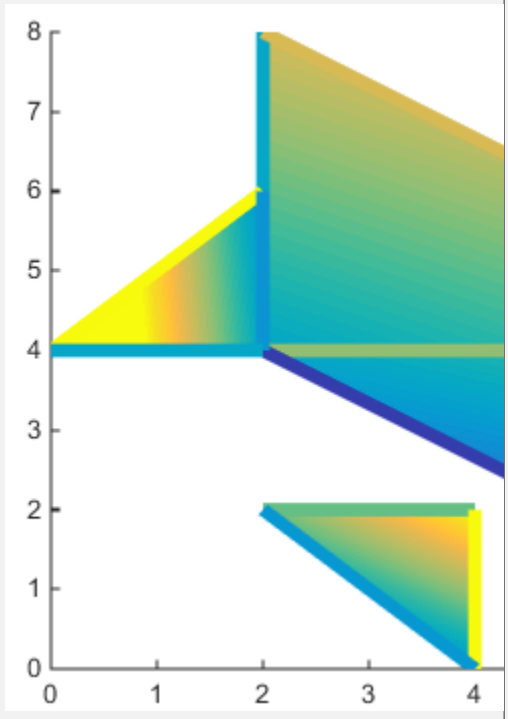
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from the whole colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','dire`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 ...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

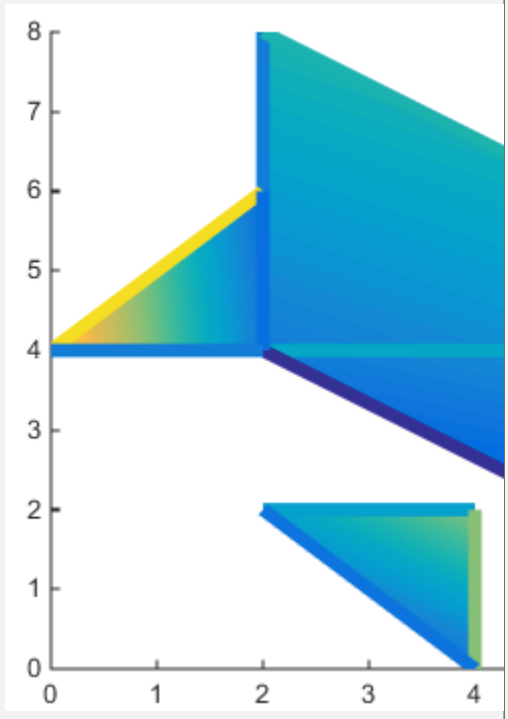
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, determined by truecolor value input. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: k-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1;`
		`0 1 0;`
		`0 1 1;`
		`1 0 0;`
		`1 0 1;`
		`1 1 0;`
		`0 0 0;`
		`0.2 0.2 0.2;`
		`0.4 0.4 0.4;`
		`0.6 0.6 0.6;`
		`0.8 0.8 0.8];`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

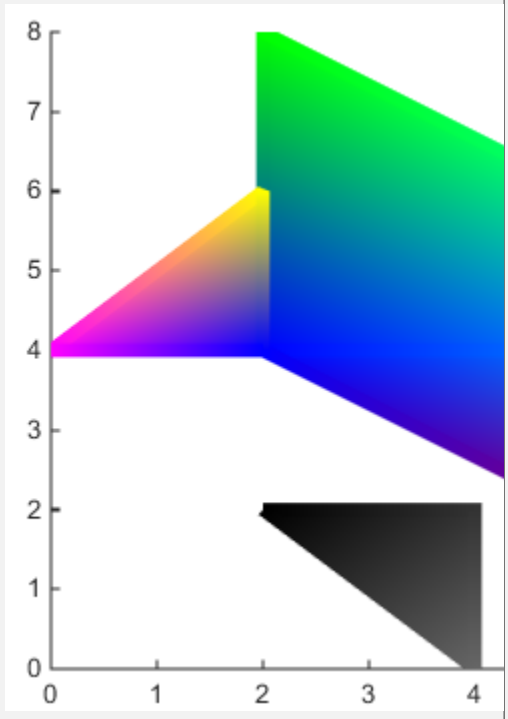
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 ...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','interp',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5,...`
		`'CDataMapping','scaled')`

Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from the whole colormap. Edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','dire`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 ...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','interp',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, determined by truecolor value input. Edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: k-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1;`
		`0 1 0;`
		`0 1 1;`
		`1 0 0;`
		`1 0 1;`
		`1 1 0;`
		`0 0 0;`
		`0.2 0.2 0.2;`
		`0.4 0.4 0.4;`
		`0.6 0.6 0.6;`
		`0.8 0.8 0.8];`
		`set(p,'FaceColor','interp',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

Specifying Face Colors Using x-, y-, and z-Coordinate Input

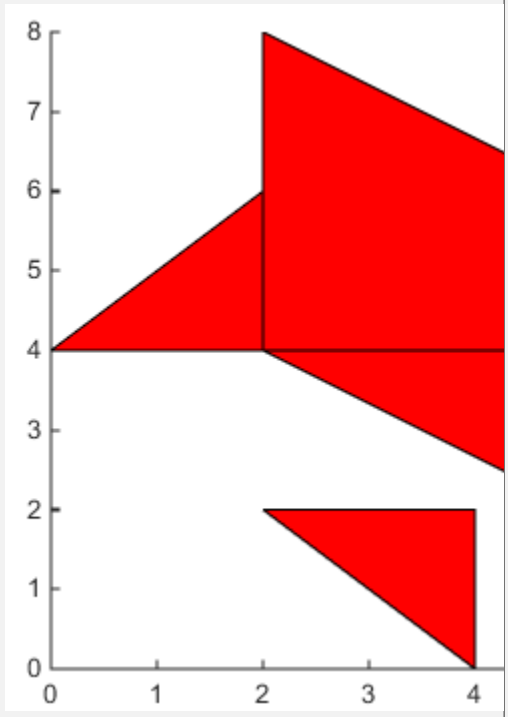
The following options apply to the face colors of your patch object when you specify the faces using *x*-, *y*-, and *z*-coordinates. To explore the options, start with a base patch object:

```
% For this example, there are five (m=5) triangles (n=3).
% The total number of vertices is mxn, or k = 15.
xdata = [2 2 0 2 5;
         2 8 2 4 5;
         8 8 2 4 8];
```

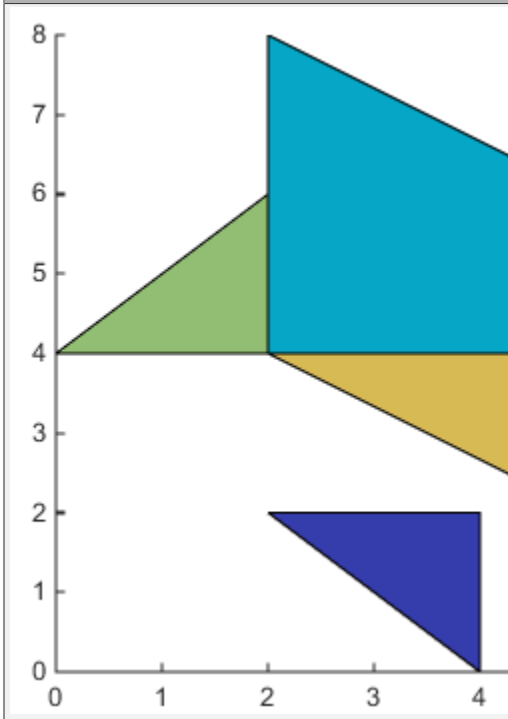
```
ydata = [4 4 4 2 0;
         8 4 6 2 2;
         4 0 4 0 0];
zdata = ones(3,5);
p = patch(xdata,ydata,zdata,'b')
```

For more information on the relevant properties, see `FaceColor`, `CData`, and `CDataMapping`.

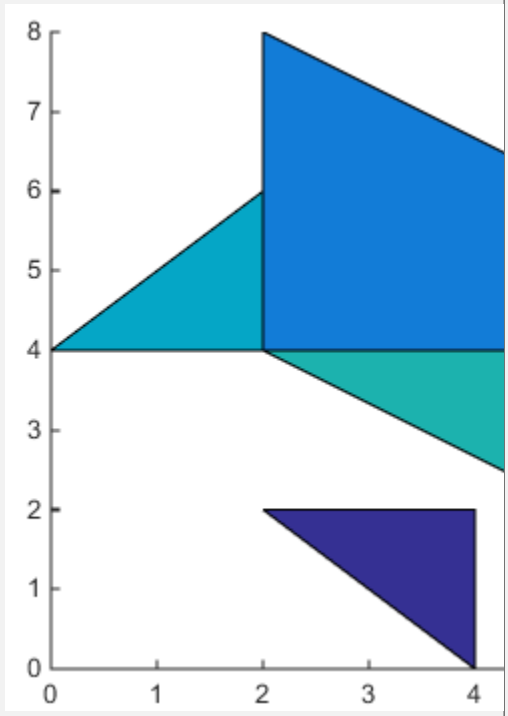
Desired Look	Parameter Values	Sample Code

All faces have the same color.
 
 	
	• `FaceColor`: `ColorSpec`
	• `CData`: `[]` (no input)
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	
	`'scaled'` is the default value, but neither affects the outcome.
		`set(p,'FaceColor','r')`
		
		or
		
		`set(p,'FaceColor',[1 0 0])`

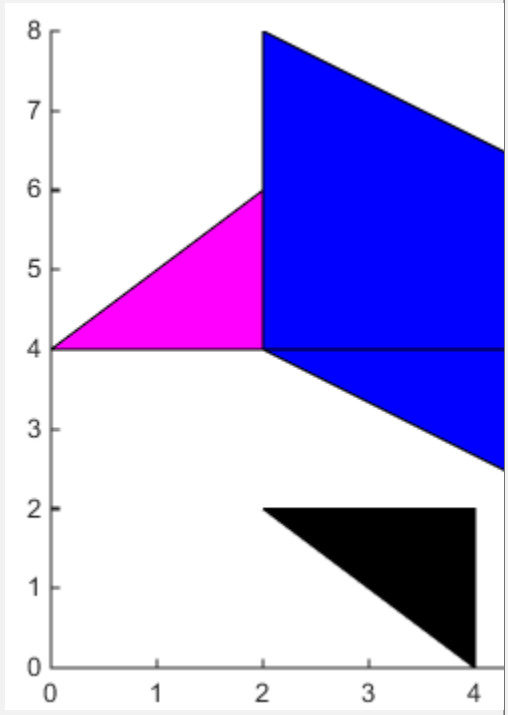
Each face has a single, unique color, indexed from a selected section of the colormap.	• `FaceColor`: `'flat'`
	• `CData`: m-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'CDataMapping','scaled')`

Desired Look	Parameter Values	Sample Code
	• `CDataMapping: 'scaled'`	

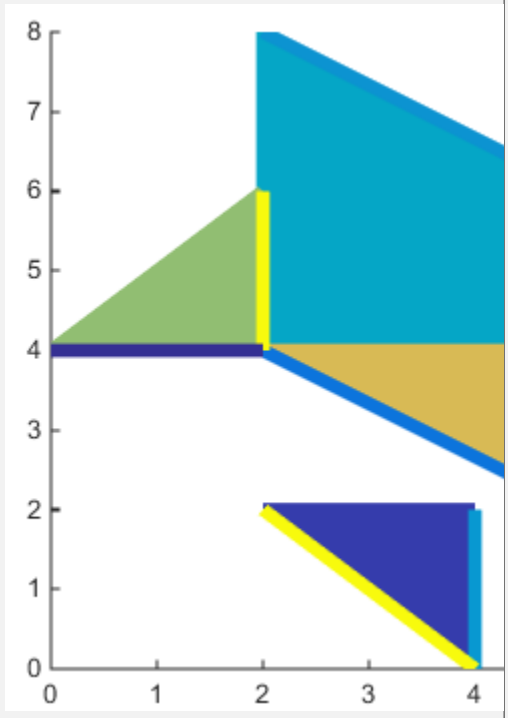
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, indexed from the whole colormap.
 
 	
	• `FaceColor`: `'flat'`
	• `CData`: `m`-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','dire`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'CDataMapping','direct')`

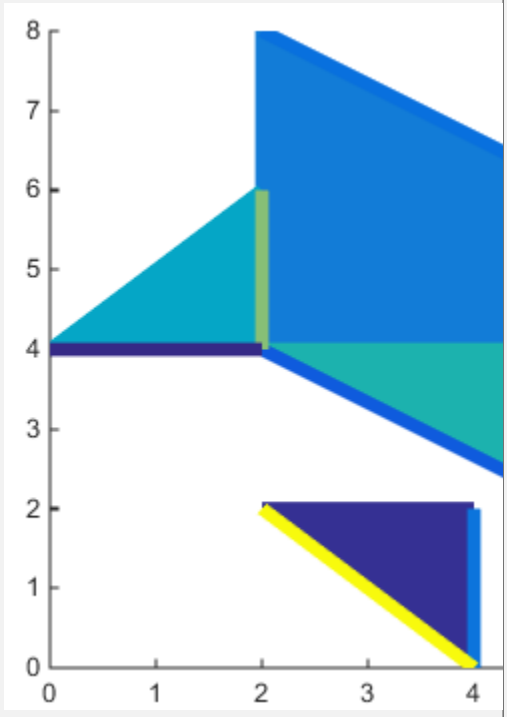
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, determined by truecolor value input.
 
 	
	• `FaceColor`: `'flat'`
	• `CData`: 1-by-`m`-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata(:,:,1) = [0 0 1 0 0.8];`
		`cdata(:,:,2) = [0 0 0 0 0.8];`
		`cdata(:,:,3) = [1 1 1 0 0.8];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata)`

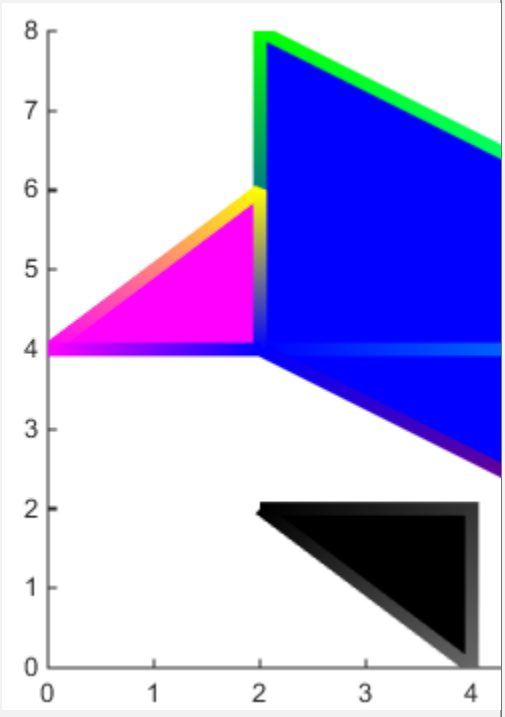
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'flat'`
	• `CData`: `m`-by-`n` matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`12 23 40 13 26;`
		`24 8 1 65 42];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5,...`
		`'CDataMapping','scaled')`

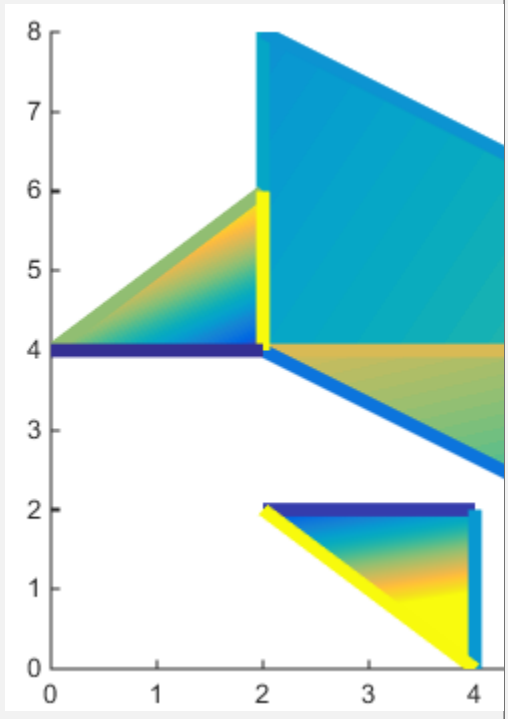
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from the whole colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'flat'`
	• `CData`: `m`-by-`n` matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','dire`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`12 23 40 13 26;`
		`24 8 1 65 42];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

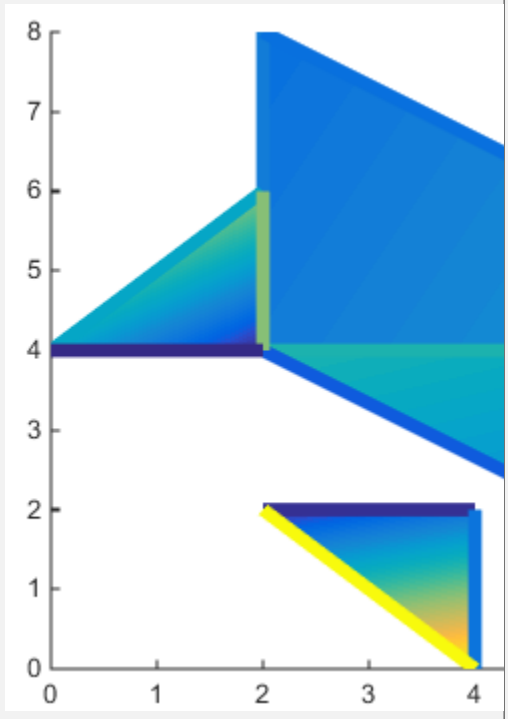
Desired Look	Parameter Values	Sample Code

Each vertex has a single, unique color, determined by truecolor value input. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'flat'`
	• `CData`: `m`-by-`n`-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata(:,:,1) = [0 0 1 0 0.8;`
		`0 0 1 0.2 0.6;`
		`0 1 0 0.4 1];`
		
		`cdata(:,:,2) = [0 0 0 0 0.8;`
		`1 1 1 0.2 0.6;`
		`1 0 0 0.4 0];`
		
		`cdata(:,:,3) = [1 1 1 0 0.8;`
		`0 1 0 0.2 0.6;`
		`1 0 1 0.4 0];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

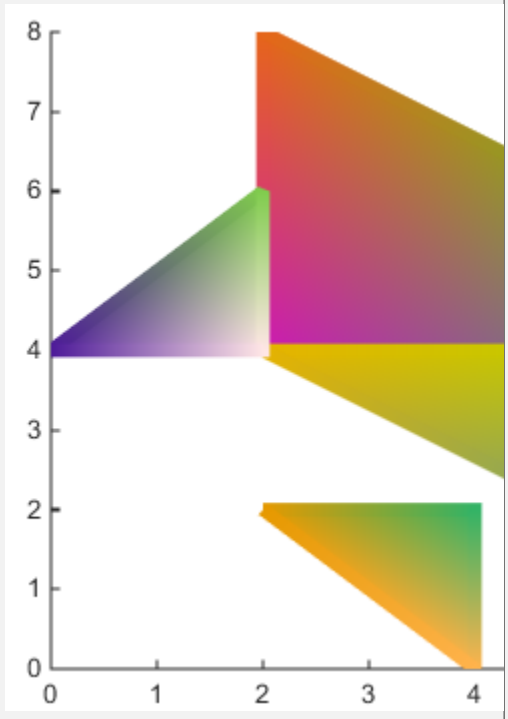
Desired Look	Parameter Values	Sample Code

Each vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'interp'`
	• `CData`: `m`-by-`n` matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`12 23 40 13 26;`
		`24 8 1 65 42];`
		`set(p,'FaceColor','interp',...`
		`'CData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5,...`
		`'CDataMapping','scaled')`

Desired Look	Parameter Values	Sample Code

Each vertex has a single, unique color, indexed from the whole colormap. Edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'interp'`
	• `CData`: `m`-by-`n` matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','dire`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`12 23 40 13 26;`
		`24 8 1 65 42];`
		`set(p,'FaceColor','interp',...`
		`'CData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

Desired Look	Parameter Values	Sample Code

Each vertex has a single, unique color, determined by truecolor value input. Edges may have `'flat'` or `'interp'` color.
 
	• `FaceColor`: `'interp'`
	• `CData`: `m`-by-`n`-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata(:,:,1) = [0.8 0.1 0.2`
		`  0.9 0.3 1;`
		`  0.1 0.5 0.9;`
		`  0.9 1 0.5;`
		`  0.6 0.9 0.8];`
		
		`cdata(:,:,2) =[0.1 0.6 0.7;`
		`  0.4 0.1 0.7;`
		`  0.9 0.8 0.3;`
		`  0.7 0.9 0.6;`
		`  0.9 0.6 0.1];`
		
		`cdata(:,:,3) =[0.7 0.8 0.4;`
		`  0.1 0.6 0.3;`
		`  0.2 0.3 0.7;`
		`  0.0 0.9 0.7;`
		`  0.0 0.0 0.1];`
		`set(p,'FaceColor','interp',...`
		`'CData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

See Also

`area` | `caxis` | `fill` | `fill3` | `isosurface`

Introduced before R2006a

Patch Properties

Control patch appearance and behavior

Patch properties control the appearance and behavior of patch objects. By changing property values, you can modify certain aspects of the patch.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = patch;
c = h.CData;
h.CDataMapping = 'scaled';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Faces

`FaceColor` — Face color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet or color string

Face color, specified as one of these values:

- `'none'` — Do not draw the faces.
- `'flat'` — Uniform face colors. Use the CData values. The color data at the first vertex determines the color for the entire face. You must first specify `CData` or `FaceVertexCData` as an array containing one value per face.
- `'interp'` — Vary the color across each face. Use a bilinear interpolation of the CData values at each vertex. You must first specify `CData` or `FaceVertexCData` as an array containing one value per vertex.
- RGB triplet or color string — Same color for all of the faces.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`

Long Name	Short Name	RGB Triplet
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

`FaceAlpha` — Face transparency

1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use the same transparency value for all of the faces. A value of `1` is fully opaque and `0` is completely transparent.
- `'flat'` — Use uniform transparency across each face. The alpha data at the first vertex determines the transparency for the entire face. You must first specify the `FaceVertexAlphaData` property as an array containing one alpha value per face.
- `'interp'` — Vary the transparency across each face. Bilinear interpolation of the `AlphaData` values at each vertex determines the transparency values. You must first specify the FaceVertexAlphaData property as an array containing one alpha value per vertex.

`FaceLighting` — Effect of light objects on faces

`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across the faces. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

Note: The `'phong'` value has been removed. Use `'gouraud'` instead.

`BackFaceLighting` — Face lighting when normals point away from camera

`'reverslit'` (default) | `'unlit'` | `'lit'`

Face lighting when the vertex normals point away from camera, specified as one of these values:

- `'reverslit'` — Light the face as if the vertex normal pointed towards the camera.
- `'unlit'` — Do not light the face.
- `'lit'` — Light the face according to the vertex normal.

Use this property to discriminate between the internal and external surfaces of an object. For an example, see “Back Face Lighting”.

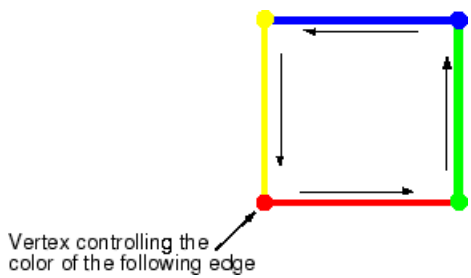
Edges

`EdgeColor` — Edge color of the patch faces

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet or color string

Color of the edges of the patch faces, specified as one of these values:

- `'none'` — Do not draw edges.
- `'flat'` — Use the color of each vertex to set the color of the edge that follows it. This means that the edge coloring is dependent on the order in which you specify the vertices:



- `'interp'` — Linearly interpolate the `CData` or `FaceVertexCData` values at the vertices determines the edge color.
- RGB triplet or a color string — Use the same color for all edges.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

`[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'flat'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`EdgeAlpha` — Transparency of patch face edges

`1` (default) | scalar value in range `[0,1]` | `'flat'` | `'interp'`

Transparency of patch face edges, specified as one of these values:

- Scalar value in range `[0,1]` — Use the same transparency value for all of the edges.. A value of `1` is fully opaque and `0` is completely transparent.
- `'flat'` — Uniform transparency along each edge. Before you set EdgeAlpha to `'flat'`, you must first define FaceVertexAlphaData as an array containing one alpha value per face. The alpha data of each vertex controls the transparency of the edge that follows it.
- `'interp'` — Very the transparency along edge segment. Before you set EdgeAlpha to `'interp'`, you must first define FaceVertexAlphaData as an array containing one alpha value per vertex. Linear interpolation of the alpha data at each vertex determines the transparency of the edge connecting those vertices.

`EdgeLighting` — Effect of light objects on edges

`'none'` (default) | `'flat'` | `'gouraud'`

Effect of light objects on edges, specified as one of these values:

- `'flat'` — Apply light uniformly across the each edges.
- `'none'` — Do not apply lights from light objects to the edges.
- `'gouraud'` — Calculate the light at the vertices, and then linearly interpolate across the edges.

Note: The `'phong'` value has been removed. Use `'gouraud'` instead.

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the patch object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | `'flat'` | RGB triplet or color string

Marker outline color, specified as specified as one of these values:

- `'auto'` — Use the same color as the EdgeColor property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- `'flat'` — Use the CData value at the vertex to set the color.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | `'flat'` | RGB triplet or color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which allows the background to show through.

- `'auto'` — Use the same color as the `Color` property for the axes.
- `'flat'` — Use the CData value of the vertex to set the color.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

This property affects only the circle, square, diamond, pentagram, hexagram, and the four triangle marker types.

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example:

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: 10

Face and Vertex Normals

`FaceNormals` — Face normal vectors

m-by-n-by-3 array (default) | array of normal vectors

Face normal vectors, specified as an array of normal vectors with one normal vector one per patch face. Define one normal per patch face, as determined by the size of the Faces property value. Face normals determine the orientation of each patch face. This data is used for lighting calculations.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the patch generates this data when the axes contains light objects. The patch computes face normals using Newell's method.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`VertextNormals` — Vertex normal vectors

m-by-n-by-3 array (default) | array of normal vectors

Vertex normal vectors, specified as an array of normal vectors with one normal vector one per patch vertex. Define one normal per patch vertex, as determined by the size of the Vertices property value. Vertex normals determine the shape and orientation of the patch. This data is used for lighting calculations.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the patch generates this data when the axes contains light objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FaceNormalsMode` — Selection mode for `FaceNormals`

`'auto'` (default) | `'manual'`

Selection mode for `FaceNormals`, specified as one of these values:

- `'auto'` — The `patch` function calculates face normals when you add a light to the scene.
- `'manual'` — Use the face normal data specified by the FaceNormals property. Assigning values to the FaceNormals property sets FaceNormalsMode to `'manual'`.

`VertexNormalsMode` — Selection mode for `VertexNormals`

`'auto'` (default) | `'manual'`

Selection mode for `VertexNormals`, specified as one of these values:

- `'auto'` — The `patch` function calculates vertex normals when you add a light to the scene.

- `'manual'` — Use the vertex normal data specified by the VertexNormals property. Assigning values to the VertexNormals property sets VertexNormalsMode to `'manual'`.

Ambient Lighting

`AmbientStrength` — Strength of ambient light

`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light

`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection

`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections

`1` (default) | scalar between `0` and `1` inclusive

Color of specular reflections, specified as a scalar between `0` and `1` inclusive.

- `0` — The color of the specular reflection depends on both the color of the object from which it reflects and the color of the light source.
- `1` — The color of the specular reflection depends only on the color or the light source (that is, the light object `Color` property).

The contributions from the light source color and the patch color to the specular reflection color vary linearly for values between `0` and `1`.

Example: `0.5`

Data Types: `single | double`

`SpecularExponent` — Expansiveness of specular reflection

`10` (default) | scalar value greater than `0`

Expansiveness of specular reflection, specified as a scalar value greater than `0`. SpecularExponent controls the size of the specular reflection spot. Greater values produce less specular reflection.

Most materials have exponents in the range of `5` to `20`.

Example: `17`

Data Types: `double`

Color and Transparency Mapping

`FaceVertexAlphaData` — Face and vertex transparency data

`[ ]` (default) | single transparency value | column vector of transparency values

Face and vertex transparency data, specified in one of these forms:

- A single transparency value, which applies the same transparency to the entire patch. The FaceAlpha property must be set to `'flat'`.
- An m-by-1 array (where m is the number of rows in the Faces property), which specifies one transparency value per face. The FaceAlpha property must be set to `'flat'`.
- An m-by-1 array (where m is the number of rows in the Vertices property), which specifies one transparency value per vertex. The FaceAlpha property must be set to `'interp'`.

The AlphaDataMapping property determines how MATLAB interprets the FaceVertexAlphaData property values.

`AlphaDataMapping` — Transparency mapping method

`'scaled'` (default) | `'direct'` | `'none'`

Transparency mapping method, specified as one of these values:

- `'none'` — Clamps the elements of FaceVertexAlphaData to the region between `0` and `1`. A value of `1` or greater means completely opaque, a value of `0` or less means completely transparent, and a value between `0` and `1` means semitransparent.
- `'scaled'` — Transforms the FaceVertexAlphaData to span the portion of the figure alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `'direct'` — Uses the FaceVertexAlphaData as indices directly into the figure alphamap, which is determined by the figure `Alphamap` property. Values with a decimal portion are fixed to the nearest lower integer. MATLAB maps values less than 1 to the first value in the alphamap and values greater than `length(alphamap)` to the last value in the alphamap. If FaceVertexAlphaData is an array of `uint8` integers, then the indexing begins at 0 (that is, MATLAB maps a value of 0 to the first alpha value in the alphamap).

`FaceVertexCData` — Face and vertex colors

`[ ]` (default) | single color for entire patch | one color per face | one color per vertex

Face and vertex colors, specified as a single color for the entire patch, one color per face, or one color per vertex for interpolated face color.

If you want to use indexed colors, then specify FaceVertexCData in one of these forms:

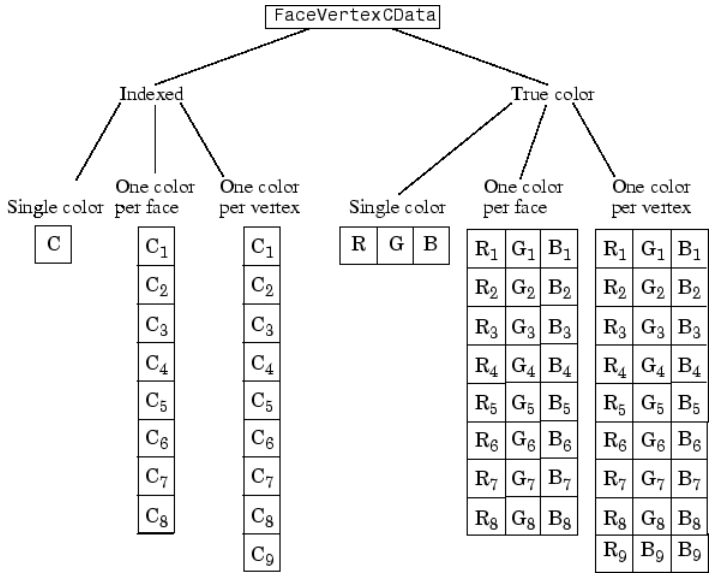
- For one color for the entire patch, use a single value.
- For one color per face, use an m-by-1 column vector, where m is the number of rows in the Faces property.
- For interpolated face color, use an m-by–1 column vector where m is the number of rows in the Vertices property.

If you want to use true colors, then specify FaceVertexCData in one of these forms:

- For one color for the entire patch, use a three-element row vector defining an RGB triplet.

- For one color per face, use an m-by-3 array of RBG triplets, where m is the number of rows in the Faces property.
- For interpolated face color, use an m-by-3 array, where m is the number of rows in the Vertices property.

The following diagram illustrates the various forms of the FaceVertexCData property for a patch having eight faces and nine vertices. The CDataMapping property determines how MATLAB interprets the FaceVertexCData property when you specify indexed colors.



`CData` — Patch color data

single color for entire patch | one color per face | one color per vertex

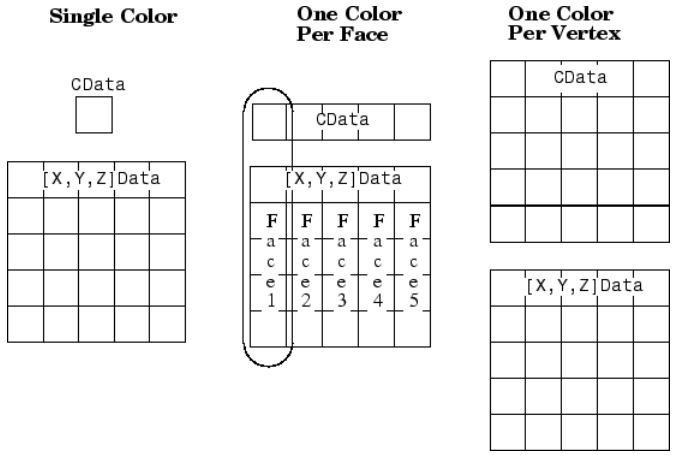
Patch color data, specified as a single color for the entire patch, one color per face, or one color per vertex.

The way the `patch` function interprets CData depends on the type of data supplied. Specify `CData` in one of these forms:

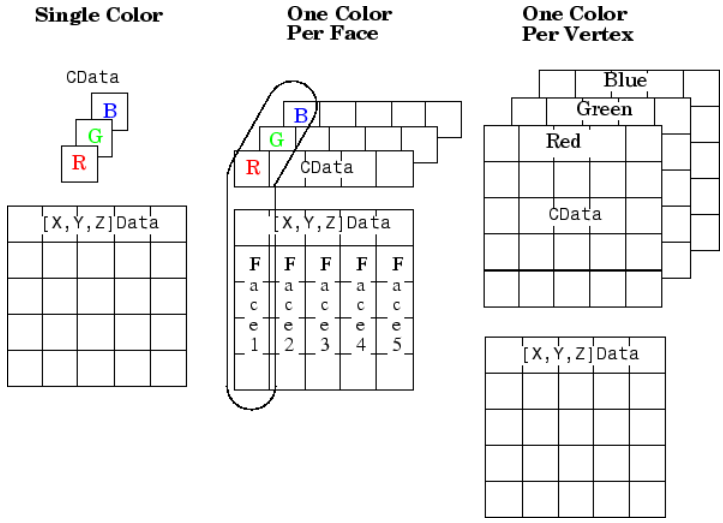
- Numeric values that are scaled to map linearly into the current colormap.
- Integer values that are used directly as indices into the current colormap.
- Arrays of RGB triplets. RGB triplets are not mapped into the current colormap, but interpreted as the colors defined.

The following diagrams illustrate the dimensions of `CData` with respect to the arrays in the `XData`, `YData`, and `ZData` properties.

These diagrams illustrates the use of indexed color.



These diagrams illustrates the use of true color. True color requires either a single RGB triplet or an array of RGB triplets.



If CData contains NaNs, then `patch` does not color the faces.

An alternative method for defining patches uses the `Faces`, `Vertices`, and `FaceVertexCData` properties.

Example: `[1,0,0]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CDataMapping` — Direct or scaled color data mapping

`'scaled'` (default) | `'direct'`

Direct or scaled color data mapping, specified as `'scaled'` (the default) or `'direct'`. The `CData` and FaceVertexCData properties contains color data. If you use true color specification for CData or FaceVertexCData, then this property has no effect.

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
  - If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
  - If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.
- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

Data

`XData` — x-coordinates of the patch vertices

vector | matrix

The *x*-coordinates of the patch vertices, specified as a vector or a matrix. If XData is a matrix, then each column represents the *x*-coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`YData` — y-coordinates of the patch vertices

vector | matrix

The *y*-coordinates defining the patch, specified as a vector or a matrix. If YData is a matrix, then each column represents the *y*-coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ZData` — z-coordinates of the patch vertices

vector | matrix

The *z*-coordinates of the patch vertices, specified as a vector or a matrix. If ZData is a matrix, then each column represents the *z*-coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

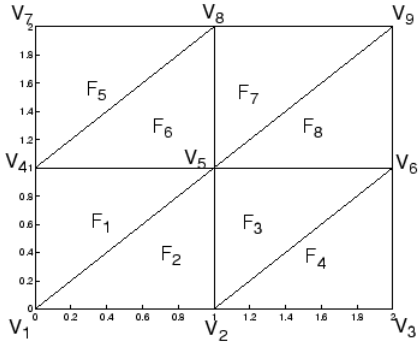
`Faces` — Vertex connection defining each face

vector | matrix

Vertex connection defining each face, specified as a vector or a matrix defining the vertices in the Vertices property that are to be connected to form each face. The Faces and Vertices properties provide an alternative way to specify a patch that can be more efficient than using XData, YData, and ZData coordinates in most cases.

Each row in the faces array designates the connections for a single face, and the number of elements in that row that are not `NaN` defines the number of vertices for that face. Therefore, an m-by-n Faces array defines m faces with up to n vertices each.

For example, consider the following patch. It is composed of eight triangular faces defined by nine vertices. The corresponding Faces and Vertices properties are shown to the right of the patch. Note how some faces share vertices with other faces. For example, the fifth vertex (`V5`) is used six times, once each by faces one, two, three, six, seven, and eight. Without sharing vertices, this same patch requires `24` vertex definitions.



Faces property

F1	V1	V4	V5
F2	V1	V5	V2
F3	V2	V5	V6
F4	V2	V6	V3
F5	V4	V7	V8
F6	V4	V8	V5
F7	V5	V8	V9
F8	V5	V9	V6

Vertices property

V1	X1	Y1	Z1
V2	X2	Y2	Z2
V3	X3	Y3	Z3
V4	X4	Y4	Z4
V5	X5	Y5	Z5
V6	X6	Y6	Z6
V7	X7	Y7	Z7
V8	X8	Y8	Z8
V9	X9	Y9	Z9

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Vertices` — Vertex coordinates

vector | matrix

Vertex coordinates, specified as a vector or a matrix defining the (*x*,*y*,*z*) coordinates of each vertex. The Faces and Vertices properties provide an alternative way to specify a patch that can be more efficient than using XData, YData, and ZData coordinates in most cases. See the `Faces` property for a description of how the vertex data is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Visibility

`Visible` — Visibility of patch

`'on'` (default) | `'off'`

Visibility of patch, specified as one of these values:

- `'on'` — Display the patch.
- `'off'` — Hide the patch without deleting it. You still can access the properties of an invisible patch object.

`Clipping` — Clipping of patch to axes limits

`'on'` (default) | `'off'`

Clipping of patch to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the patch that are outside the axes limits.
- `'off'` — Display the entire patch, even if parts of it appear outside the axes limits. Parts of the patch might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the patch that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the

printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'patch'`

Type of graphics object, returned as `'patch'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — User-specified tag

`''` (default) | string

Tag to associate with the patch, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with patch

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the patch object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the patch.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the patch object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the patch from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the patch object in the legend as one entry (default).
   - `'off'` — Do not include the patch object in the legend.
   - `'children'` — Include only children of the patch object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of patch

axes object | group object | transform object

Parent of patch, specified as an axes, group, or transform object.

`Children` — Children of patch

empty `GraphicsPlaceholder` array

The patch has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of patch object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The patch object handle is always visible.
- `'off'` — The patch object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The patch object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the patch at the command-line, but allows callback functions to access it.

If the patch object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle

- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the patch. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The patch object — You can access properties of the patch object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the patch. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the patch when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the patch.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the patch that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the patch responds to the click or if an ancestor does.
- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'` and you can click a part of the patch that has no color. The HitTest property determines if the patch responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the patch passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the patch. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the patch that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the patch object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the patch is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.

  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the patch tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the patch. Setting the `CreateFcn` property on an existing patch has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during patch creation. MATLAB executes the callback after creating the patch and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The patch object — You can access properties of the patch object from within the callback function. You also can access the patch object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the patch. MATLAB executes the callback before destroying the patch so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The patch object — You can access properties of the patch object from within the callback function. You also can access the patch object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of patch

`'off'` (default) | `'on'`

Deletion status of patch, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the patch begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the patch no longer exists.

Check the value of the BeingDeleted property to verify that the patch is not about to be deleted before querying or modifying it.

See Also

`patch`

More About

- "Access Property Values"
- "Graphics Object Properties"

path

View or change search path

Alternatives

As an alternative to the `path` function, use the Set Path dialog box.

Syntax

```
path
path('newpath')
path(path,'newpath')
path('newpath',path)
p = path
```

Description

`path` displays the MATLAB search path, which is stored in `pathdef.m`.

`path('newpath')` changes the search path to `newpath`, where `newpath` is a string array of folders.

`path(path,'newpath')` adds the `newpath` folder to the end of the search path. If `newpath` is already on the search path, then `path(path,'newpath')` moves `newpath` to the end of the search path.

`path('newpath',path)` adds the `newpath` folder to the top of the search path. If `newpath` is already on the search path, then `path('newpath',path)` moves `newpath` to the top of the search path. To add multiple folders in one statement, instead use `addpath`.

`p = path` returns the search path to string variable `p`.

Examples

Display the search path:

```
path
```

MATLAB returns, for example

```
MATLABPATH

 H:\My Documents\MATLAB
 C:\Program Files\MATLAB\R2ONnn\toolbox\matlab\general
 C:\Program Files\MATLAB\R2ONnn\toolbox\matlab\ops
 C:\Program Files\MATLAB\R2ONnn\toolbox\matlab\lang
 C:\Program Files\MATLAB\R2ONnn\toolbox\matlab\elmat
 C:\Program Files\MATLAB\R2ONnn\toolbox\matlab\elfun
...
```

`R2ONnn` represents the folder for the MATLAB release, for example, `R2009b`.

Add a new folder to the search path on Microsoft Windows platforms:

```
path(path,'c:/tools/goodstuff')
```

Add a new folder to the search path on UNIX[3] platforms:

```
path(path,'/home/tools/goodstuff')
```

Temporarily add the folder `my_files` to the search path, run `my_function` in `my_files`, then restore the previous search path:

```
p = path
path(p,'my_files')
my_function
path(p)
```

More About

- "What Is the MATLAB Search Path?"
- "Files and Folders that MATLAB Accesses"

See Also

`addpath` | `cd` | `dir` | `genpath` | `matlabroot` | `pathsep` | `pathtool` | `rehash` | `restoredefaultpath` | `rmpath` | `savepath` | `startup` | `userpath` | `what`

3. UNIX is a registered trademark of The Open Group in the United States and other countries.

Introduced before R2006a

path2rc

Save current search path to `pathdef.m` file

Syntax

```
path2rc
```

Description

`path2rc` runs `savepath`. The `savepath` function is replacing `path2rc`. Use `savepath` instead of `path2rc` and replace instances of `path2rc` with `savepath`.

Introduced before R2006a

pathsep

Search path separator for current platform

Syntax

```
c = pathsep
```

Description

`c = pathsep` returns the search path separator character for this platform. The search path separator is the character that separates path names in the `pathdef.m` file, as returned by the `path` function. The character is a semicolon (`;`). For versions of MATLAB software earlier than version 7.7 (R2008b), the character on UNIX[4] platforms was a colon (`:`). Use `pathsep` to work programmatically with the content of the search path file.

More About

- "What Is the MATLAB Search Path?"

See Also

`fileparts` | `filesep` | `fullfile` | `path`

Introduced before R2006a

4. UNIX is a registered trademark of The Open Group in the United States and other countries.

pathtool

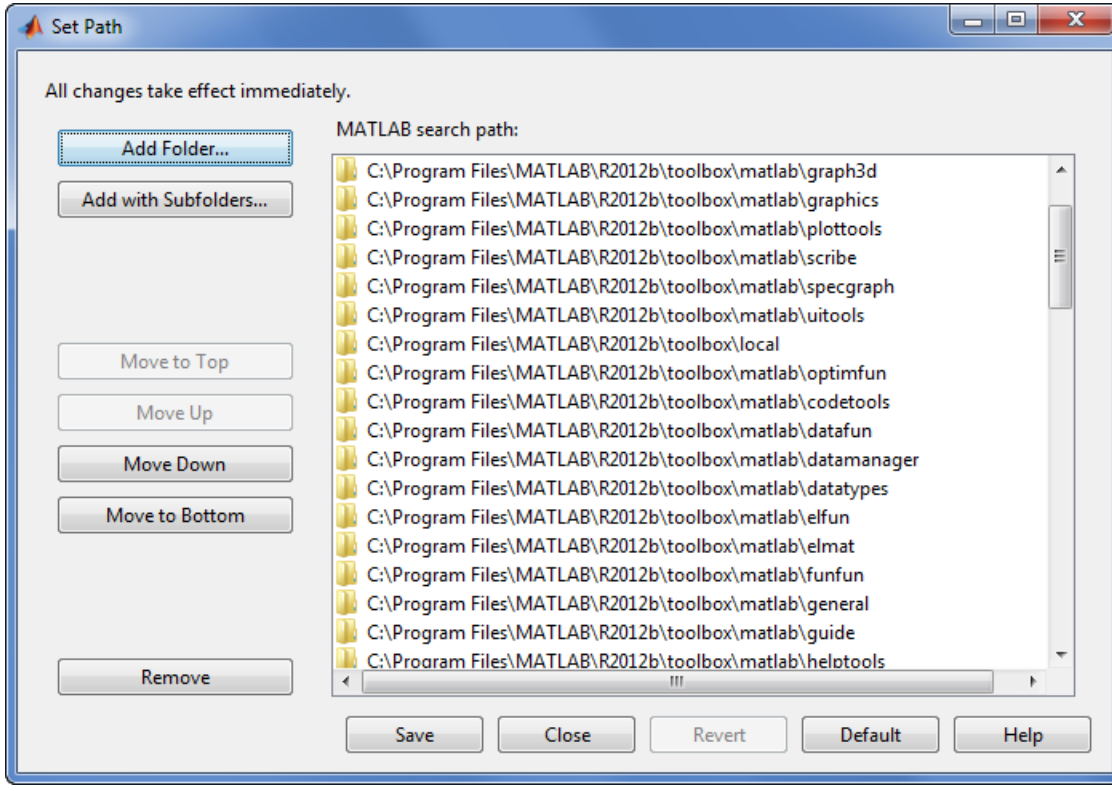
Open Set Path dialog box to view and change search path

Syntax

```
pathtool
```

Description

`pathtool` opens the Set Path dialog box, a graphical user interface you use to view and modify the MATLAB search path.



More About

- "What Is the MATLAB Search Path?"

See Also

`addpath` | `cd` | `dir` | `genpath` | `matlabroot` | `path` | `pathsep` | `rehash` | `restoredefaultpath` | `rmpath` | `savepath` | `startup` | `what`

Introduced before R2006a

pause

Halt execution temporarily

Syntax

```
pause
pause(n)
pause on
pause off
pause query
state = pause('query')
oldstate = pause(newstate)
```

Description

`pause`, by itself, causes the currently executing function to stop and wait for you to press any key before continuing. Pausing must be enabled for this to take effect. (See `pause on`, below). `pause` without arguments also blocks execution of Simulink models, but not repainting of them.

`pause(n)` pauses execution for `n` seconds before continuing, where `n` is any nonnegative real number. Pausing must be enabled for this to take effect.

Typing `pause(inf)` puts you into an infinite loop. To return to the MATLAB prompt, type **Ctrl+C**.

`pause on` enables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. Pausing remains enabled until you enter `pause off` in your function or at the command line.

`pause off` disables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. This allows normally interactive scripts to run unattended. Pausing remains disabled until you enter `pause on` in your function or at the command line, or start a new MATLAB session.

`pause query` displays `'on'` if pausing is currently enabled. Otherwise, it displays `'off'`.

`state = pause('query')` returns `'on'` in character array `state` if pausing is currently enabled. Otherwise, the value of `state` is `'off'`.

`oldstate = pause(newstate)`, enables or disables pausing, depending on the `'on'` or `'off'` value in `newstate`, and returns the former setting (also either `'on'` or `'off'`) in character array `oldstate`.

More About

Tips

The accuracy of `pause` is subject to the scheduling resolution of the operating system you are using, and also to other system activity. It cannot be guaranteed with 100% confidence. Asking for finer resolutions shows higher relative error.

While MATLAB is paused, the following continue to execute:

- Repainting of figure windows, Simulink block diagrams, and Java windows
- HG callbacks from figure windows
- Event handling from Java windows

See Also

`keyboard` | `input` | `drawnow`

Introduced before R2006a

pbaspect

Set or query plot box aspect ratio

Syntax

```
pbaspect
pbaspect([aspect_ratio])
pbaspect('mode')
pbaspect('auto')
pbaspect('manual')
pbaspect(axes_handle,...)
```

Description

The plot box aspect ratio determines the relative size of the *x*-, *y*-, and *z*-axes.

`pbaspect` with no arguments returns the plot box aspect ratio of the current axes.

`pbaspect([aspect_ratio])` sets the plot box aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the *x*-, *y*-, and *z*-axes size. For example, a value of `[1 1 1]` (the default) means the plot box is a cube (although with stretch-to-fill enabled, it may not appear as a cube). See Tips.

`pbaspect('mode')` returns the current value of the plot box aspect ratio mode, which can be either `auto` (the default) or `manual`. See Remarks.

`pbaspect('auto')` sets the plot box aspect ratio mode to `auto`.

`pbaspect('manual')` sets the plot box aspect ratio mode to `manual`.

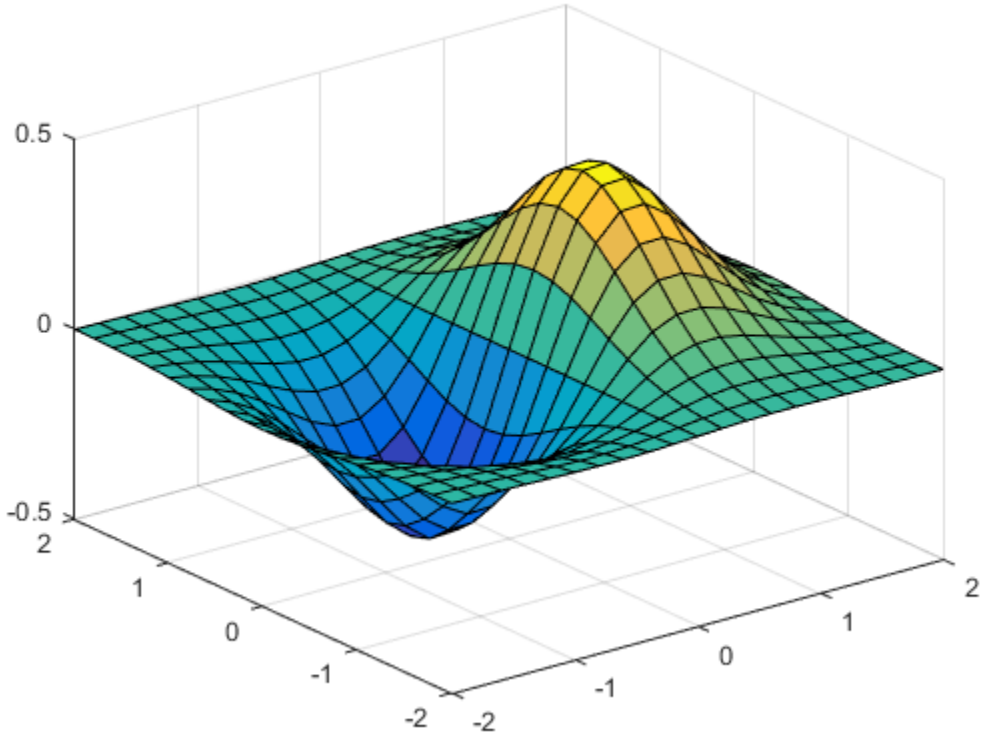
`pbaspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. If you do not specify an axes handle, `pbaspect` operates on the current axes.

Examples

Query Plot Box Aspect Ratio

Plot the function $z = xe^{(-x^2-y^2)}$ over the range $-2 \le x \le 2$ and $-2 \le y \le 2$.

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```



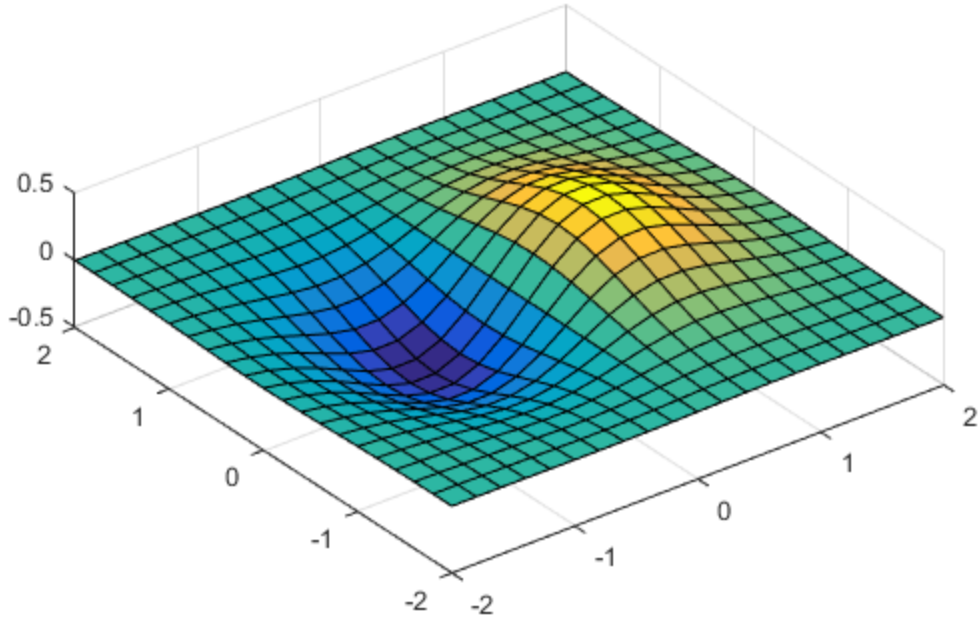
Query the plot box aspect ratio to show that the plot box is square.

```
plotboxaspect = pbaspect
```

```
plotboxaspect =

    1.0000    0.9419    0.8518
```

Change the data aspect ratio.

```
daspect([1,1,1])
```



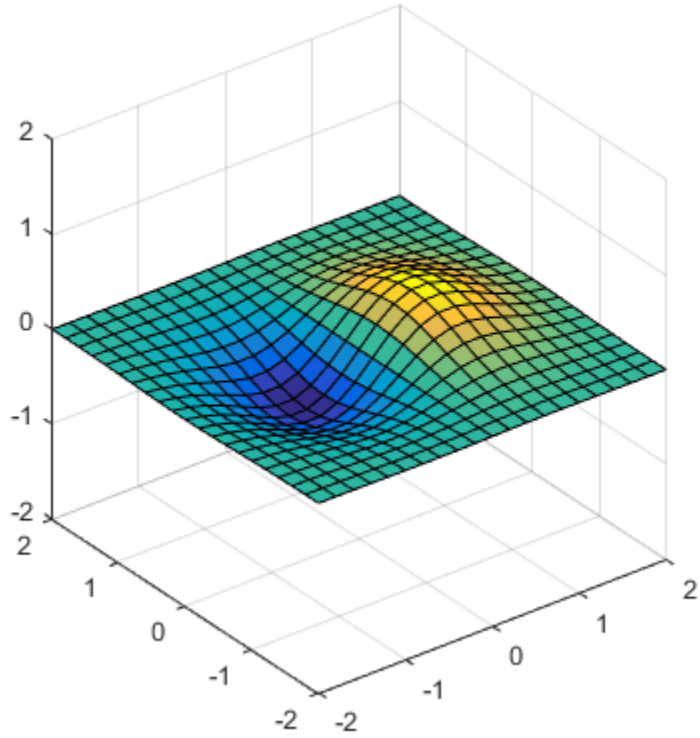
Query the plot box aspect ratio to show how it changes to accommodate the specified data aspect ratio.

```
plotboxaspect = pbaspect
```

```
plotboxaspect =

     4     4     1
```

Make the plot box square again by changing the plot box aspect ratio to `[1,1,1]`.

```
pbaspect([1,1,1])
```



More About

Tips

`pbaspect` sets or queries values of the axes object `PlotBoxAspectRatio` and `PlotBoxAspectRatioMode` properties.

When the plot box aspect ratio mode is `auto`, the MATLAB software sets the ratio to `[1 1 1]`, but may change it to accommodate manual settings of the data aspect ratio, camera view angle, or axis limits. See the axes `DataAspectRatio` property for a table listing the interactions between various properties.

Setting a value for the plot box aspect ratio or setting the plot box aspect ratio mode to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the plot box aspect ratio to its current value,

```
pbaspect(pbaspect)
```

can cause a change in the way the graphs look. See the Remarks section of the `axes` reference description, “Axes Aspect Ratio Properties” in the 3-D Visualization manual, and “Setting Aspect Ratio” in the MATLAB Graphics manual for a discussion of stretch-to-fill.

- Setting Aspect Ratio
- Axes Aspect Ratio Properties

See Also

`axis` | `daspect` | `xlim` | `ylim` | `zlim`

Introduced before R2006a

pcg

Preconditioned conjugate gradients method

Syntax

```
x = pcg(A,b)
pcg(A,b,tol)
pcg(A,b,tol,maxit)
pcg(A,b,tol,maxit,M)
pcg(A,b,tol,maxit,M1,M2)
pcg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = pcg(A,b,...)
[x,flag,relres] = pcg(A,b,...)
[x,flag,relres,iter] = pcg(A,b,...)
[x,flag,relres,iter,resvec] = pcg(A,b,...)
```

Description

`x = pcg(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric and positive definite, and should also be large and sparse. The column vector `b` must have length `n`. You also can specify `A` to be a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `pcg` converges, a message to that effect is displayed. If `pcg` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`pcg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `pcg` uses the default, `1e-6`.

`pcg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `pcg` uses the default, `min(n,20)`.

`pcg(A,b,tol,maxit,M)` and `pcg(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `pcg` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`pcg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `pcg` uses the default, an all-zero vector.

`[x,flag] = pcg(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`pcg` converged to the desired tolerance `tol` within `maxit` iterations.
1	`pcg` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`pcg` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `pcg` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = pcg(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = pcg(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = pcg(A,b,...)` also returns a vector of the residual norms at each iteration including `norm(b-A*x0)`.

Examples

Using pcg with Large Matrices

This example shows how to use pcg with a matrix input and with a function handle.

```
n1 = 21;
A = gallery('moler',n1);
b1 = sum(A,2);
tol = 1e-6;
maxit = 15;
M1 = spdiags((1:n1)',0,n1,n1);
[x1,flag1,rr1,iter1,rv1] = pcg(A,b1,tol,maxit,M1);
```

Alternatively, you can use the following function in place of the matrix A:

```
function y = applyMoler(x)
y = x;
y(end-1:-1:1) = y(end-1:-1:1) - cumsum(y(end:-1:2));
y(2:end) = y(2:end) - cumsum(y(1:end-1));
```

By using this function, you can solve larger systems more efficiently as there is no need to store the entire matrix A:

```
n2 = 21;
b2 = applyMoler(ones(n2,1));
tol = 1e-6;
maxit = 15;
M2 = spdiags((1:n2)',0,n2,n2);
[x2,flag2,rr2,iter2,rv2] = pcg(@applyMoler,b2,tol,maxit,M2);
```

Using pcg with a Preconditioner

This example demonstrates how to use a preconditioner matrix with pcg.

Create an input matrix and try to solve the system with pcg.

```
A = delsq(numgrid('S',100));
b = ones(size(A,1),1);
[x0,fl0,rr0,it0,rv0] = pcg(A,b,1e-8,100);
```

fl0 is 1 because pcg does not converge to the requested tolerance of 1e-8 within the requested maximum 100 iterations. A preconditioner can make the system converge more quickly.

Use ichol with only one input argument to construct an incomplete Cholesky factorization with zero fill.

```
L = ichol(A);
```

```
[x1,fl1,rr1,it1,rv1] = pcg(A,b,1e-8,100,L,L');
```

`fl1` is `0` because `pcg` drives the relative residual to `9.8e-09` (the value of `rr1`) which is less than the requested tolerance of `1e-8` at the seventy-seventh iteration (the value of `it1`) when preconditioned by the zero-fill incomplete Cholesky factorization. `rv1(1) = norm(b)` and `rv1(78) = norm(b-A*x1)`.

The previous matrix represents the discretization of the Laplacian on a 100x100 grid with Dirichlet boundary conditions. This means that a modified incomplete Cholesky preconditioner might perform even better.

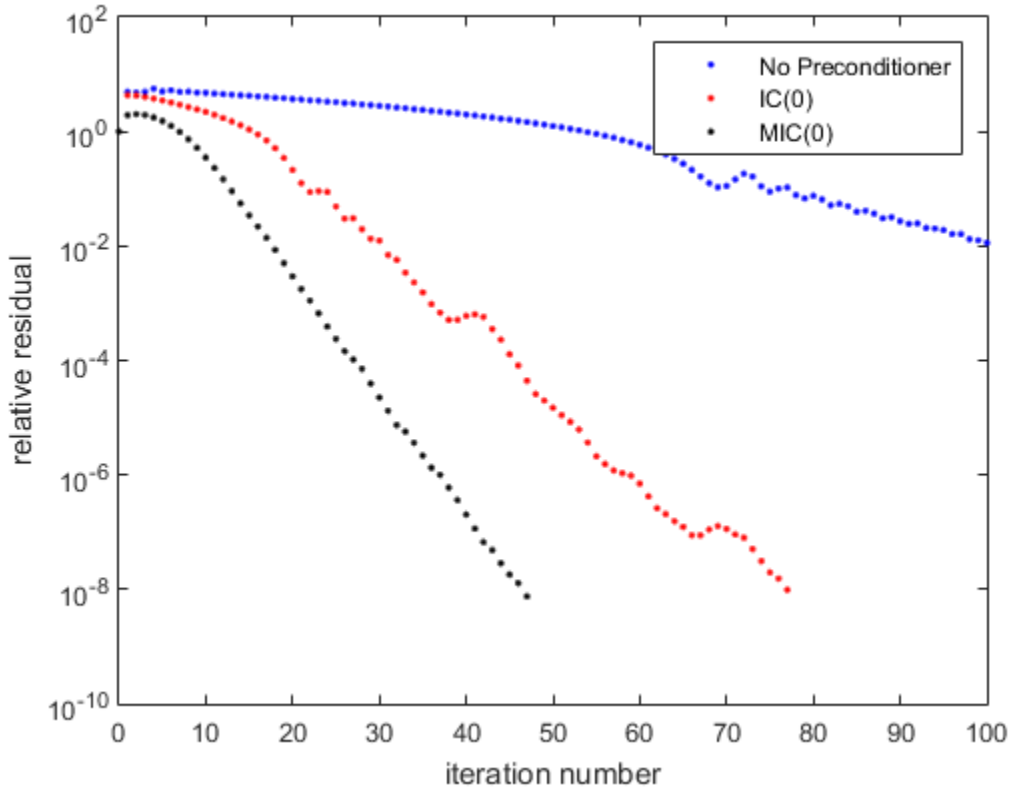
Use the `michol` option to create a modified incomplete Cholesky preconditioner.

```
L = ichol(A,struct('michol','on'));
[x2,fl2,rr2,it2,rv2] = pcg(A,b,1e-8,100,L,L');
```

In this case you attain convergence in only forty-seven iterations.

You can see how the preconditioners affect the rate of convergence of `pcg` by plotting each of the residual histories starting from the initial estimate (iterate number `0`).

```
figure;
semilogy(0:it0,rv0/norm(b),'b.');
hold on;
semilogy(0:it1,rv1/norm(b),'r.');
semilogy(0:it2,rv2/norm(b),'k.');
legend('No Preconditioner','IC(0)','MIC(0)');
xlabel('iteration number');
ylabel('relative residual');
hold off;
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

See Also

`bicg` | `bicgstab` | `cgs` | `function_handle` | `gmres` | `ichol` | `lsqr` | `minres` | `mldivide` | `qmr` | `symmlq`

Introduced before R2006a

pchip

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)

Syntax

```
yi = pchip(x,y,xi)
pp = pchip(x,y)
```

Description

`yi = pchip(x,y,xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by piecewise cubic interpolation within vectors `x` and `y`. The vector `x` specifies the points at which the data `y` is given, so `x` and `y` must have the same length. If `y` is a matrix or array, then the values in the last dimension, `y(:,...,:,j)`, are taken as the values to match with `x`. In that case, the last dimension of `y` must be the same length as `x`. If `y` has `n` dimensions, then output `yi` is of size `[size(y,1) size(y,2) ... size(y,n-1) length(xi)]`. For example, if `y` is a matrix, then `yi` is of size `[size(y,1) length(xi)]`.

`pp = pchip(x,y)` returns a piecewise polynomial structure for use by `ppval`. `x` can be a row or column vector. `y` is a row or column vector of the same length as `x`, or a matrix with `length(x)` columns.

`pchip` finds values of an underlying interpolating function $P(x)$ at intermediate points, such that:

- On each subinterval $x_k \le x \le x_{k+1}$, $P(x)$ is the cubic Hermite interpolant to the given values and certain slopes at the two endpoints.
- $P(x)$ interpolates y, i.e., $P(x_j) = y_j$, and the first derivative $\frac{dP}{dx}$ is continuous. The second derivative $\frac{d^2P}{dx^2}$ is probably not continuous; there may be jumps at the x_j.

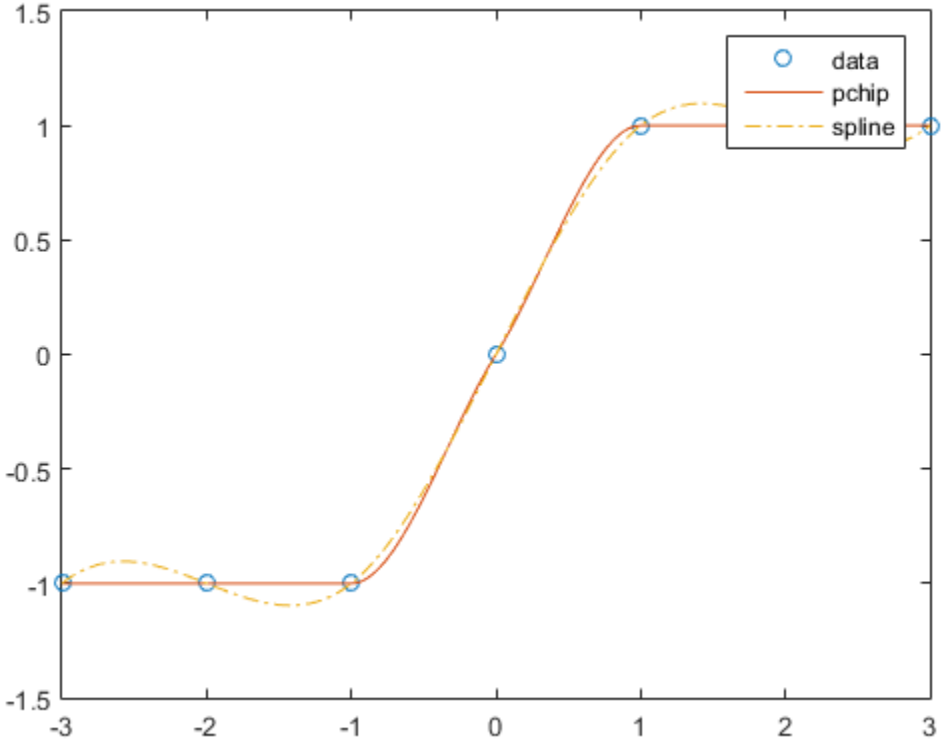
- The slopes at the x_j are chosen in such a way that $P(x)$ preserves the shape of the data and respects monotonicity. This means that, on intervals where the data are monotonic, so is $P(x)$; at points where the data has a local extremum, so does $P(x)$.

Note If y is a matrix, $P(x)$ satisfies the above for each column of y.

Examples

Data Interpolation Using `spline` and `pchip`

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
t = -3:.01:3;
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'o',t,p,'-',t,s,'-.')
legend('data','pchip','spline',4)
```



More About

Tips

`spline` constructs $S(x)$ in almost the same way `pchip` constructs $P(x)$. However, `spline` chooses the slopes at the x_j differently, namely to make even $S''(x)$ continuous. This has the following effects:

- `spline` produces a smoother result, i.e. $S''(x)$ is continuous.

- `spline` produces a more accurate result if the data consists of values of a smooth function.
- `pchip` has no overshoots and less oscillation if the data are not smooth.
- `pchip` is less expensive to set up.
- The two are equally expensive to evaluate.

References

[1] Fritsch, F. N. and R. E. Carlson, "Monotone Piecewise Cubic Interpolation," *SIAM J. Numerical Analysis*, Vol. 17, 1980, pp.238-246.

[2] Kahaner, David, Cleve Moler, Stephen Nash, *Numerical Methods and Software*, Prentice Hall, 1988.

See Also

`interp1` | `spline` | `ppval`

Introduced before R2006a

pcode

Create protected function file

Syntax

```
pcode(fun)
pcode(fun1,...,funN)
pcode(fun,'-inplace')
```

Description

`pcode(fun)` obfuscates the code in `fun.m` and produces a file called `fun.p`, known as a P-file. If `fun` is a folder, then all the script or function files in that folder are obfuscated in P-files. MATLAB creates the P-files in the current folder. The original `.m` file or folder can be anywhere on the search path.

`pcode(fun1,...,funN)` creates `N` P-files from the listed files. If any inputs are folders, then MATLAB creates a P-file for every `.m` file the folders contain.

`pcode(fun,'-inplace')` creates P-files in the same folder as the script or function files.

Note: The `pcode` function *obfuscates* your `.m` files, it does not *encrypt* them. While the content in a `.p` file is difficult to understand, it should not be considered secure. It is not recommended that you P-code files to protect your intellectual property.

Input Arguments

`fun`

MATLAB file or directory containing MATLAB files. If `fun` resides within a package and/or class folder, then `pcode` creates the same package and/or class structure to house the resulting P-files.

An input argument with no file extension and that is not a folder must be a function in the MATLAB path or in the current folder.

When using wild cards `*`, `pcode` ignores all files with extensions other than `.m`.

Default:

Examples

P-Coding Multiple Files

Convert selected files from the `sparfun` folder into P-files.

Create a temporary folder and define an existing path to `.m` files.

```
tmp = tempname;
mkdir(tmp)
cd(tmp)
fun = fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m');
```

Create the P-files.

```
pcode(fun)
dir(tmp)
```

```
.           ..          sprand.p     sprandn.p    sprandsym.p  sprank.p
```

The temporary folder now contains encoded P-files.

P-Coding Files That Belong to a Package and/or Class

Generate P-files from input files that are part of a package and/or class. This example uses an existing MATLAB example class.

Define `funclass` as an existing a class folder that contains `.m` files.

```
funclass = fullfile(docroot, 'techdoc', 'matlab_oop', ...
    'examples', '@BankAccount')
dir(funclass)
```

```
funclass =

C:\Program Files\MATLAB\R2013a\help\techdoc\matlab_oop\examples\@BankAccount
```

```
.            ..           BankAccount.m
```

Create a temporary folder. This folder has no package or class structure at this time.

```
tmp = tempname;
mkdir(tmp);
cd(tmp);
dir(tmp)
```

```
.            ..
```

Create a P-file for every `.m` file in the path `funclass`. Because the input files are part of a package and/or class, MATLAB creates a folder structure so that the output file belongs to the same package and/or class.

```
pcode(funclass)
dir(tmp)
```

```
.            ..            @BankAccount
```

You see that the P-file resides in the same folder structure.

```
dir('@BankAccount')
```

```
.            ..            BankAccount.p
```

P-Coding In Place

Generate P-files in the same folder as the input files using the option `inplace`

Copy several MATLAB files to a temporary folder.

```
fun = fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m');
tmp = tempname;
mkdir(tmp);
copyfile(fun,tmp)
dir(tmp)
```

```
.          ..          sprand.m     sprandn.m    sprandsym.m  sprank.m
```

Create P-files in the same folder as the original `.m` files.

```
pcode(tmp,'-inplace')
dir(tmp)
```

```
.            sprand.m     sprandn.m    sprandsym.m  sprank.m
```

```
..          sprand.p     sprandn.p    sprandsym.p  sprank.p
```

More About

Tips

- The `pcode` algorithm was redesigned in MATLAB 7.5 (Release R2007b). You can run older P-files in any current version of MATLAB; however, upcoming releases will not run P-files created before version 7.5. Files generated in 7.5, or later versions, cannot run in MATLAB 7.4 or earlier.
- When obfuscating all files in a folder, `pcode` does not obfuscate any files within subfolders.
- A P-file takes precedence over the corresponding `.m` file for execution, even after modifications to the `.m` file.
- MATLAB does not display any of the help comments that might be in the original `.m` file.
- "Protect Your Source Code"

See Also

`matlab.codetools.requiredFilesAndProducts`

Introduced before R2006a

pcolor

Pseudocolor (checkerboard) plot

Syntax

```
pcolor(C)
pcolor(X,Y,C)
pcolor(axes_handles,...)
h = pcolor(...)
```

Description

A pseudocolor plot is a rectangular array of cells with colors determined by `C`. MATLAB creates a pseudocolor plot using each set of four adjacent points in `C` to define a surface rectangle (i.e., cell).

The default `shading` is `faceted`, which colors each cell with a single color. The last row and column of `C` are not used in this case. With `shading interp`, each cell is colored by bilinear interpolation of the colors at its four vertices, using all elements of `C`.

The minimum and maximum elements of `C` are assigned the first and last colors in the colormap. Colors for the remaining elements in `C` are determined by a linear mapping from value to colormap element.

`pcolor(C)` draws a pseudocolor plot. The elements of `C` are linearly mapped to an index into the current colormap. The mapping from `C` to the current colormap is defined by `colormap` and `caxis`.

`pcolor(X,Y,C)` draws a pseudocolor plot of the elements of `C` at the locations specified by `X` and `Y`. The plot is a logically rectangular, two-dimensional grid with vertices at the points `[X(i,j), Y(i,j)]`. `X` and `Y` are vectors or matrices that specify the spacing of the grid lines. If `X` and `Y` are vectors, `X` corresponds to the columns of `C` and `Y` corresponds to the rows. If `X` and `Y` are matrices, they must be the same size as `C`.

`pcolor(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

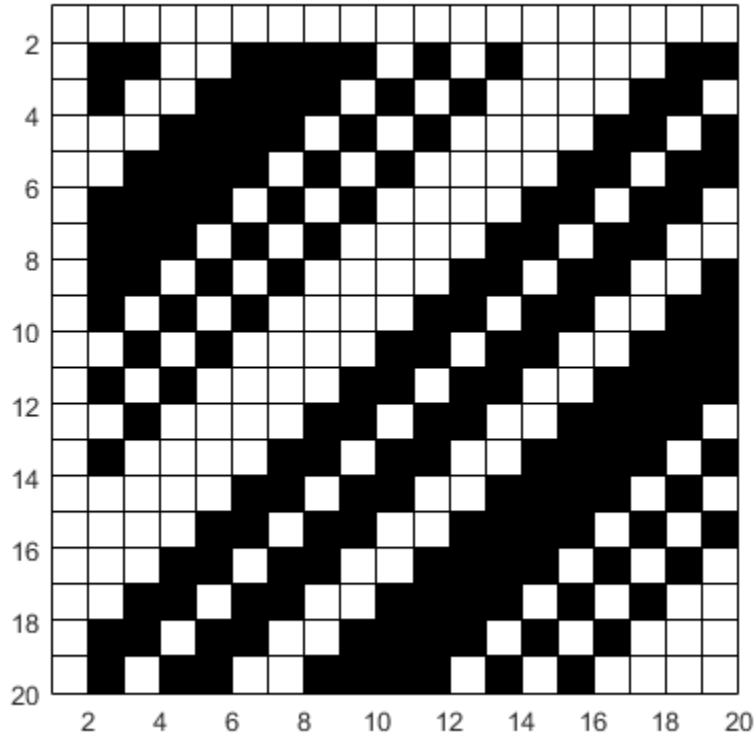
`h = pcolor(...)` returns a handle to a `surface` graphics object.

Examples

Pseudocolor Plot of Hadamard Matrix

A Hadamard matrix has elements that are `+1` and `-1`. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix.

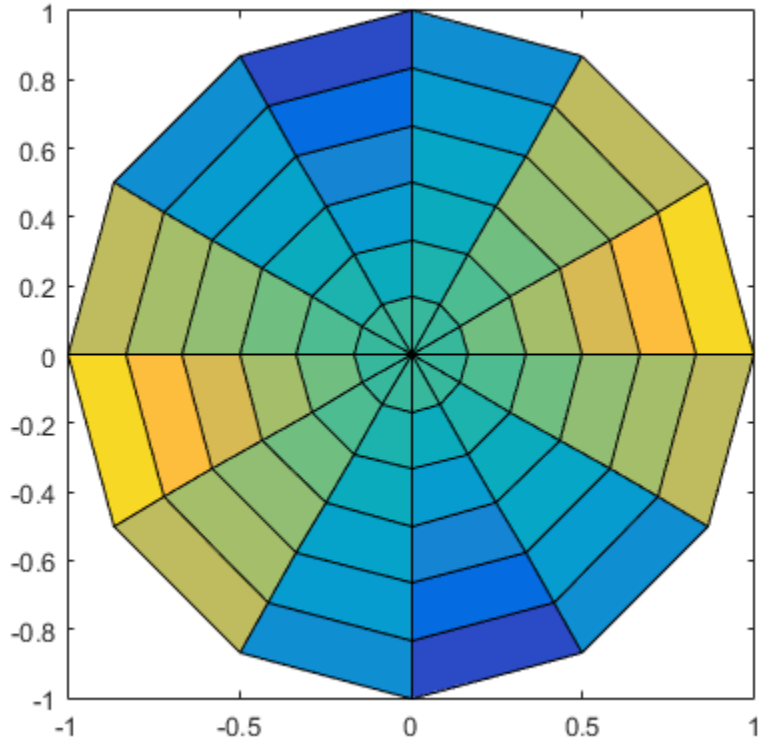
```
pcolor(hadamard(20))
colormap(gray(2))
axis ij
axis square
```



Polar Coordinate System

A simple color wheel illustrates a polar coordinate system.

```
n = 6;
r = (0:n)'/n;
theta = pi*(-n:n)/n;
X = r*cos(theta);
Y = r*sin(theta);
C = r*cos(2*theta);
pcolor(X,Y,C)
axis equal tight
```



More About

Tips

A pseudocolor plot is a flat surface plot viewed from above. `pcolor(X,Y,C)` is the same as viewing `surf(X,Y,zeros(size(X)),C)` using `view([0 90])`.

When you use `shading faceted` or `shading flat`, the constant color of each cell is the color associated with the corner having the smallest *x*-*y* coordinates. Therefore, `C(i,j)` determines the color of the cell in the *i*th row and *j*th column. The last row and column of `C` are not used.

When you use `shading interp`, each cell's color results from a bilinear interpolation of the colors at its four vertices, and all elements of `C` are used.

Algorithms

Use the `pcolor`, `image`, or `imagesc` function to display image data. Each function displays a rectangular array of cells and uses `C` to determine the colors.

- `pcolor(C)` uses the values in `C` to define the vertex colors by scaling the values to map to the full range of the colormap. The size of `C` determines the number of vertices. `pcolor` determines the cell colors using the colors defined at the cell vertices.
- `image(C)` uses `C` to define the cell colors by mapping the values directly into the colormap. The size of `C` determines the number of cells.
- `imagesc(C)` uses `C` to define the cell colors by scaling the values to map to the full range of the colormap. The size of `C` determines the number of cells.

`pcolor(X,Y,C)` can produce parametric grids, which is not possible with `image` or `imagesc`.

See Also

`caxis` | `image` | `mesh` | `shading` | `surf` | `view`

Introduced before R2006a

pdepe

Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D

Syntax

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
[sol,tsol,sole,te,ie] =
pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
```

Arguments

`m`	A parameter corresponding to the symmetry of the problem. `m` can be slab = `0`, cylindrical = `1`, or spherical = `2`.
`pdefun`	A handle to a function that defines the components of the PDE.
`icfun`	A handle to a function that defines the initial conditions.
`bcfun`	A handle to a function that defines the boundary conditions.
`xmesh`	A vector [`x0`, `x1`, ..., `xn`] specifying the points at which a numerical solution is requested for every value in `tspan`. The elements of `xmesh` must satisfy `x0 < x1 < ... < xn`. The length of `xmesh` must be >= `3`.
`tspan`	A vector [`t0`, `t1`, ..., `tf`] specifying the points at which a solution is requested for every value in `xmesh`. The elements of `tspan` must satisfy `t0 < t1 < ... < tf`. The length of `tspan` must be >= `3`.
`options`	Some options of the underlying ODE solver are available in `pdepe`: `RelTol`, `AbsTol`, `NormControl`, `InitialStep`, `MaxStep`, and `Events`. In most cases, default values for these options provide satisfactory solutions. See `odeset` for details.

Description

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)` solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time

t. `pdefun`, `icfun`, and `bcfun` are function handles. See the `function_handle` reference page for more information. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at times specified in `tspan`. The `pdepe` function returns values of the solution on a mesh provided in `xmesh`.

"Parameterizing Functions" explains how to provide additional parameters to the functions `pdefun`, `icfun`, or `bcfun`, if necessary.

`pdepe` solves PDEs of the form:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t}=x^{-m}\frac{\partial}{\partial x}\left(x^{m}f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right)+s\left(x,t,u,\frac{\partial u}{\partial x}\right)$$

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. The interval $[a,b]$ must be finite. m can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m > 0$, then a must be ≥ 0.

In Equation 1-4, $f\,(x,t,u,\partial u/\partial x)$ is a flux term and $s\,(x,t,u,\partial u/\partial x)$ is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c\,(x,t,u,\partial u/\partial x)$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if those values of x are mesh points. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

For $t = t_0$ and all x, the solution components satisfy initial conditions of the form

$$u(x,t_0)=u_0(x)$$

For all t and either $x = a$ or $x = b$, the solution components satisfy a boundary condition of the form

$$p(x,t,u)+q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right)=0$$

Elements of q are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux f rather than $\partial u/\partial x$. Also, of the two coefficients, only p can depend on u.

In the call `sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)`:

- `m` corresponds to m.
- `xmesh(1)` and `xmesh(end)` correspond to a and b.
- `tspan(1)` and `tspan(end)` correspond to t_0 and t_f.
- `pdefun` computes the terms c, f, and s (Equation 1-4). It has the form

  ```
  [c,f,s] = pdefun(x,t,u,dudx)
  ```

  The input arguments are scalars `x` and `t` and vectors `u` and `dudx` that approximate the solution u and its partial derivative with respect to x, respectively. `c`, `f`, and `s` are column vectors. `c` stores the diagonal elements of the matrix c (Equation 1-4).
- `icfun` evaluates the initial conditions. It has the form

  ```
  u = icfun(x)
  ```

  When called with an argument `x`, `icfun` evaluates and returns the initial values of the solution components at `x` in the column vector `u`.
- `bcfun` evaluates the terms p and q of the boundary conditions (Equation 1-6). It has the form

  ```
  [pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
  ```

  `ul` is the approximate solution at the left boundary `xl` = a and `ur` is the approximate solution at the right boundary `xr` = b. `pl` and `ql` are column vectors corresponding to p and q evaluated at `xl`, similarly `pr` and `qr` correspond to `xr`. When $m > 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. `pdepe` imposes this boundary condition automatically and it ignores values returned in `pl` and `ql`.

`pdepe` returns the solution as a multidimensional array `sol`. u_i = `ui` = `sol(:,:,i)` is an approximation to the `i`th component of the solution vector u. The element `ui(j,k)` = `sol(j,k,i)` approximates u_i at (t,x) = (`tspan(j)`,`xmesh(k)`).

`ui = sol(j,:,i)` approximates component `i` of the solution at time `tspan(j)` and mesh points `xmesh(:)`. Use `pdeval` to compute the approximation and its partial derivative $\partial u_i/\partial x$ at points not included in `xmesh`. See `pdeval` for details.

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)` solves as above with default integration parameters replaced by values in `options`, an argument created

with the odeset function. Only some of the options of the underlying ODE solver are available in pdepe: RelTol, AbsTol, NormControl, InitialStep, and MaxStep. The defaults obtained by leaving off the input argument options will generally be satisfactory. See odeset for details.

[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options) with the 'Events' property in options set to a function handle Events, solves as above while also finding where event functions g(t,u(x,t))are zero. For each function you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Three column vectors are returned by events: [value,isterminal,direction] = events(m,t,xmesh,umesh). xmesh contains the spatial mesh and umesh is the solution at the mesh points. Use pdeval to evaluate the solution between mesh points. For the I-th event function, value(i) is the value of the function, ISTERMINAL(I) = 1 if the integration is to terminate at a zero of this event function and 0 otherwise. direction(i) = 0 if all zeros are to be computed (the default), +1 if only zeros where the event function is increasing, and -1 if only zeros where the event function is decreasing. Output tsol is a column vector of times specified in tspan, prior to first terminal event. SOL(j,:,:) is the solution at T(j). TE is a vector of times at which events occur. SOLE(j,:,:) is the solution at TE(j) and indices in vector IE specify which event occurred.

If UI = SOL(j,:,i) approximates component i of the solution at time TSPAN(j) and mesh points XMESH, pdeval evaluates the approximation and its partial derivative $\partial u_i/\partial x$ at the array of points XOUT and returns them in UOUT and DUOUTDX: [UOUT,DUOUTDX] = PDEVAL(M,XMESH,UI,XOUT)

Note: The partial derivative $\partial u_i/\partial x$ is evaluated here rather than the flux. The flux is continuous, but at a material interface the partial derivative may have a jump.

Examples

Example 1. This example illustrates the straightforward formulation, computation, and plotting of the solution of a single PDE.

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial condition

$$u(x,0) = \sin \pi x$$

and boundary conditions

$$u(0,t) \equiv 0$$
$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0$$

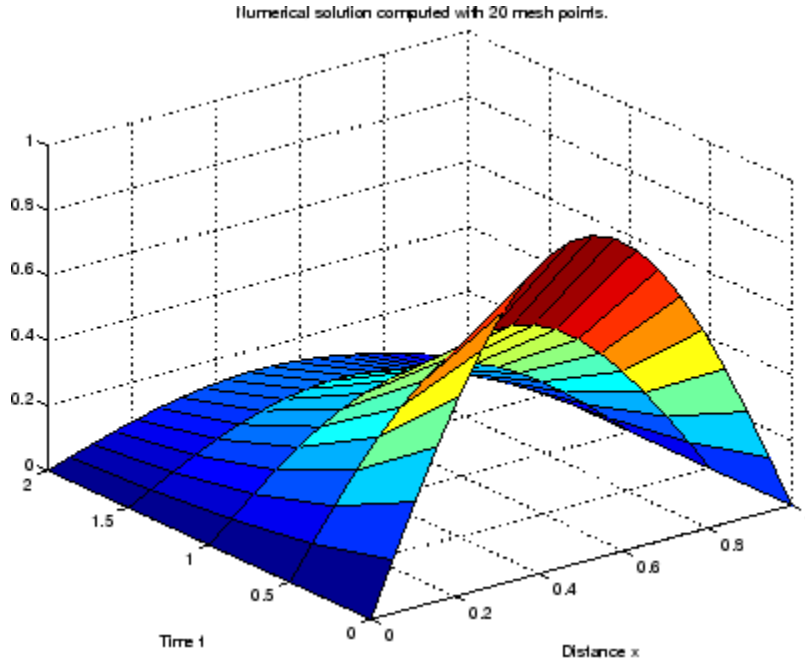
It is convenient to use local functions to place all the functions required by `pdepe` in a single function.

```
function pdex1

m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);

sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
% Extract the first solution component as u.
u = sol(:,:,1);

% A surface plot is often a good way to study a solution.
surf(x,t,u)
title('Numerical solution computed with 20 mesh points.')
xlabel('Distance x')
ylabel('Time t')

% A solution profile can also be illuminating.
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
% --------------------------------------------------------------
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
```

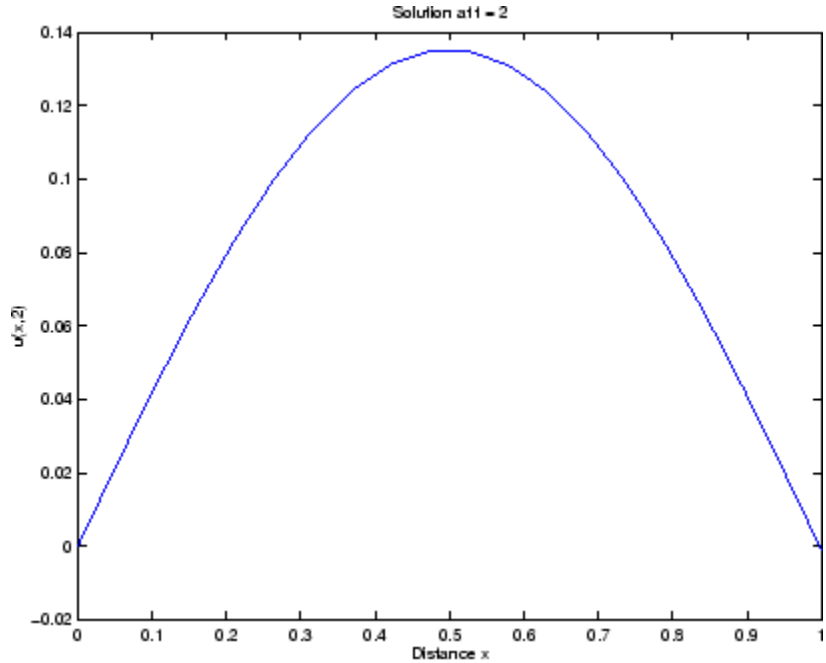
```
s = 0;
% --------------------------------------------------------------
function u0 = pdex1ic(x)
u0 = sin(pi*x);
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

In this example, the PDE, initial condition, and boundary conditions are coded in local functions `pdex1pde`, `pdex1ic`, and `pdex1bc`.

The surface plot shows the behavior of the solution.



The following plot shows the solution profile at the final value of `t` (i.e., `t = 2`).



Example 2. This example illustrates the solution of a system of PDEs. The problem has boundary layers at both ends of the interval. The solution changes rapidly for small t.

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024\frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$
$$\frac{\partial u_2}{\partial t} = 0.170\frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$.

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial conditions

$$u_1(x,0) \equiv 1$$
$$u_2(x,0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0,t) \equiv 0$$
$$u_2(0,t) \equiv 0$$
$$u_1(1,t) \equiv 1$$
$$\frac{\partial u_2}{\partial x}(1,t) \equiv 0$$

In the form expected by `pdepe`, the equations are

$$\begin{bmatrix}1\\1\end{bmatrix} .* \frac{\partial}{\partial t}\begin{bmatrix}u_1\\u_2\end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix}0.024(\partial u_1/\partial x)\\0.170(\partial u_2/\partial x)\end{bmatrix} + \begin{bmatrix}-F(u_1-u_2)\\F(u_1-u_2)\end{bmatrix}$$

The boundary conditions on the partial derivatives of u have to be written in terms of the flux. In the form expected by `pdepe`, the left boundary condition is

$$\begin{bmatrix}0\\u_2\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix} .* \begin{bmatrix}0.024(\partial u_1/\partial x)\\0.170(\partial u_2/\partial x)\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

and the right boundary condition is

$$\begin{bmatrix}u_1-1\\0\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix} .* \begin{bmatrix}0.024(\partial u_1/\partial x)\\0.170(\partial u_2/\partial x)\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

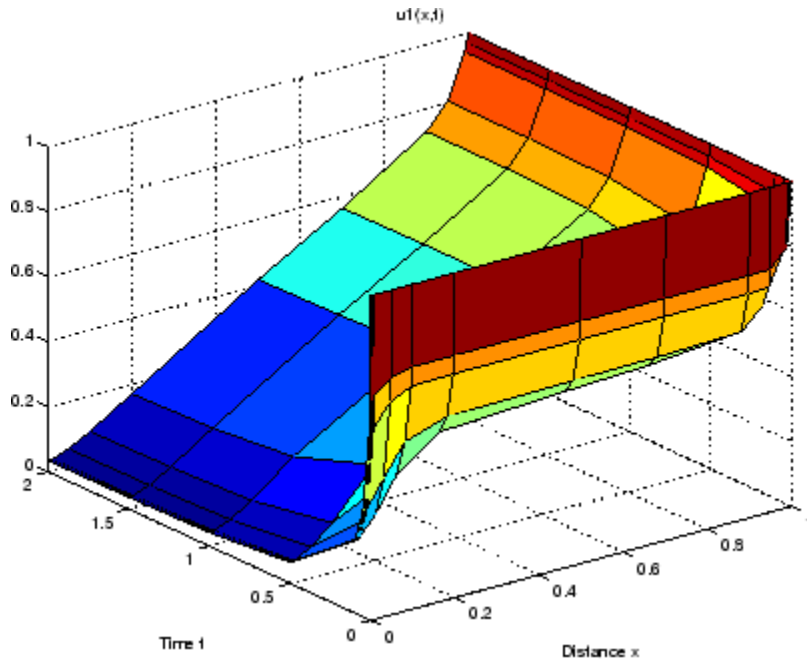
The solution changes rapidly for small t. The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, the example must select the output times accordingly. There are boundary layers in the solution at both ends of [0,1], so the example places mesh points near `0` and `1` to resolve these sharp changes. Often some experimentation is needed to select a mesh that reveals the behavior of the solution.

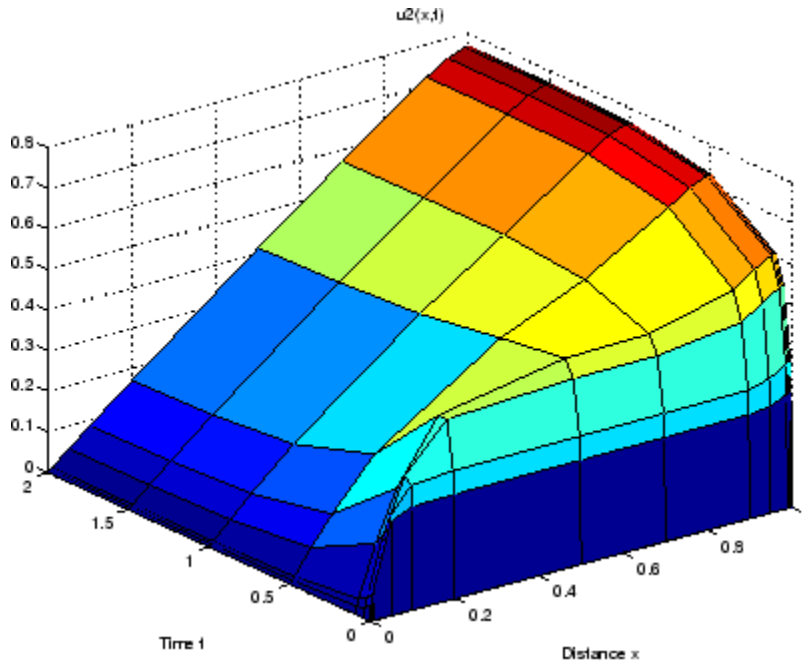
```
function pdex4
```

```
m = 0;
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
u1 = sol(:,:,1);
u2 = sol(:,:,2);

figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')

figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
% --------------------------------------------------------------
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
% --------------------------------------------------------------
function u0 = pdex4ic(x);
u0 = [1; 0];
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];
```

In this example, the PDEs, initial conditions, and boundary conditions are coded in local functions `pdex4pde`, `pdex4ic`, and `pdex4bc`.

The surface plots show the behavior of the solution components.


u1(x,t)
1
0.8
0.6
0.4
0.2
0
2
1.5
1
0.5
0
Time t
0
0.2
0.4
0.6
0.8
1
Distance x



More About

Tips

- The arrays `xmesh` and `tspan` play different roles in `pdepe`.

  `tspan` – The `pdepe` function performs the time integration with an ODE solver that selects both the time step and formula dynamically. The elements of `tspan` merely specify where you want answers and the cost depends weakly on the length of `tspan`.

  `xmesh` – Second order approximations to the solution are made on the mesh specified in `xmesh`. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. `pdepe` does *not* select the mesh in x automatically. You must provide an appropriate fixed mesh in `xmesh`. The cost depends strongly on the length of `xmesh`. When $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

- The time integration is done with `ode15s`. `pdepe` exploits the capabilities of `ode15s` for solving the differential-algebraic equations that arise when Equation 1-4 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.
- After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not "consistent" with the discretization, `pdepe` tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, `pdepe` can find consistent initial conditions close to the given ones. If `pdepe` displays a message that it has difficulty finding consistent initial conditions, try refining the mesh.

  No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

References

[1] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp.1–32.

See Also

`function_handle` | `pdeval` | `ode15s` | `odeset` | `odeget`

Introduced before R2006a

pdeval

Evaluate numerical solution of PDE using output of pdepe

Syntax

```
[uout,duoutdx] = pdeval(m,x,ui,xout)
```

Arguments

`m`	Symmetry of the problem: slab = `0`, cylindrical = `1`, spherical = `2`. This is the first input argument used in the call to `pdepe`.
`x`	A vector [`x0`, `x1`, ..., `xn`] specifying the points at which the elements of `ui` were computed. This is the same vector with which `pdepe` was called.
`ui`	A vector `sol(j,:,i)` that approximates component `i` of the solution at time t_f and mesh points `xmesh`, where `sol` is the solution returned by `pdepe`.
`xout`	A vector of points from the interval [`x0`,`xn`] at which the interpolated solution is requested.

Description

`[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution u_i and its partial derivative $\partial u_i/\partial x$ at points from the interval [`x0`,`xn`]. The `pdeval` function returns the computed values in `uout` and `duoutdx`, respectively.

Note `pdeval` evaluates the partial derivative $\partial u_i/\partial x$ rather than the flux f. Although the flux is continuous, the partial derivative may have a jump at a material interface.

See Also

`pdepe`

Introduced before R2006a

peaks

Example function of two variables

Syntax

```
Z = peaks;
Z = peaks(n);
Z = peaks(V);
Z = peaks(X,Y);
peaks(...)
[X,Y,Z] = peaks(...);
```

Description

`peaks` is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating `mesh`, `surf`, `pcolor`, `contour`, and so on.

`Z = peaks;` returns a 49-by-49 matrix.

`Z = peaks(n);` returns an `n`-by-`n` matrix.

`Z = peaks(V);` returns an `n`-by-`n` matrix, where `n = length(V)`.

`Z = peaks(X,Y);` evaluates `peaks` at the given `X` and `Y` (which must be the same size) and returns a matrix the same size.

`peaks(...)` (with no output argument) plots the peaks function with `surf`. Use any of the input argument combinations in the previous syntaxes.

`[X,Y,Z] = peaks(...);` returns two additional matrices, `X` and `Y`, for parametric plots, for example, `surf(X,Y,Z,del2(Z))`. If not given as input, the underlying matrices `X` and `Y` are

```
[X,Y] = meshgrid(V,V)
```

where V is a given vector, or V is a vector of length n with elements equally spaced from -3 to 3. If no input argument is given, the default n is 49.

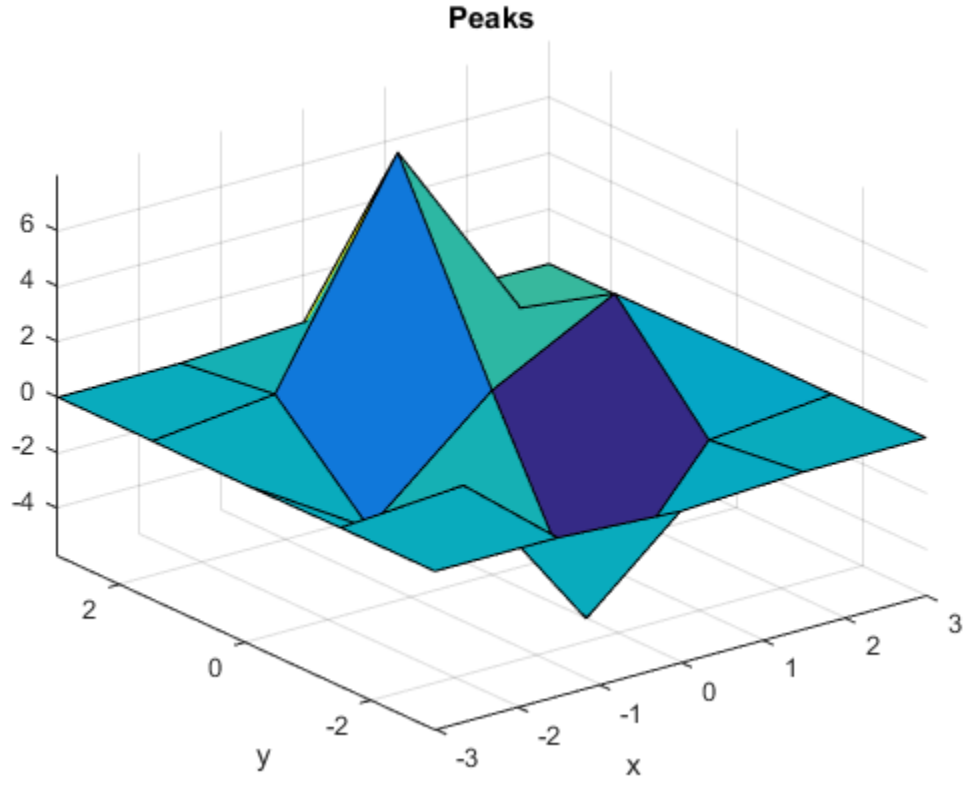
Examples

Peaks Surface

Create a 5-by-5 matrix of peaks and display the surface.

```
figure
peaks(5);
```

```
z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2)
```



See Also

`meshgrid` | `surf`

Introduced before R2006a

perl

Call Perl script using appropriate operating system executable

Syntax

```
perl('perlfile')
perl('perlfile',arg1,arg2,...)
result = perl(...)
[result, status] = perl(...)
```

Description

`perl('perlfile')` calls the Perl script `perlfile`, using the appropriate operating system Perl executable. Perl is included with the MATLAB software on Microsoft Windows systems, and thus MATLAB users can run user-created MATLAB functions containing the `perl` function. On Linux and Macintosh systems, MATLAB calls the Perl interpreter available with the operating system.

`perl('perlfile',arg1,arg2,...)` calls the Perl script `perlfile`, using the appropriate operating system Perl executable, and passes the arguments `arg1`, `arg2`, and so on, to `perlfile`.

`result = perl(...)` returns the results of attempted Perl call to `result`.

`[result, status] = perl(...)` returns the results of attempted Perl call to `result` and its exit status to `status`.

It is sometimes beneficial to use Perl scripts instead of MATLAB code. The `perl` function allows you to run those scripts from MATLAB. Specific examples where you might choose to use a Perl script include:

- Perl script already exists
- Perl script preprocesses data quickly, formatting it in a way more easily read by MATLAB
- Perl has features not supported by MATLAB

Examples

Given the Perl script, `hello.pl`:

```
$input = $ARGV[0];
print "Hello $input.";
```

At the MATLAB command line, type:

```
perl('hello.pl','World')
```

```
ans =
Hello World.
```

See Also

`dos` | `regexp` | `system` | `unix` | `!` (exclamation point)

Introduced before R2006a

perms

All possible permutations

Syntax

```
P = perms(v)
```

Description

`P = perms(v)` returns a matrix containing all permutations of the elements of vector `v` in reverse lexicographic order. Each row of `P` contains a different permutation of the n elements in `v`. Matrix `P` has the same data type as `v`, and it has $n!$ rows and n columns.

Examples

All Permutations of Double Integers

```
v = [2 4 6];
P = perms(v)
```

```
P =

     6     4     2
     6     2     4
     4     6     2
     4     2     6
     2     4     6
     2     6     4
```

All Permutations of Unsigned Integers

```
v = uint16([1023 4095 65535]);
P = perms(v)
```

```
P =

  65535   4095   1023
  65535   1023   4095
```

```
   4095  65535   1023
   4095   1023  65535
   1023   4095  65535
   1023  65535   4095
```

All Permutations of Complex Numbers

```
v = [1+1i 2+1i 3+1i];
P = perms(v)
```

```
P =

   3.0000 + 1.0000i   2.0000 + 1.0000i   1.0000 + 1.0000i
   3.0000 + 1.0000i   1.0000 + 1.0000i   2.0000 + 1.0000i
   2.0000 + 1.0000i   3.0000 + 1.0000i   1.0000 + 1.0000i
   2.0000 + 1.0000i   1.0000 + 1.0000i   3.0000 + 1.0000i
   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i
   1.0000 + 1.0000i   3.0000 + 1.0000i   2.0000 + 1.0000i
```

Input Arguments

v — Set of items

vector of numeric, `logical`, or `char` values

Set of items, specified as a vector of numeric, `logical`, or `char` values.

Example: `[1 2 3 4]`

Example: `[1+1i 2+1i 3+1i 4+1i]`

Example: `int16([1 2 3 4])`

Example: `['abcd']`

Example: `[true false true false]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Limitations

`perms(v)` is practical when `length(v)` is less than about 10.

See Also

`nchoosek` | `permute` | `randperm`

Introduced before R2006a

permute

Rearrange dimensions of N-D array

Syntax

```
B = permute(A,order)
```

Description

`B = permute(A,order)` rearranges the dimensions of `A` so that they are in the order specified by the vector `order`. `B` has the same values of `A` but the order of the subscripts needed to access any particular element is rearranged as specified by `order`. All the elements of `order` must be unique, real, positive, integer values.

Examples

Given any matrix `A`, the statement

```
permute(A,[2 1])
```

is the same as `A.'`.

For example:

```
A = [1 2; 3 4]; permute(A,[2 1])
ans =
     1     3
     2     4
```

The following code permutes a three-dimensional array:

```
X = rand(12,13,14);
Y = permute(X,[2 3 1]);
size(Y)
ans =
    13    14    12
```

More About

Tips

`permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

See Also

`ipermute` | `fliplr` | `flipud` | `circshift` | `reshape` | `shiftdim`

Introduced before R2006a

persistent

Define persistent variable

Syntax

```
persistent X Y Z
```

Description

`persistent X Y Z` defines `X`, `Y`, and `Z` as variables that are local to the function in which they are declared; yet their values are retained in memory between calls to the function. Persistent variables are similar to global variables because the MATLAB software creates permanent storage for both. They differ from global variables in that persistent variables are known only to the function in which they are declared. This prevents persistent variables from being changed by other functions or from the MATLAB command line.

Whenever you clear or modify a function that is in memory, MATLAB also clears all persistent variables declared by that function. To keep a function in memory until MATLAB quits, use `mlock`.

If the persistent variable does not exist the first time you issue the `persistent` statement, it is initialized to the empty matrix.

It is an error to declare a variable persistent if a variable with the same name exists in the current workspace. MATLAB also errors if you declare any of a function's input or output arguments as persistent within that same function. For example, the following persistent declaration is invalid:

```
function myfun(argA, argB, argC)
persistent argB
```

Examples

This function writes a large array to a spreadsheet file and then reads several rows from the same file. Because you only need to write the array to the spreadsheet one time, the

program tests whether an array can be read from the file and, if so, does not waste time in repeating that task. By defining the `dblArray` variable as persistent, you can easily check whether the array has been read from the spreadsheet file.

Here is the `arrayToXLS` function:

```
function arrayToXLS(A, xlsfile, x1, x2)
persistent dblArray;

if isempty(dblArray)
    disp 'Writing spreadsheet file ...'
    xlswrite(xlsfile, A);
end

disp 'Reading array from spreadsheet ...'
dblArray = xlsread(xlsfile, 'Sheet1', [x1 ':' x2])
fprintf('\n');
```

Run the function three times and observe the time elapsed for each run. The second and third run take approximately one tenth the time of the first run in which the function must create the spreadsheet:

```
largeArray = rand(4000, 200);

tic,  arrayToXLS(largeArray, 'myTest.xls','E254', 'J256'),  toc
Writing spreadsheet file ...
Reading array from spreadsheet ...
dblArray =
    0.0982    0.3783    0.1264    0.7880    0.1902    0.5811
    0.2251    0.2704    0.5682    0.7271    0.8028    0.2834
    0.6453    0.5568    0.8254    0.4961    0.9096    0.5402

Elapsed time is 8.990525 seconds.


tic,  arrayToXLS(largeArray, 'myTest.xls','E257', 'J258'),  toc
Reading array from spreadsheet ...
dblArray =
    0.4620    0.3781    0.6386    0.5930    0.0946    0.4865
    0.1605    0.1251    0.8709    0.5188    0.6702    0.2138

Elapsed time is 0.912534 seconds.
```

```
tic,  arrayToXLS(largeArray, 'myTest.xls','E259', 'J262'),  toc
Reading array from spreadsheet ...
dblArray =
    0.7015    0.6588    0.4023    0.0359    0.4512    0.6097
    0.1308    0.6441    0.0431    0.6396    0.7481    0.8688
    0.8278    0.2686    0.5475    0.8550    0.5896    0.1080
    0.9437    0.1671    0.0505    0.1203    0.2461    0.7306

Elapsed time is 0.928843 seconds.
```

Now clear the `arrayToXLS` function from memory and observe that running it takes much longer again:

```
clear functions

tic,  arrayToXLS(largeArray, 'myTest.xls','E263', 'J264'),  toc
Writing spreadsheet file ...
Reading array from spreadsheet ...
dblArray =
    0.6292    0.7788    0.0732    0.6481    0.9299    0.8631
    0.7700    0.5181    0.9805    0.5092    0.8658    0.4070

Elapsed time is 7.603461 seconds.
```

More About

Tips

There is no function form of the `persistent` command (i.e., you cannot use parentheses and quote the variable names).

See Also

`global` | `clear` | `mislocked` | `mlock` | `munlock` | `isempty`

Introduced before R2006a

pi

Ratio of circle's circumference to its diameter

Syntax

```
pi
```

Description

`pi` returns the floating-point number nearest the value of π. The expressions `4*atan(1)` and `imag(log(-1))` provide the same value.

Examples

Find the sine of π:

```
sin(pi)
```

returns

```
ans =

   1.2246e-16
```

The expression `sin(pi)` is not exactly zero because `pi` is not exactly π.

Introduced before R2006a

pie

Pie chart

Syntax

```
pie(X)
pie(X,explode)
pie(X,labels)
pie(X,explode,labels)

pie(ax, ___ )

p = pie( ___ )
```

Description

`pie(X)` draws a pie chart using the data in `X`. Each slice of the pie chart represents an element in `X`.

- If `sum(X) ≤ 1`, then the values in `X` directly specify the areas of the pie slices. `pie` draws only a partial pie if `sum(X) < 1`.
- If `sum(X) > 1`, then `pie` normalizes the values by `X/sum(X)` to determine the area of each slice of the pie.
- If `X` is of data type `categorical`, the slices correspond to categories. The area of each slice is the number of elements in the category divided by the number of elements in `X`.

`pie(X,explode)` offsets slices from the pie. `explode` is a vector or matrix of zeros and nonzeros that correspond to `X`. The `pie` function offsets slices for the nonzero elements only in `explode`.

If `X` is of data type `categorical`, then `explode` can be a vector of zeros and nonzeros corresponding to categories, or a cell array of the names of categories to offset.

`pie(X,labels)` specifies text labels for the slices. The number of labels must equal the number of slices. `X` must be numeric.

`pie(X,explode,labels)` specifies text labels for the slices. The number of labels must equal the number of slices.

`pie(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

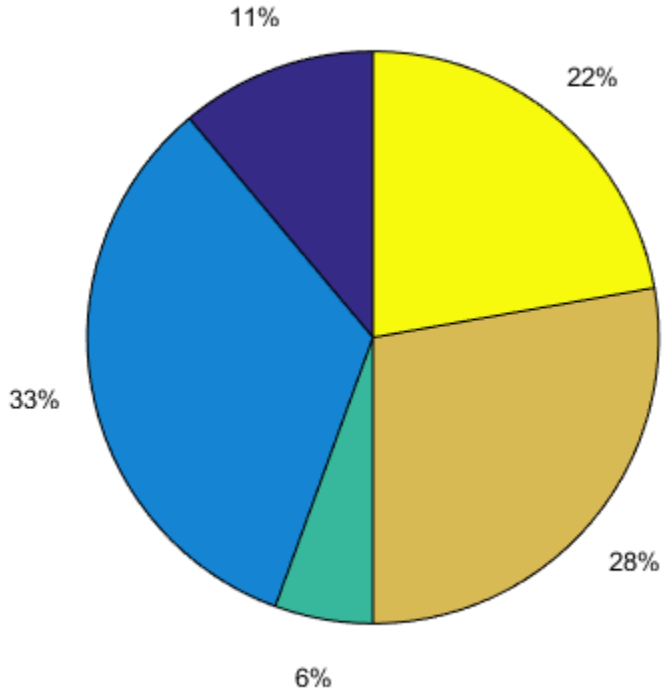
`p = pie( ___ )` returns a vector of patch and text graphics objects. The input can be any of the input argument combinations in the previous syntaxes.

Examples

Create Pie Chart with Offset Slices

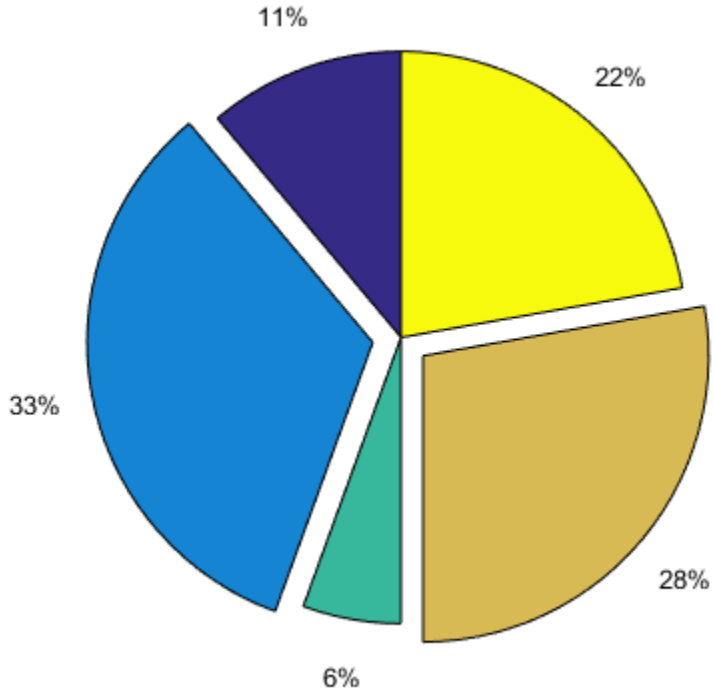
Create a pie chart of vector `X`.

```
X = [1 3 0.5 2.5 2];
pie(X)
```



Offset the second and fourth pie slices by setting the corresponding `explode` elements to 1.

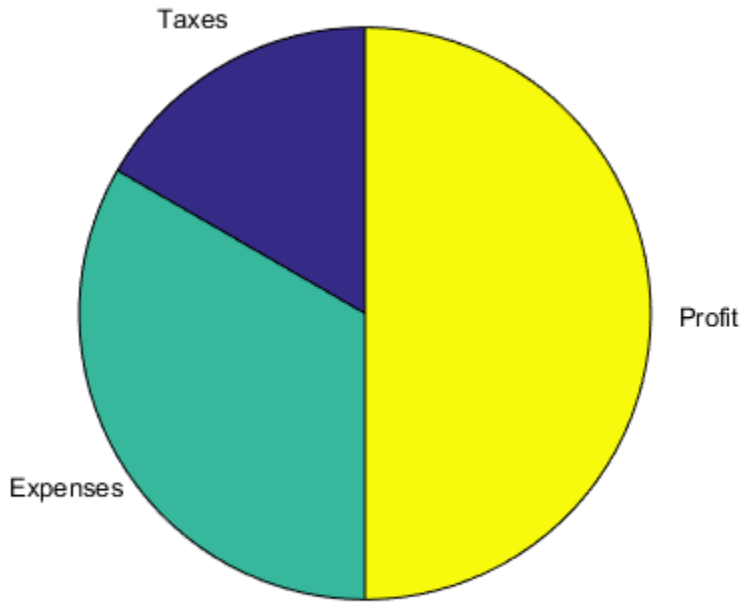
```
explode = [0 1 0 1 0];
pie(X,explode)
```



Specify Text Labels for Pie Chart

Create a pie chart of vector `X` and label the slices.

```
X = 1:3;
labels = {'Taxes','Expenses','Profit'};
pie(X,labels)
```

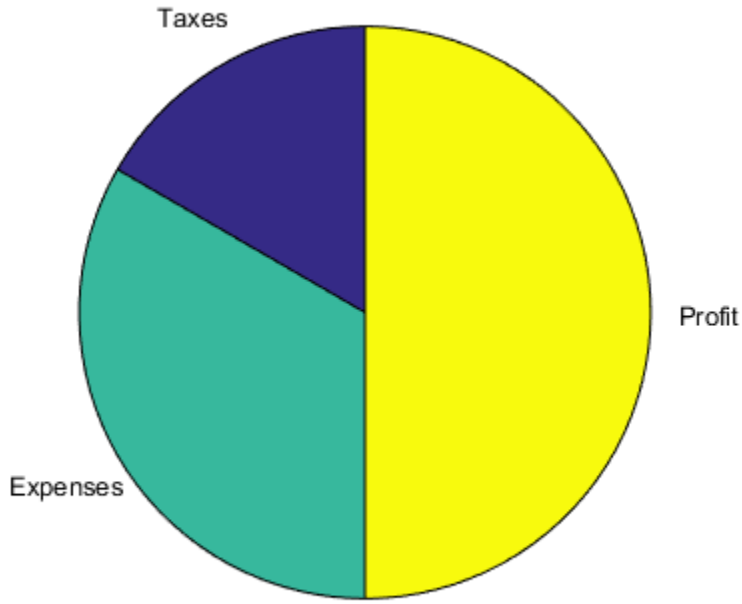


Modify Text Label for Pie Chart

Create a labeled pie chart, and then modify the color and font size of the text labels.

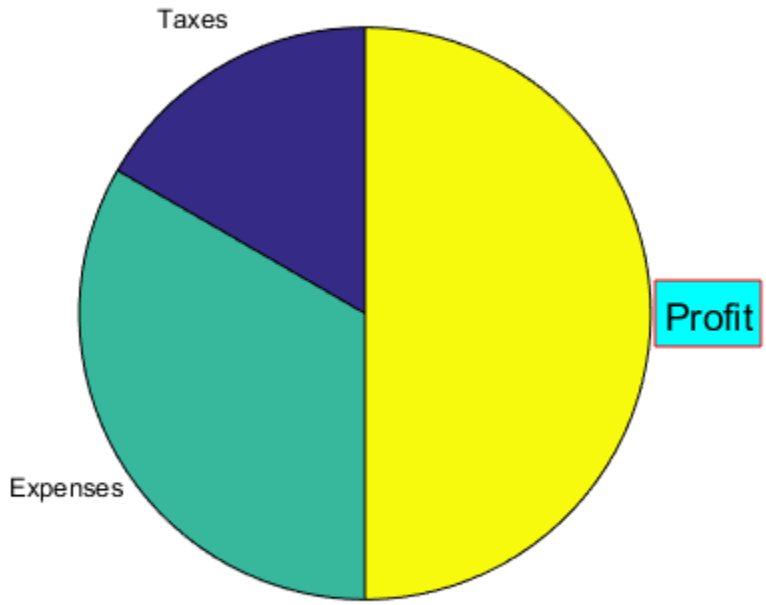
```
X = 1:3;
labels = {'Taxes','Expenses','Profit'};
p = pie(X,labels)
```

```
p =

  1x6 graphics array:

    Patch    Text     Patch    Text     Patch    Text
```



Get the text object for the label 'Profit'. Change its color and font size. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

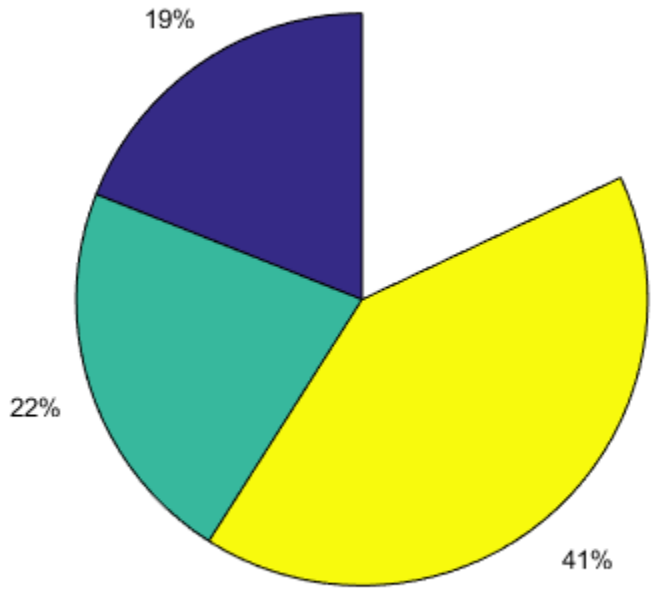
```
t = p(6);
t.BackgroundColor = 'cyan';
t.EdgeColor = 'red';
t.FontSize = 14;
```



Plot Partial Pie Chart

Create a pie chart of vector `X` where the sum of the elements is less than 1.

```
X = [0.19 0.22 0.41];
pie(X)
```



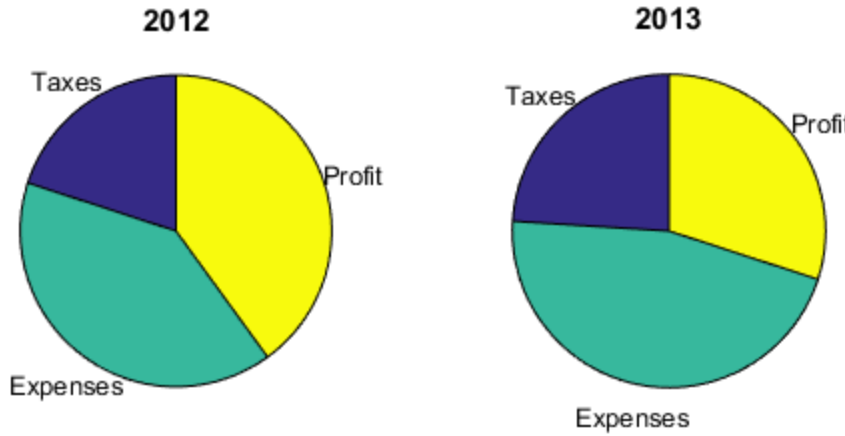
`pie` draws a partial pie because the sum of the elements is less than 1.

Plot Multiple Pie Charts

Create two vectors of data and plot and label each one in its own pie chart.

```
X = [0.2 0.4 0.4];
labels = {'Taxes','Expenses','Profit'};
ax1 = subplot(1,2,1);
pie(ax1,X,labels)
title(ax1,'2012');

Y = [0.24 0.46 0.3];
ax2 = subplot(1,2,2);
```

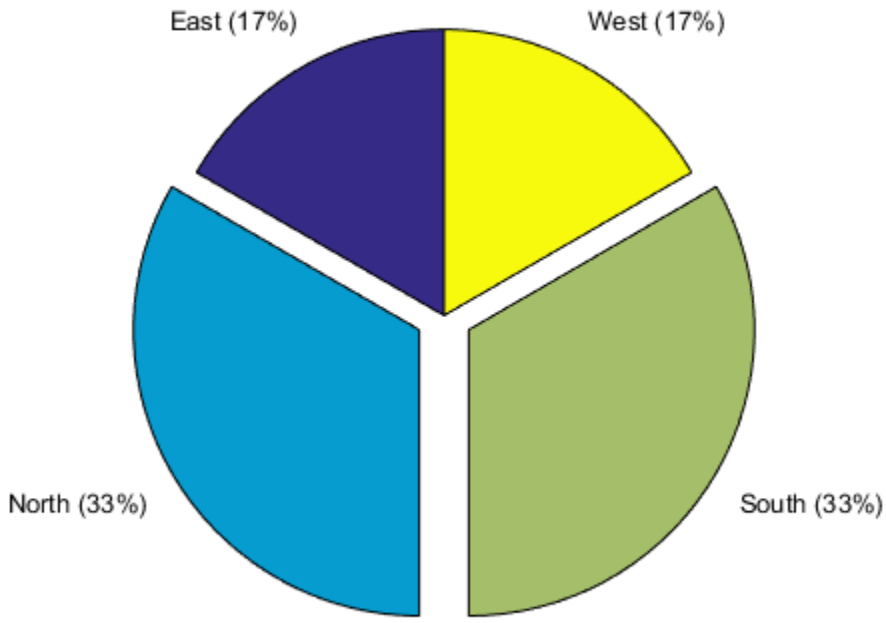
```
pie(ax2,Y,labels)
title(ax2,'2013');
```



Plot Categorical Pie Chart with Offsets

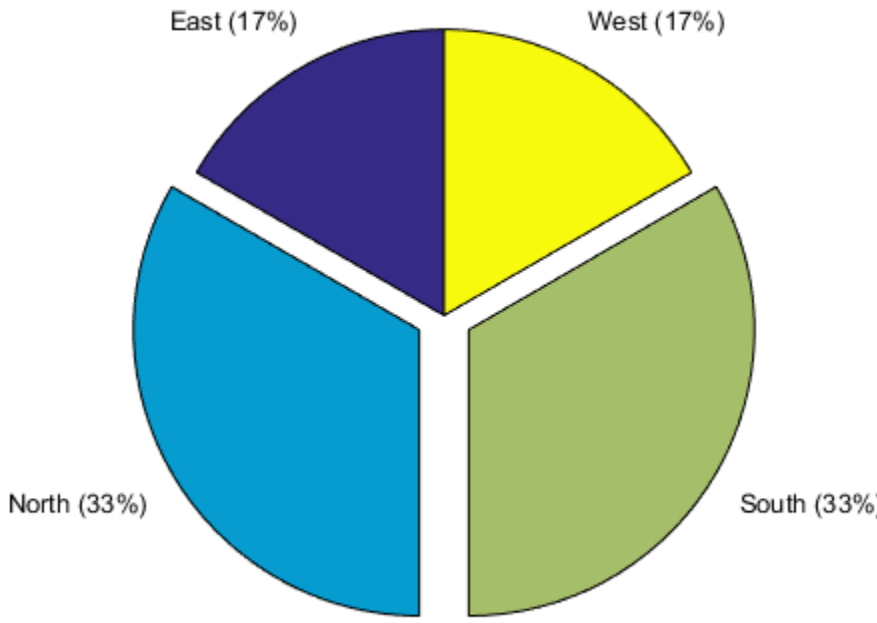
Plot a categorical pie chart with offset slices corresponding to categories.

```
X = categorical({'North','South','North','East','South','West'});
explode = {'North','South'};
pie(X,explode)
```



Now, use a logical vector to offset the same slices.

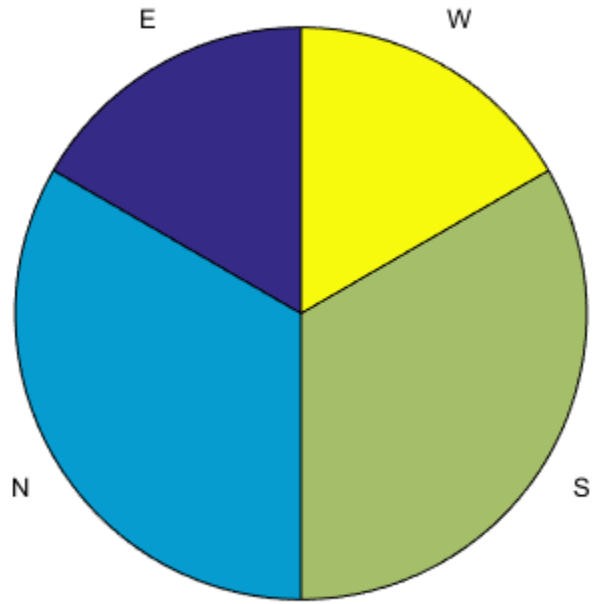
```
explode = [0 1 1 0];
pie(X,explode)
```



Plot Categorical Pie Chart with Labels

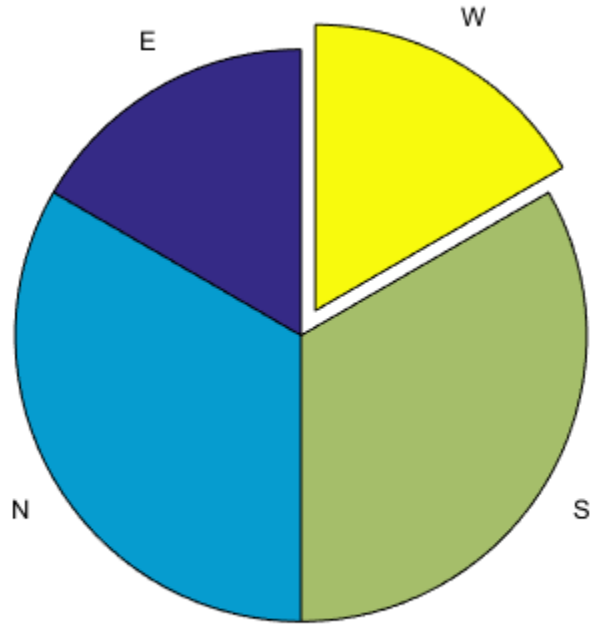
Plot a categorical pie chart without any offset slices and label the slices. When `X` is of data type `categorical` you must specify the input argument `explode`. To specify labels without any offset slices, specify `explode` as an empty cell array, and `labels` as the labels.

```
X = categorical({'North','South','North','East','South','West'});
explode = {};
labels = {'E','N','S','W'};
pie(X,explode,labels)
```



Now, offset a slice and label all slices.

```
X = categorical({'North','South','North','East','South','West'});
explode = {'West'};
labels = {'E','N','S','W'};
pie(X,explode,labels)
```



- “Offset Pie Slice with Greatest Contribution”
- “Label Pie Chart With Text and Percentages”

Input Arguments

X — Input array

vector or matrix

Input vector or matrix.

- If `X` is numeric, then all values in `X` must be finite. `pie` ignores nonpositive values.

- If `X` is categorical, then `pie` ignores undefined elements.

Data Types: `double|logical|categorical`

`explode` — Offset slices
vector or matrix

Offset slices, specified as a vector or matrix.

- If `X` is numeric, then `explode` must be a logical or numeric vector or matrix of zeros and nonzeros that correspond to `X`. A true (nonzero) value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.
- If `X` is categorical, then `explode` can be a cell array of strings that are category names. `pie` offsets slices corresponding to categories in `explode`.
- If `X` is categorical, then `explode` also can be a logical or numeric vector with elements that correspond to each category in `X`. The `pie` function offsets slices corresponding to true (nonzero) in category order.

`labels` — Text labels
cell array of strings

Text labels for slices, specified as a cell array of strings.

`ax` — Axes
axes object

Axes object. Use `ax` to plot the pie chart in a specific axes instead of the current axes (`gca`).

Output Arguments

`p` — Patch and text objects
vector

Patch and text objects, returned as a vector.

See Also
`pie3`

Introduced before R2006a

pie3

3-D pie chart

Syntax

```
pie3(X)
pie3(X,explode)
pie3(...,labels)
pie3(axes_handle,...)
h = pie3(...)
```

Description

`pie3(X)` draws a three-dimensional pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

- If `sum(X) ≤ 1`, then the values in `X` directly specify the area of the pie slices. `pie3` draws only a partial pie if `sum(X) < 1`.
- If the sum of the elements in `X` is greater than one, then `pie3` normalizes the values by `X/sum(X)` to determine the area of each slice of the pie.

`pie3(X,explode)` specifies whether to offset a slice from the center of the pie chart. `X(i,j)` is offset from the center of the pie chart if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie3(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`.

`pie3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

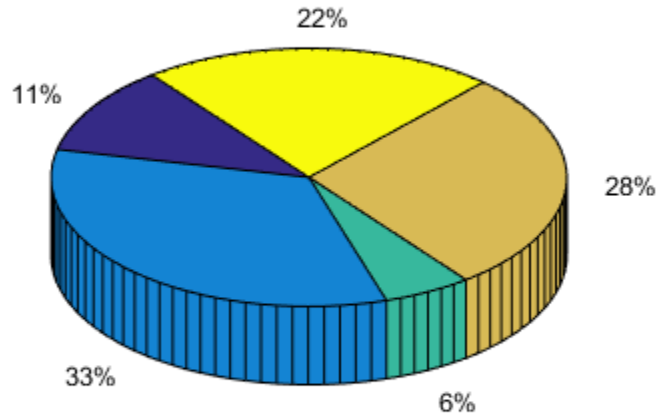
`h = pie3(...)` returns a vector of handles to patch, surface, and text graphics objects.

Examples

Create 3-D Pie Chart

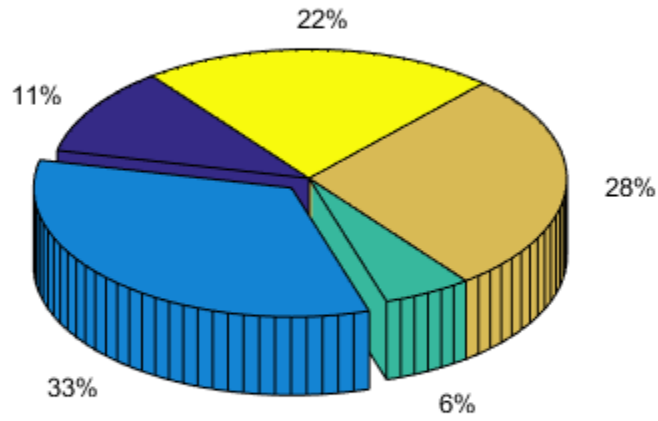
Create a 3-D pie chart of vector `x`.

```
x = [1,3,0.5,2.5,2];
figure
pie3(x)
```



To offset the second pie slice, set the corresponding `explode` element to 1.

```
explode = [0,1,0,0,0];
figure
pie3(x,explode)
```

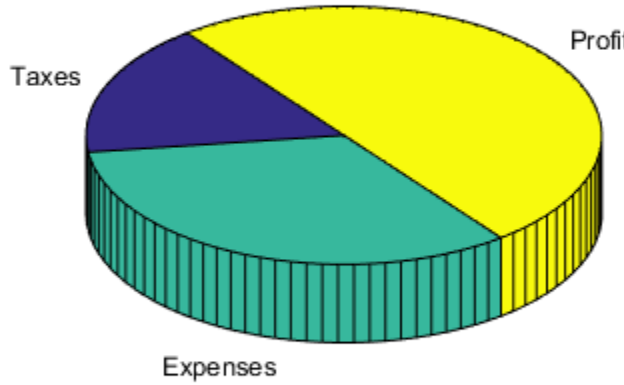


Specify Text Labels for 3-D Pie Chart

Create a 3-D pie chart and specify the text labels.

```
x = 1:3;
labels = {'Taxes','Expenses','Profit'};

figure
```

```
pie3(x,labels)
```



See Also

`pie`

Introduced before R2006a

pinv

Moore-Penrose pseudoinverse of matrix

Syntax

```
B = pinv(A)
B = pinv(A,tol)
```

Definitions

The Moore-Penrose pseudoinverse is a matrix `B` of the same dimensions as `A'` satisfying four conditions:

```
A*B*A = A
B*A*B = B
A*B is Hermitian
B*A is Hermitian
```

The computation is based on `svd(A)` and any singular values less than `tol` are treated as zero.

Description

`B = pinv(A)` returns the Moore-Penrose pseudoinverse of `A`.

`B = pinv(A,tol)` returns the Moore-Penrose pseudoinverse and overrides the default tolerance, `max(size(A))*norm(A)*eps`.

Examples

If `A` is square and not singular, then `pinv(A)` is an expensive way to compute `inv(A)`. If `A` is not square, or is square and singular, then `inv(A)` does not exist. In these cases, `pinv(A)` has some of, but not all, the properties of `inv(A)`.

If A has more rows than columns and is not of full rank, then the overdetermined least squares problem

```
minimize norm(A*x-b)
```

does not have a unique solution. Two of the infinitely many solutions are

```
x = pinv(A)*b
```

and

```
y = A\b
```

These two are distinguished by the facts that `norm(x)` is smaller than the norm of any other solution and that `y` has the fewest possible nonzero components.

For example, the matrix generated by

```
A = magic(8); A = A(:,1:6)
```

is an 8-by-6 matrix that happens to have `rank(A) = 3`.

```
A =
    64     2     3    61    60     6
     9    55    54    12    13    51
    17    47    46    20    21    43
    40    26    27    37    36    30
    32    34    35    29    28    38
    41    23    22    44    45    19
    49    15    14    52    53    11
     8    58    59     5     4    62
```

The right-hand side is `b = 260*ones(8,1)`,

```
b =
     260
     260
     260
     260
     260
     260
     260
     260
```

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to `A*x = b` would be a vector of all 1's. With only six columns, the equations are still consistent,

so a solution exists, but it is not all 1's. Since the matrix is rank deficient, there are infinitely many solutions. Two of them are

```
x = pinv(A)*b
```

which is

```
x =
    1.1538
    1.4615
    1.3846
    1.3846
    1.4615
    1.1538
```

and

```
y = A\b
```

which produces this result.

```
Warning: Rank deficient, rank = 3  tol =   1.8829e-013.
y =
    4.0000
    5.0000
         0
         0
         0
   -1.0000
```

Both of these are exact solutions in the sense that `norm(A*x-b)` and `norm(A*y-b)` are on the order of roundoff error. The solution `x` is special because

```
norm(x) = 3.2817
```

is smaller than the norm of any other solution, including

```
norm(y) = 6.4807
```

On the other hand, the solution `y` is special because it has only three nonzero components.

See Also

`inv` | `qr` | `rank` | `svd`

Introduced before R2006a

planerot

Givens plane rotation

Syntax

```
[G,y] = planerot(x)
```

Description

`[G,y] = planerot(x)` where `x` is a 2-component column vector, returns a 2-by-2 orthogonal matrix G so that `y = G*x` has `y(2) = 0`.

Examples

```
x = [3 4];
[G,y] = planerot(x')

G  =
    0.6000    0.8000
   -0.8000    0.6000

y =
     5
     0
```

See Also

`qrdelete` | `qrinsert`

Introduced before R2006a

play

Play audio from `audioplayer` object

Syntax

```
play(playerObj)
play(playerObj,start)
play(playerObj,[start,stop])
```

Description

`play(playerObj)` plays the audio associated with `audioplayer` object `playerObj` from beginning to end.

`play(playerObj,start)` plays audio from the sample indicated by `start` to the end.

`play(playerObj,[start,stop])` plays audio from the sample indicated by `start` to the sample indicated by `stop`.

Examples

Play with and without Blocking

Play two audio samples with and without blocking using the `play` and `playblocking` methods.

Load data from example files `chirp.mat` and `gong.mat`.

```
chirpData = load('chirp.mat');
chirpObj = audioplayer(chirpData.y,chirpData.Fs);

gongData = load('gong.mat');
gongObj = audioplayer(gongData.y,gongData.Fs);
```

Play the samples with blocking, one after the other.

```
playblocking(chirpObj);
```

```
playblocking(gongObj);
```

Play without blocking. The audio can overlap.

```
play(chirpObj);
play(gongObj);
```

Starting Sample

Play audio from the example file `handel.mat` starting 4 seconds from the beginning.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = playerObj.SampleRate * 4;

play(playerObj,start);
```

Sample Range

Play the first 3 seconds of audio from the example file `handel.mat`.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = 1;
stop = playerObj.SampleRate * 3;

play(playerObj,[start,stop]);
```

See Also

`audioplayer` | `playblocking`

How To

- "Play Audio"

play

Play audio from `audiorecorder` object

Syntax

```
player = play(recObj)
player = play(recObj, start)
player = play(recObj, [start stop])
```

Description

`player = play(recObj)` plays the audio associated with `audiorecorder` object `recObj` from beginning to end, and returns an `audioplayer` object.

`player = play(recObj, start)` plays audio from the sample indicated by `start` to the end.

`player = play(recObj, [start stop])` plays audio from the sample indicated by `start` to the sample indicated by `stop`.

Examples

Record 5 seconds of your speech with a microphone, and play it back. Display the properties of the `audioplayer` object.

```
myVoice = audiorecorder;

disp('Start speaking.');
recordblocking(myVoice, 5);
disp('End of recording. Playing back ...');

playerObj = play(myVoice);

disp('Properties of playerObj:');
get(playerObj)
```

Play back only the first 3 seconds of the speech recorded in the previous example:

```
play(myVoice, [1 myVoice.SampleRate*3]);
```

See Also

audioplayer | audiorecorder

playblocking

Play audio from `audioplayer` object, holding control until playback completes

Syntax

```
playblocking(playerObj)
playblocking(playerObj,start)
playblocking(playerObj,[start,stop])
```

Description

`playblocking(playerObj)` plays the audio associated with `audioplayer` object `playerObj` from beginning to end. `playblocking` does not return control until playback completes.

`playblocking(playerObj,start)` plays audio from the sample indicated by `start` to the end.

`playblocking(playerObj,[start,stop])` plays audio from the sample indicated by `start` to the sample indicated by `stop`.

Examples

Play with and without Blocking

Play two audio samples with and without blocking using the `play` and `playblocking` methods.

Load data from example files `chirp.mat` and `gong.mat`.

```
chirpData = load('chirp.mat');
chirpObj = audioplayer(chirpData.y,chirpData.Fs);

gongData = load('gong.mat');
gongObj = audioplayer(gongData.y,gongData.Fs);
```

Play the samples with blocking, one after the other.

```
playblocking(chirpObj);
playblocking(gongObj);
```

Play without blocking. The audio can overlap.

```
play(chirpObj);
play(gongObj);
```

Starting Sample

Play audio from the example file `handel.mat` starting 4 seconds from the beginning.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = playerObj.SampleRate * 4;

playblocking(playerObj,start);
beep;
```

Sample Range

Play the first 3 seconds of audio from the example file `handel.mat`.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = 1;
stop = playerObj.SampleRate * 3;

playblocking(playerObj,[start,stop]);
beep;
```

See Also

`audioplayer` | `play`

How To

- "Play Audio"

plot

2-D line plot

Syntax

```
plot(X,Y)
plot(X,Y,LineSpec)
plot(X1,Y1,...,Xn,Yn)
plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)

plot(Y)
plot(Y,LineSpec)

plot( ___ ,Name,Value)
plot(ax, ___ )

h = plot( ___ )
```

Description

`plot(X,Y)` creates a 2-D line plot of the data in `Y` versus the corresponding values in `X`.

- If `X` and `Y` are both vectors, then they must have equal length. The `plot` function plots `Y` versus `X`.
- If `X` and `Y` are both matrices, then they must have equal size. The `plot` function plots columns of `Y` versus columns of `X`.
- If one of `X` or `Y` is a vector and the other is a matrix, then the matrix must have dimensions such that one of its dimensions equals the vector length. If the number of matrix rows equals the vector length, then the `plot` function plots each matrix column versus the vector. If the number of matrix columns equals the vector length, then the function plots each matrix row versus the vector. If the matrix is square, then the function plots each column versus the vector.
- If one of `X` or `Y` is a scalar and the other is either a scalar or a vector, then the `plot` function plots discrete points. However, to see the points you must specify a marker symbol, for example, `plot(X,Y,'o')`.

`plot(X,Y,LineSpec)` sets the line style, marker symbol, and color.

`plot(X1,Y1,...,Xn,Yn)` plots multiple `X`, `Y` pairs using the same axes for all lines.

`plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)` sets the line style, marker type, and color for each line. You can mix `X`, `Y`, `LineSpec` triplets with `X`, `Y` pairs. For example, `plot(X1,Y1,X2,Y2,LineSpec2,X3,Y3)`.

`plot(Y)` creates a 2-D line plot of the data in `Y` versus the index of each value.

- If `Y` is a vector, then the *x*-axis scale ranges from 1 to `length(Y)`.
- If `Y` is a matrix, then the `plot` function plots the columns of `Y` versus their row number. The *x*-axis scale ranges from 1 to the number of rows in `Y`.
- If `Y` is complex, then the `plot` function plots the imaginary part of `Y` versus the real part of `Y`, such that `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`.

`plot(Y,LineSpec)` sets the line style, marker symbol, and color.

`plot( ___ ,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes. `Name,Value` pair settings apply to all the lines plotted. You cannot specify different `Name,Value` pairs for each line using this syntax.

`plot(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax` can precede any of the input combinations in the previous syntaxes.

`h = plot( ___ )` returns a column vector of chart line objects. Use `h` to modify a specific chart lines after it is created.

Examples

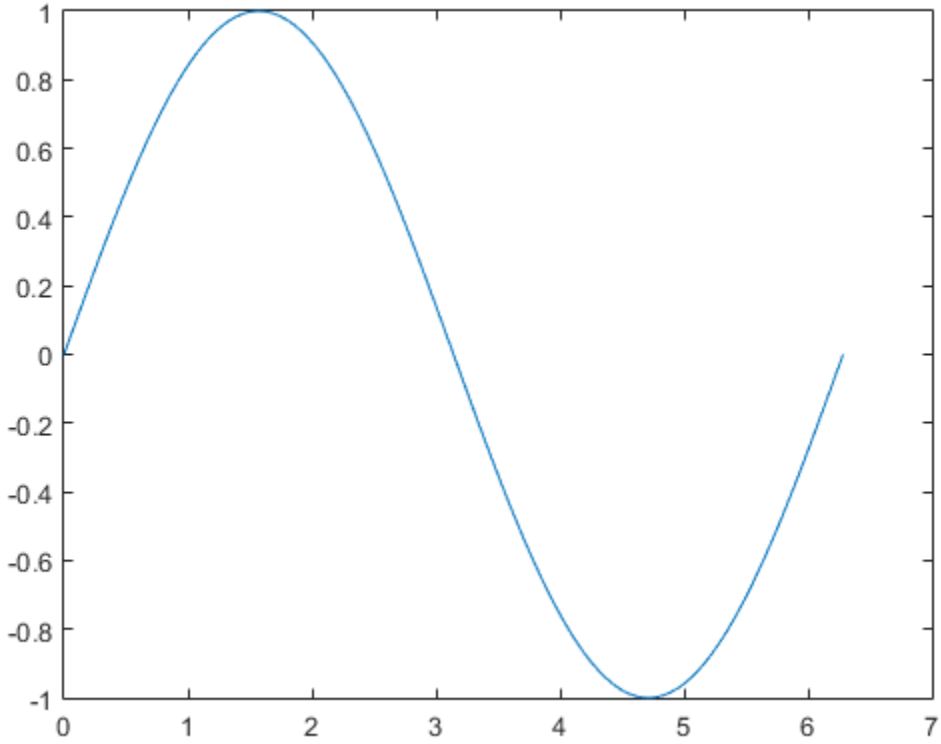
Create Line Plot

Define `x` as a vector of linearly spaced values between 0 and 2π. Use an increment of $\pi/100$ between the values. Define `y` as sine values of `x`.

```
x = 0:pi/100:2*pi;
y = sin(x);
```

Create a line plot of the data.

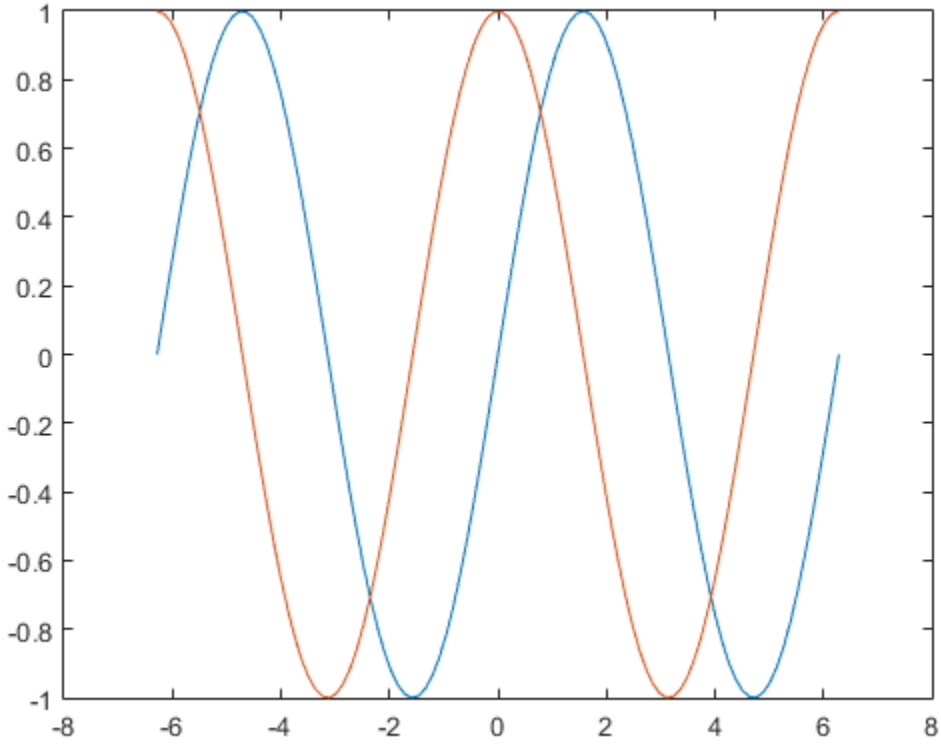
```
figure % opens new figure window
plot(x,y)
```



Plot Multiple Lines

Define `x` as 100 linearly spaced values between -2π and 2π. Define `y1` and `y2` as sine and cosine values of `x`. Create a line plot of both sets of data.

```
x = linspace(-2*pi,2*pi);
y1 = sin(x);
y2 = cos(x);

figure
plot(x,y1,x,y2)
```



Create Line Plot From Matrix

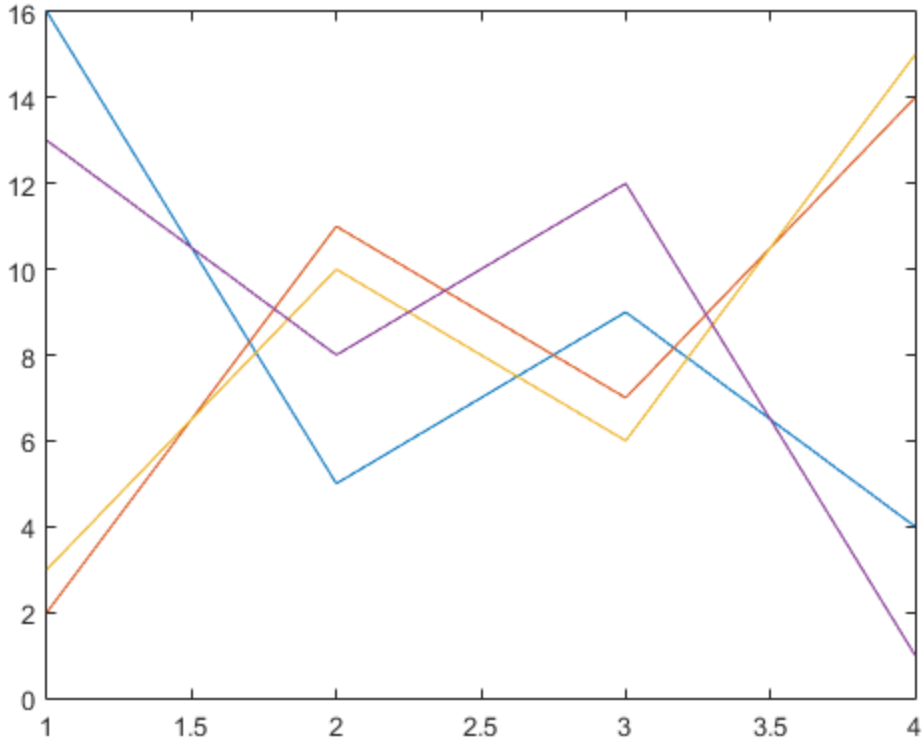
Define `Y` as the 4-by-4 matrix returned by the `magic` function.

```
Y = magic(4)
```

```
Y =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Create a 2-D line plot of `Y`. MATLAB® plots each matrix column as a separate line.

```
figure
plot(Y)
```

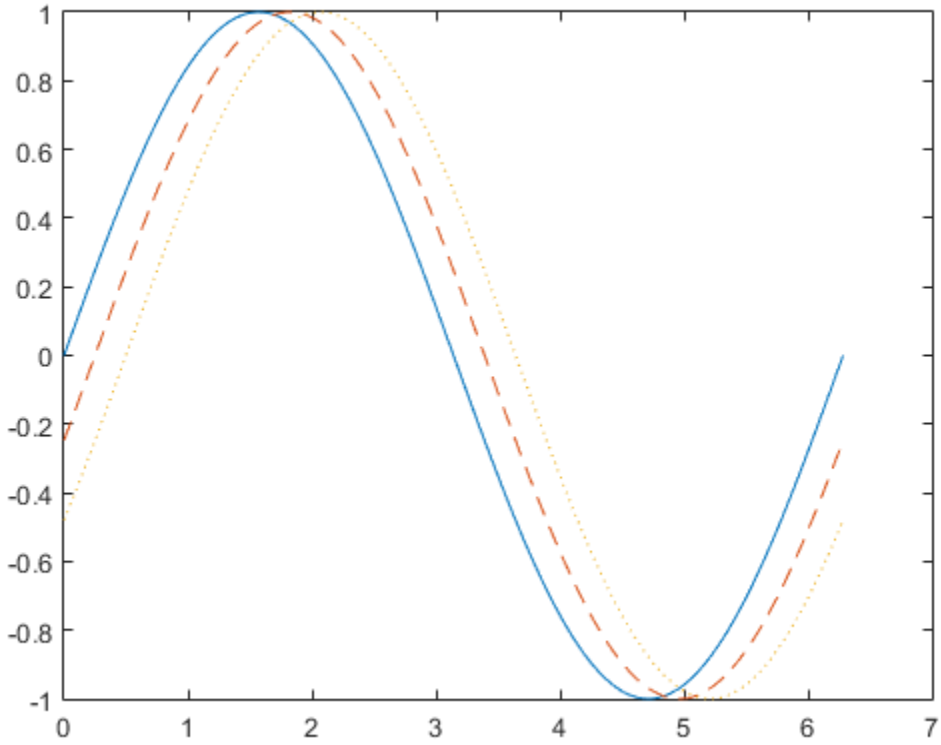


Specify Line Style

Plot three sine curves with a small phase shift between each line. Use the default line style for the first line. Specify a dashed line style for the second line and a dotted line style for the third line.

```
x = 0:pi/100:2*pi;
y1 = sin(x);
y2 = sin(x-0.25);
```

```
y3 = sin(x-0.5);

figure
plot(x,y1,x,y2,'--',x,y3,':')
```



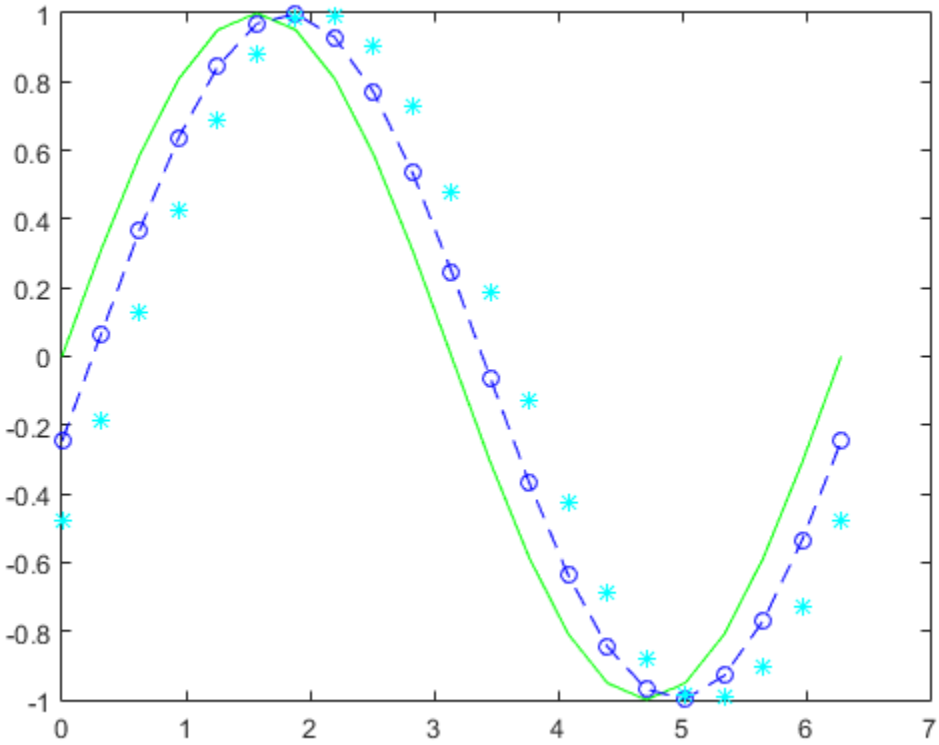
MATLAB® cycles the line color through the default color order.

Specify Line Style, Color, and Marker

Plot three sine curves with a small phase shift between each line. Use a green line with no markers for the first sine curve. Use a blue dashed line with circle markers for the second sine curve. Use only cyan star markers for the third sine curve.

```
x = 0:pi/10:2*pi;
```

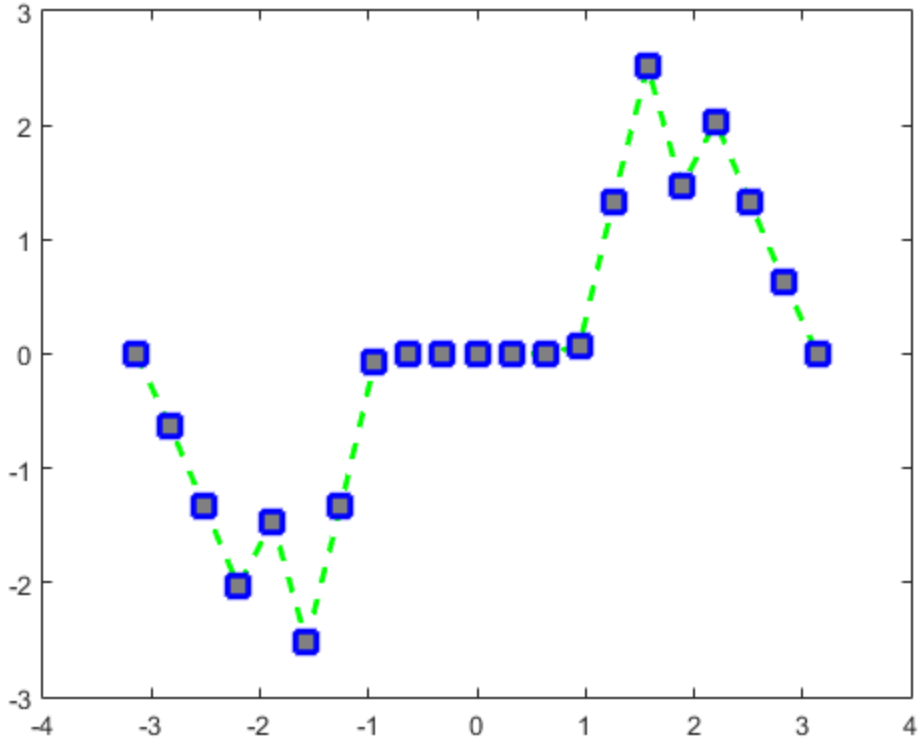
```
y1 = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.5);

figure
plot(x,y1,'g',x,y2,'b--o',x,y3,'c*')
```



Specify Line Width, Marker Size, and Marker Color

Create a line plot and use the `LineSpec` option to specify a dashed green line with square markers. Use `Name,Value` pairs to specify the line width, marker size, and marker colors. Set the marker edge color to blue and set the marker face color using an RGB color value.

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));

figure
plot(x,y,'--gs',...
    'LineWidth',2,...
    'MarkerSize',10,...
    'MarkerEdgeColor','b',...
    'MarkerFaceColor',[0.5,0.5,0.5])
```



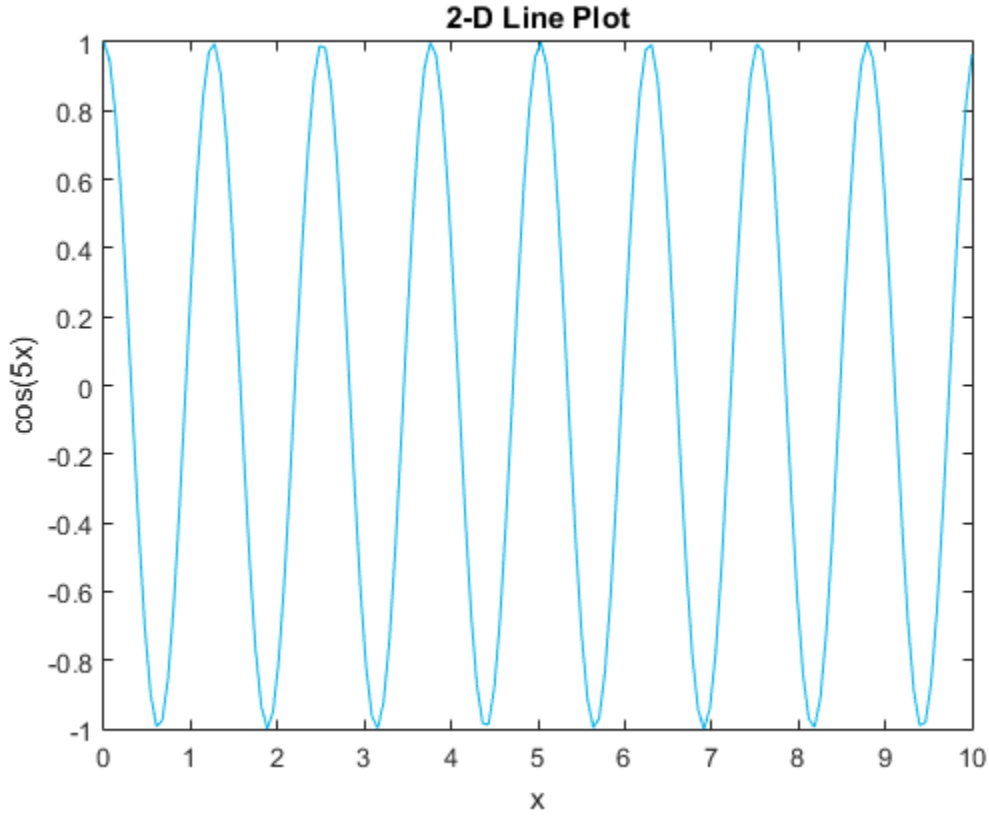
Add Title and Axis Labels

Use the `linspace` function to define `x` as a vector of 150 values between 0 and 10. Define `y` as cosine values of `x`.

```
x = linspace(0,10,150);
y = cos(5*x);
```

Create a 2-D line plot of the cosine curve. Change the line color to a shade of blue-green using an RGB color value. Add a title and axis labels to the graph using the `title`, `xlabel`, and `ylabel` functions.

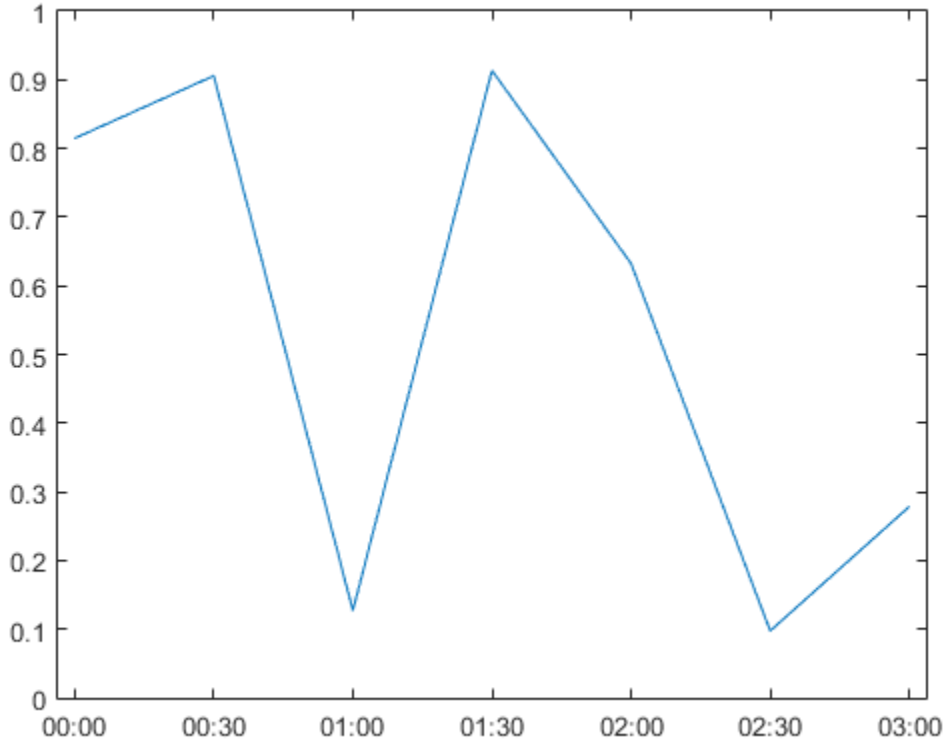
```
figure
plot(x,y,'Color',[0,0.7,0.9])

title('2-D Line Plot')
xlabel('x')
ylabel('cos(5x)')
```



Plot Durations and Specify Tick Format

Define `t` as seven linearly spaced `duration` values between 0 and 3 minutes. Plot random data and specify the format of the `duration` tick marks using the `'DurationTickFormat'` name-value pair argument.

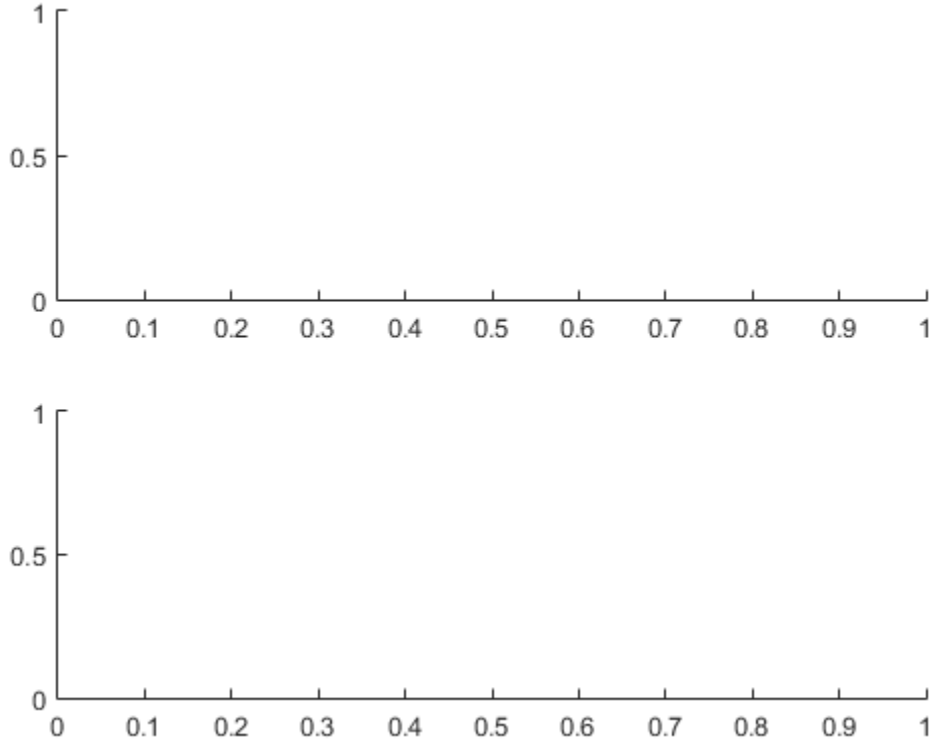
```
t = 0:seconds(30):minutes(3);
y = rand(1,7);

plot(t,y,'DurationTickFormat','mm:ss')
```



Specify Axes for Line Plot

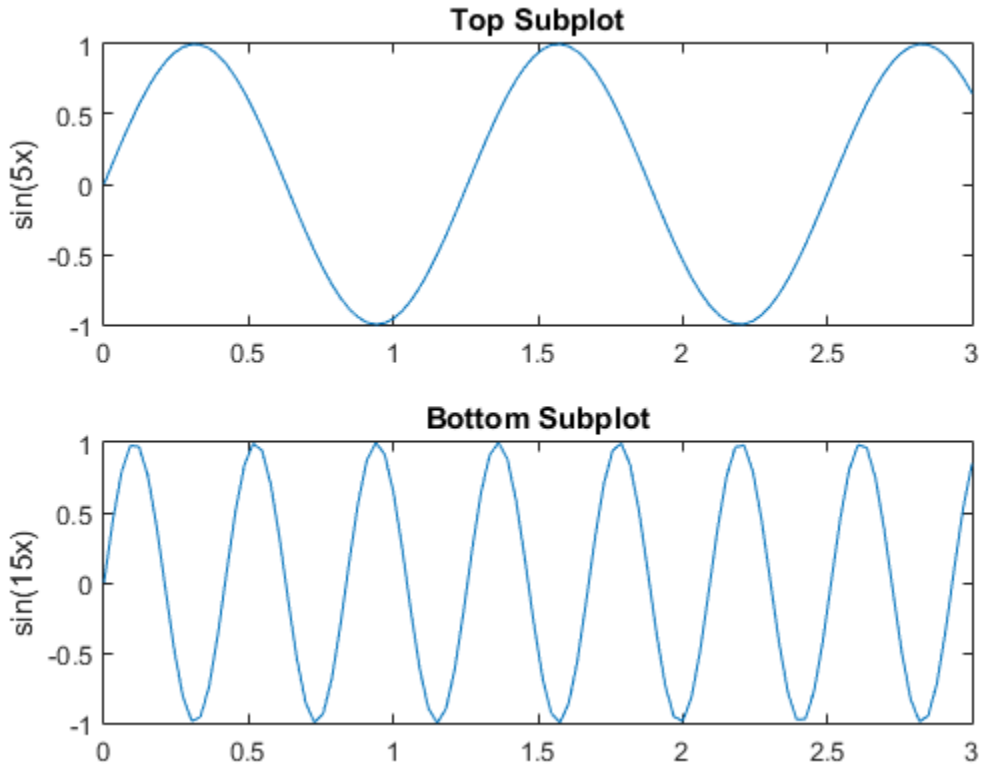
Create a figure with two subplots and return the handle to each subplot axes, `ax1` and `ax1`.

```
figure % new figure
ax1 = subplot(2,1,1); % top subplot
ax2 = subplot(2,1,2); % bottom subplot
```



Create a 2-D line plot in each axes by referring to the axes handles. Add a title and *y*-axis label to each axes by passing the axes handles to the `title` and `ylabel` functions.

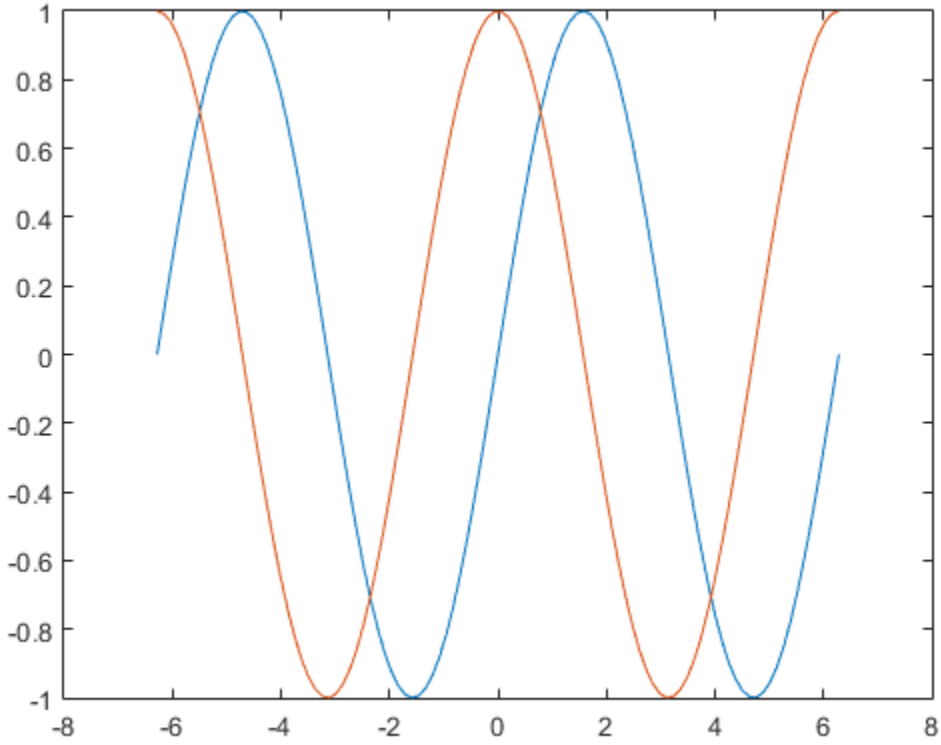
```
x = linspace(0,3);
y1 = sin(5*x);
y2 = sin(15*x);

plot(ax1,x,y1)
title(ax1,'Top Subplot')
ylabel(ax1,'sin(5x)')

plot(ax2,x,y2)
title(ax2,'Bottom Subplot')
ylabel(ax2,'sin(15x)')
```



Modify Lines After Creation

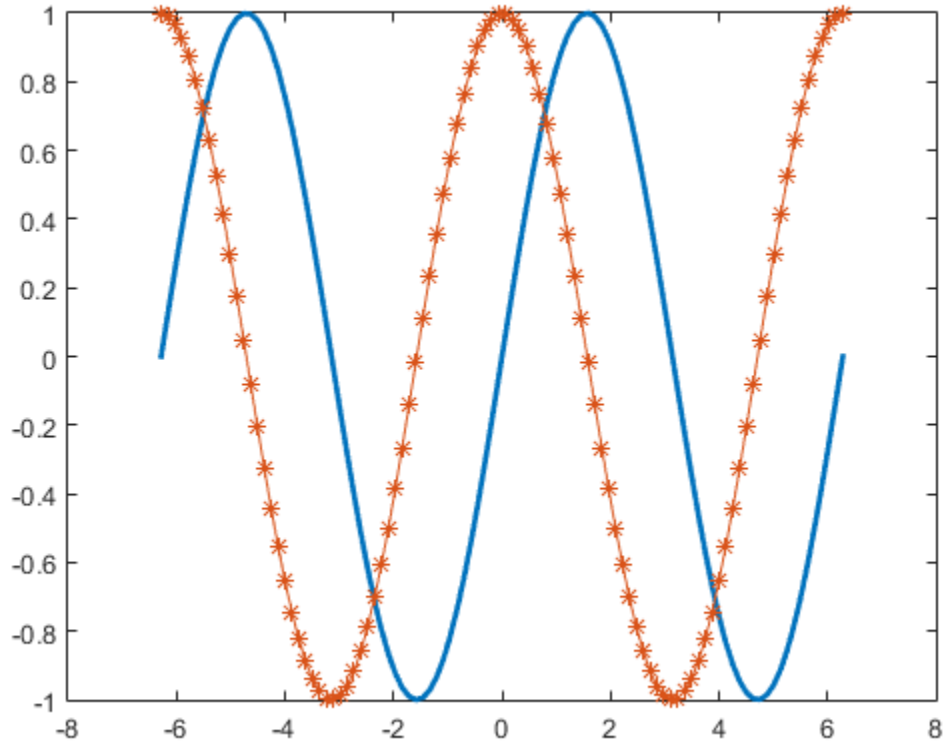
Define `x` as 100 linearly spaced values between -2π and 2π. Define `y1` and `y2` as sine and cosine values of `x`. Create a line plot of both sets of data and return the two chart lines in `p`.

```
x = linspace(-2*pi,2*pi);
y1 = sin(x);
y2 = cos(x);
p = plot(x,y1,x,y2);
```



Change the line width of the first line to 2. Add star markers to the second line. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

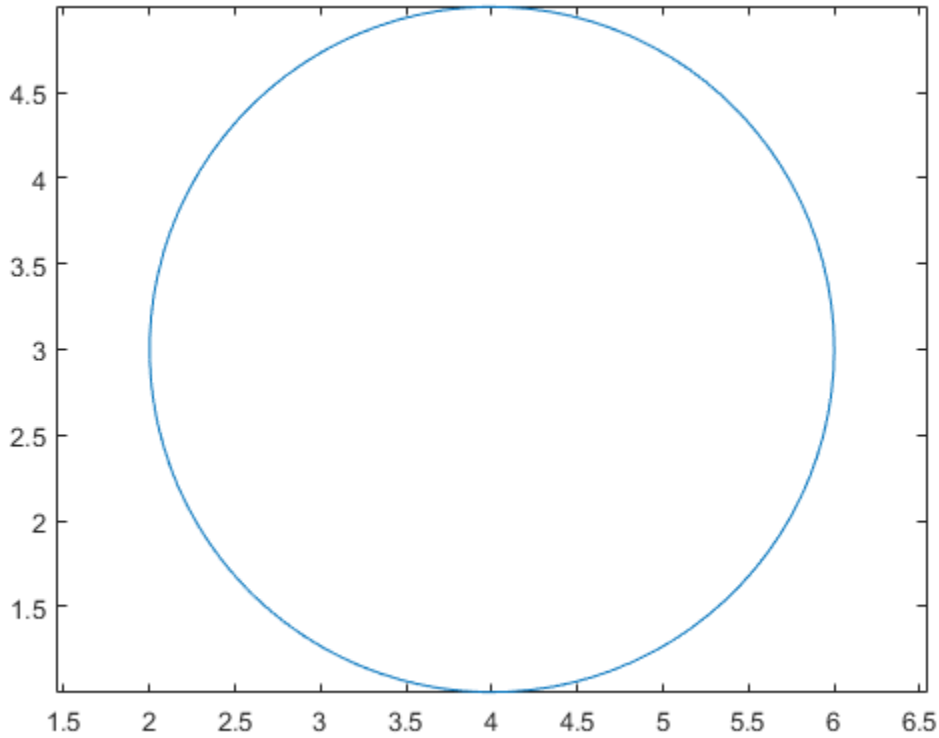
```
p(1).LineWidth = 2;
p(2).Marker = '*';
```



Plot Circle

Plot a circle centered at the point (4,3) with a radius equal to 2. Use `axis equal` to use equal data units along each coordinate direction.

```
r = 2;
xc = 4;
yc = 3;

theta = linspace(0,2*pi);
x = r*cos(theta) + xc;
y = r*sin(theta) + yc;
plot(x,y)
axis equal
```



Input Arguments

Y — y values

scalar | vector | matrix

y values, specified as a scalar, a vector, or a matrix. Y can be a numeric array, logical array, datetime array, or duration array. To plot against specific *x* values you must also specify X.

X — x values

scalar | vector | matrix

x values, specified as a scalar, a vector, or a matrix. `X` can be a numeric array, logical array, datetime array, or duration array.

`LineSpec` — Line style, marker symbol, and color

string

Line style, marker symbol, and color, specified as a string. The elements of the string can appear in any order, and you can omit one or more options from the string specifier. If you omit the line style and specify the marker character, then the plot shows only the marker and no line.

If `Y` is a matrix and you specify a color with `LineSpec`, then all the lines use the specified color. If you specify a marker type or line style and do not specify a color, then the lines cycle through the color order.

Example: `'--or'` is a red dashed line with circle markers

Specifier	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Specifier	Marker
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle

Specifier	Marker
p	Pentagram
h	Hexagram

Specifier	Color
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

`ax` — Axes object

axes object

Axes object. If you do not specify an axes, then the `plot` function uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The chart line properties listed here are only a subset. For a complete list, see Chart Line Properties.

Example: `'Marker','o','MarkerFaceColor','red'`

`'Color'` — Line color

`[0 0.4470 0.7410]` (default) | RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string, or `'none'`. If you specify the `Color` as `'none'`, then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`'DatetimeTickFormat'` — Format for `datetime` tick labels

string

Format for `datetime` tick labels, specified as the comma-separated pair consisting of `'DatetimeTickFormat'` and a string. Use the letters `A-Z` and `a-z` to construct a custom string value. These letters correspond to the Unicode Locale Data Markup Language (LDML) standard for dates. You can include non-ASCII letter characters such as a hyphen, space, or colon to separate the fields.

If you do not specify a value for `'DatetimeTickFormat'`, then `plot` automatically optimizes and updates the tick labels based on the axis limits.

Example: `'DatetimeTickFormat','eeee, MMMM d, yyyy HH:mm:ss'` displays a date and time such as `Saturday, April 19, 2014 21:41:06`.

The following table shows several common display formats and examples of the formatted output for the date, Saturday, April 19, 2014 at 9:41:06 PM in New York City.

Value of `DatetimeTickFormat`	Example
`'yyyy-MM-dd'`	2014-04-19

Value of `DatetimeTickFormat`	Example
`'dd/MM/yyyy'`	`19/04/2014`
`'dd.MM.yyyy'`	`19.04.2014`
`'yyyy# MM# dd#'`	`2014# 04# 19#`
`'MMMM d, yyyy'`	`April 19, 2014`
`'eeee, MMMM d, yyyy HH:mm:ss'`	`Saturday, April 19, 2014 21:41:06`
`'MMMM d, yyyy HH:mm:ss Z'`	`April 19, 2014 21:41:06 -0400`

For a complete list of valid letter identifiers, see the `Format` property for datetime arrays.

`DatetimeTickFormat` is not a chart line property. You must set the tick format using the name-value pair argument when creating a plot.

`'DurationTickFormat'` — Format for `duration` tick labels

string

Format for `duration` tick labels, specified as the comma-separated pair consisting of `'DurationTickFormat'` and a string.

If you do not specify a value for `'DurationTickFormat'`, then `plot` automatically optimizes and updates the tick labels based on the axis limits.

To display a duration as a single number that includes a fractional part, for example, 1.234 hours, specify one of the following strings.

Value of `DurationTickFormat`	Description
`'y'`	Number of exact fixed-length years. A fixed-length year is equal to 365.2425 days.
`'d'`	Number of exact fixed-length days. A fixed-length day is equal to 24 hours.
`'h'`	Number of hours
`'m'`	Number of minutes
`'s'`	Number of seconds

Example: `'DurationTickFormat','h'` displays duration values in terms of fixed-length days.

To display a duration in the form of a digital timer, specify one of the following strings.

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

In addition, you can display up to nine fractional second digits by appending up to nine `S` characters.

Example: `'DurationTickFormat','hh:mm:ss.SSS'` displays the milliseconds of a duration value to three digits.

`DurationTickFormat` is not a chart line property. You must set the tick format using the name-value pair argument when creating a plot.

`'LineStyle'` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`'LineWidth'` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`'Marker'` — Marker symbol

`'none'` (default) | string

Marker symbol, specified as one of the marker strings in this table. By default, a chart line does not have markers. Add markers at each data point along the line by specifying a marker symbol.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'Marker','+'`

Example: `'Marker','diamond'`

`'MarkerEdgeColor'` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

[0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.5 0.5 0.5]

Example: 'blue'

'MarkerFaceColor' — Marker fill color

'none' (default) | 'auto' | RGB triplet | color string

Marker fill color, specified as one of these values:

- 'none' — Use no color, which makes the interior invisible.
- 'auto' — Use the same color as the Color property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]

Long Name	Short Name	RGB Triplet
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.3 0.2 0.1]

Example: 'green'

'MarkerSize' — Marker size
6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: 10

Output Arguments

h — One or more chart line objects
scalar | vector

One or more chart line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line.

More About

Tips

- Use NaN and Inf values to create breaks in the lines. For example, this code plots the first two elements, skips the third element, and draws another line using the last two elements:

  ```
  plot([1,2,NaN,4,5])
  ```

- plot uses colors and line styles based on the ColorOrder and LineStyleOrder properties of the axes. plot cycles through the colors with the first line style. Then, it cycles through the colors again with each additional line style.

You can change the default colors and line styles by setting default values for the `ColorOrder` and `LineStyleOrder` properties. For example, to set the default line styles to a solid line with asterisk markers, a dotted line, and circle markers with no line, use this command:

```
set(groot,'defaultAxesLineStyleOrder',{'-*',':','o'})
```

For more information about setting defaults, see “Default Property Values”.

See Also

Functions

`gca` | `hold` | `legend` | `loglog` | `plot3` | `plotyy` | `title` | `xlabel` | `xlim` | `ylabel` | `ylim`

Properties

Chart Line Properties

Introduced before R2006a

plot3

3-D line plot

Syntax

```
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
plot3(axes_handle,...)
h = plot3(...)
```

Description

The `plot3` function displays a three-dimensional plot of a set of data points.

`plot3(X1,Y1,Z1,...)`, where `X1`, `Y1`, `Z1` are vectors or matrices, plots one or more lines in three-dimensional space through the points whose coordinates are the elements of `X1`, `Y1`, and `Z1`.

`plot3(X1,Y1,Z1,LineSpec,...)` creates and displays all lines defined by the `Xn,Yn,Zn,LineSpec` quads, where `LineSpec` is a line specification that determines line style, marker symbol, and color of the plotted lines.

`plot3(...,'PropertyName',PropertyValue,...)` sets line properties to the specified property values for all the charting lines created by `plot3`. See Chart Line Properties for a description of the properties you can set.

`plot3(axes_handle,...)` plots into the axes specified by `axes_handle` instead of into the current axes (`gca`). The option, `axes_handle` can precede any of the input combinations in the previous syntaxes.

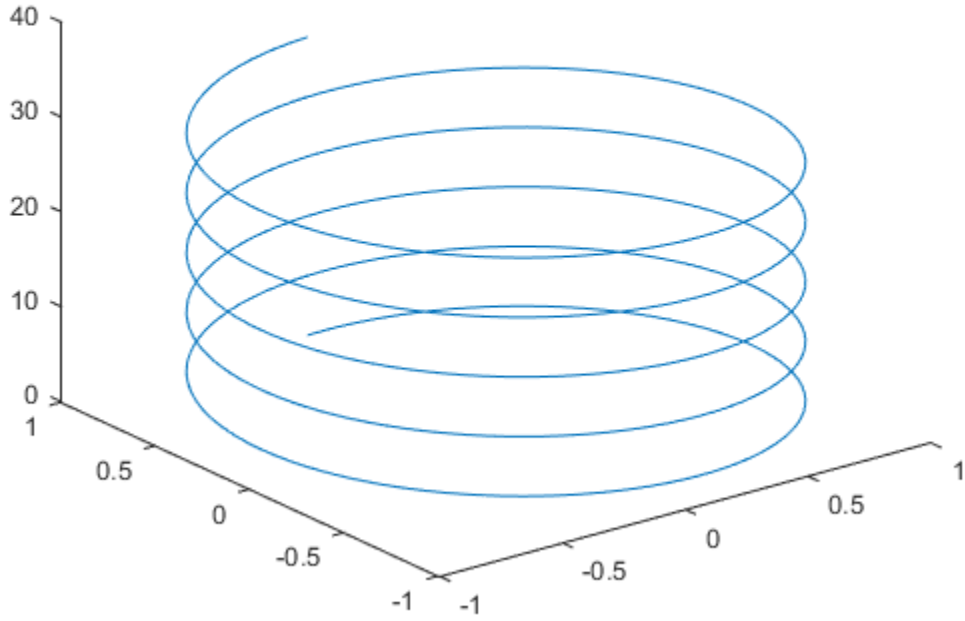
`h = plot3(...)` returns a column vector of charting line handles, with one handle per object.

Examples

Plot 3-D Helix

Define `t` as values between 0 and 10π. Define `st` and `ct` as vectors of sine and cosine values. Plot a 3-D helix.

```
t = 0:pi/50:10*pi;
st = sin(t);
ct = cos(t);

figure
plot3(st,ct,t)
```



More About

Tips

If one or more of `X1`, `Y1`, `Z1` is a vector, the vectors are plotted versus the rows or columns of the matrix, depending whether the vectors' lengths equal the number of rows or the number of columns.

You can mix `Xn,Yn,Zn` triples with `Xn,Yn,Zn,`*`LineSpec`* quads, for example,

```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

See `LineSpec` and `plot` for information on line types and markers.

See Also

| | | | | | | | | semilogx | semilogy

See Also

Functions

axis | bar3 | LineSpec | loglog | plot | scatter3 | subplot

Properties

Chart Line Properties

Introduced before R2006a

plotbrowser

Show or hide figure **Plot Browser**

Syntax

```
plotbrowser('on')
plotbrowser('off')
plotbrowser
plotbrowser(figure_handle,...)
```

Description

`plotbrowser('on')` displays the Plot Browser on the current figure.

`plotbrowser('off')` hides the Plot Browser on the current figure.

`plotbrowser` toggles the visibility of the Plot Browser on the current figure. You can use `plotbrowser('toggle')` instead for the same functionality.

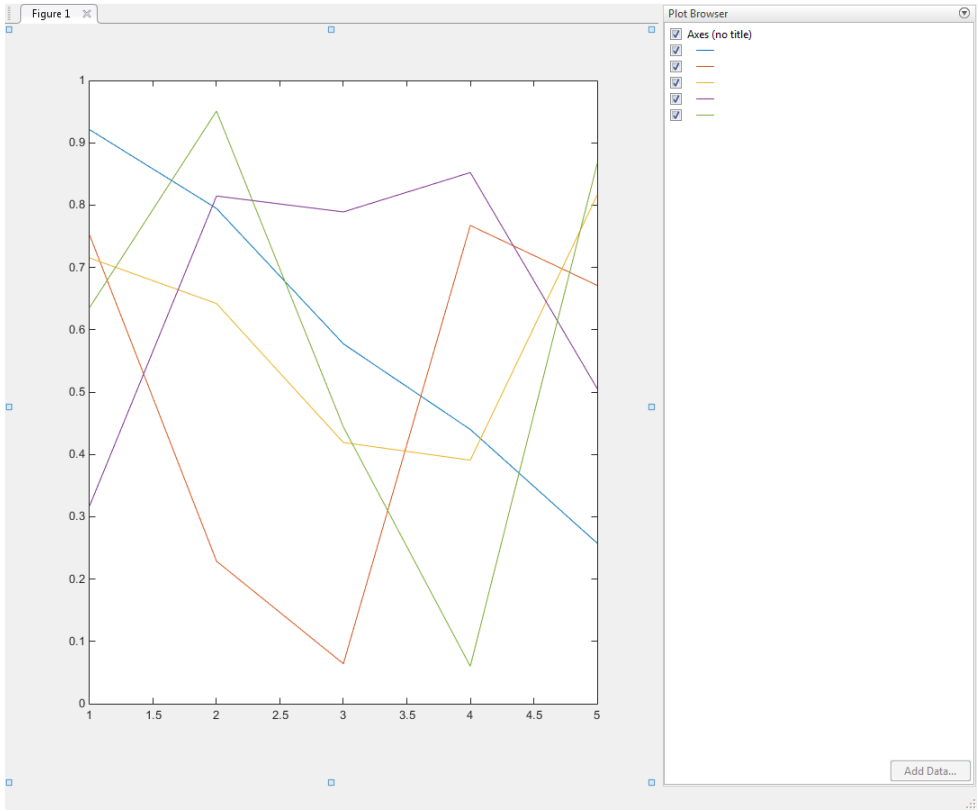
`plotbrowser(figure_handle,...)` shows or hides the Plot Browser on the figure specified by `figure_handle`.

Examples

Open Plot Browser

Plot a 5-by-5 matrix of random numbers. Then, open the plot browser.

```
plot(rand(5))
plotbrowser('on')
```



Alternatives

To collectively enable Plotting Tools, use the large Plotting Tool icon on the figure toolbar. To collectively disable the Plotting Tools, use the smaller icon. Open or close the **Plot Browser** tool from the figure's **View** menu.

More About

Tips

If you call `plotbrowser` in a MATLAB program and subsequent lines depend on the Plot Browser being fully initialized, follow it by `drawnow` to ensure complete initialization.

See Also

`plottools` | `figurepalette` | `propertyeditor`

Introduced before R2006a

plotedit

Interactively edit and annotate plots

Syntax

```
plotedit on
plotedit off
plotedit
plotedit(h)
plotedit(state)
plotedit(h,state)
```

Description

`plotedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, change line styles, and add text, line, and arrow annotations.

`plotedit off` ends plot mode for the current figure.

`plotedit` toggles the plot edit mode for the current figure.

`plotedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plotedit(state)` specifies the `plotedit` state for the current figure. Values for `state` can be as shown.

Value for state	Description
`'on'`	Starts plot edit mode
`'off'`	Ends plot edit mode
`'showtoolsmenu'`	Displays the **Tools** menu in the menu bar
`'hidetoolsmenu'`	Removes the **Tools** menu from the menu bar

Note `'hidetoolsmenu'` is intended for UI developers who do not want the **Tools** menu to appear in applications that use the figure window.

`plotedit(h,state)` specifies the `plotedit` `state` for figure handle `h`.

Examples

Start plot edit mode for figure 2.

```
plotedit(2)
```

End plot edit mode for figure 2.

```
plotedit(2, 'off')
```

Hide the **Tools** menu for the current figure:

```
plotedit('hidetoolsmenu')
```

See Also

`axes` | `line` | `open` | `plot` | `print` | `saveas` | `text` | `propedit`

Introduced before R2006a

plotmatrix

Scatter plot matrix

Syntax

```
plotmatrix(X,Y)
plotmatrix(X)
plotmatrix(___,LineSpec)

[H,AX,BigAx,P,PAx] = plotmatrix(___)
```

Description

`plotmatrix(X,Y)` creates a matrix of subaxes containing scatter plots of the columns of `X` against the columns of `Y`. If `X` is *p*-by-*n* and `Y` is *p*-by-*m*, then `plotmatrix` produces an *n*-by-*m* matrix of subaxes.

`plotmatrix(X)` is the same as `plotmatrix(X,X)` except that the subaxes along the diagonal are replaced with histogram plots of the data in the corresponding column of `X`. For example, the subaxes along the diagonal in the *i*th column is replaced by `hist(X(:,i))`.

`plotmatrix(___,LineSpec)` specifies the line style, marker symbol, and color for the scatter plots. The option `LineSpec` can be preceded by any of the input argument combinations in the previous syntaxes.

`[H,AX,BigAx,P,PAx] = plotmatrix(___)` returns handles to the graphic objects created as follows:

- `H` – Matrix of handles to the line objects used to create the scatter plots
- `AX` – Matrix of handles to the individual subaxes
- `BigAx` – Handle to the big axes that frames the subaxes
- `P` – Vector of handles for the patch objects that create the histogram plots
- `PAx` – Vector of handles to the invisible histogram axes

`BigAx` is left as the current axes (`gca`) so that a subsequent `title`, `xlabel`, or `ylabel` command will center text with respect to the big axes.

Examples

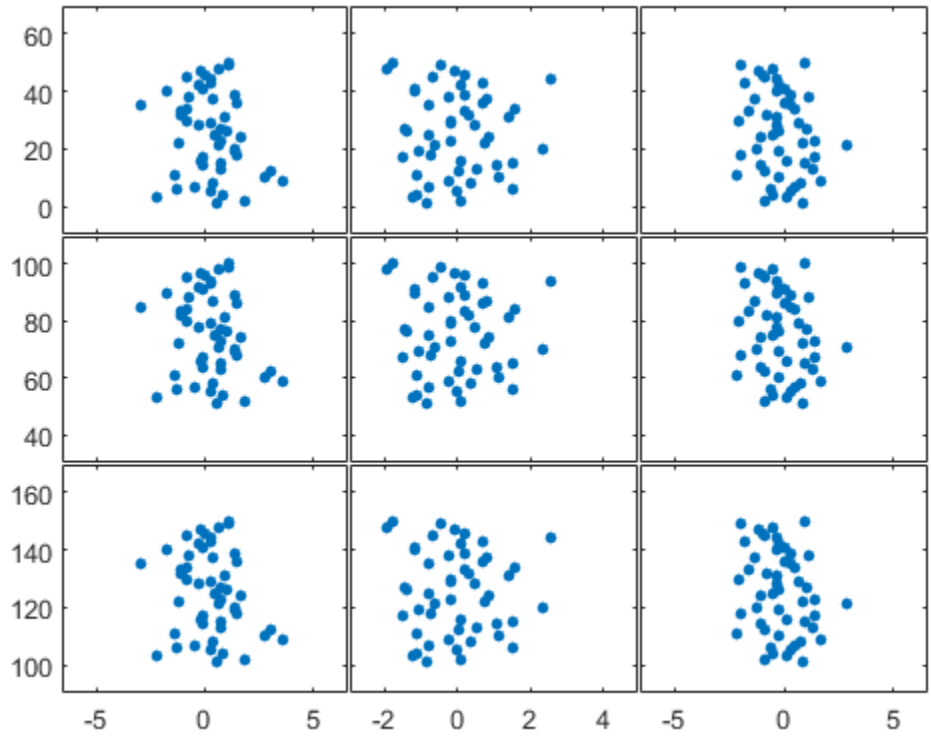
Create Scatter Plot Matrix with Two Matrix Inputs

Initialize the random-number generator to make the output of `randn` repeatable. Define `X` as a matrix of normally distributed pseudorandom data and `Y` as a matrix of integer values.

```
rng default
X = randn(50,3);
Y = reshape(1:150,50,3);
```

Create a scatter plot matrix of the columns of `X` against the columns of `Y`.

```
figure
plotmatrix(X,Y)
```



The subplot in the ith row, jth column of the figure is a scatter plot of the ith column of `Y` against the jth column of `X`.

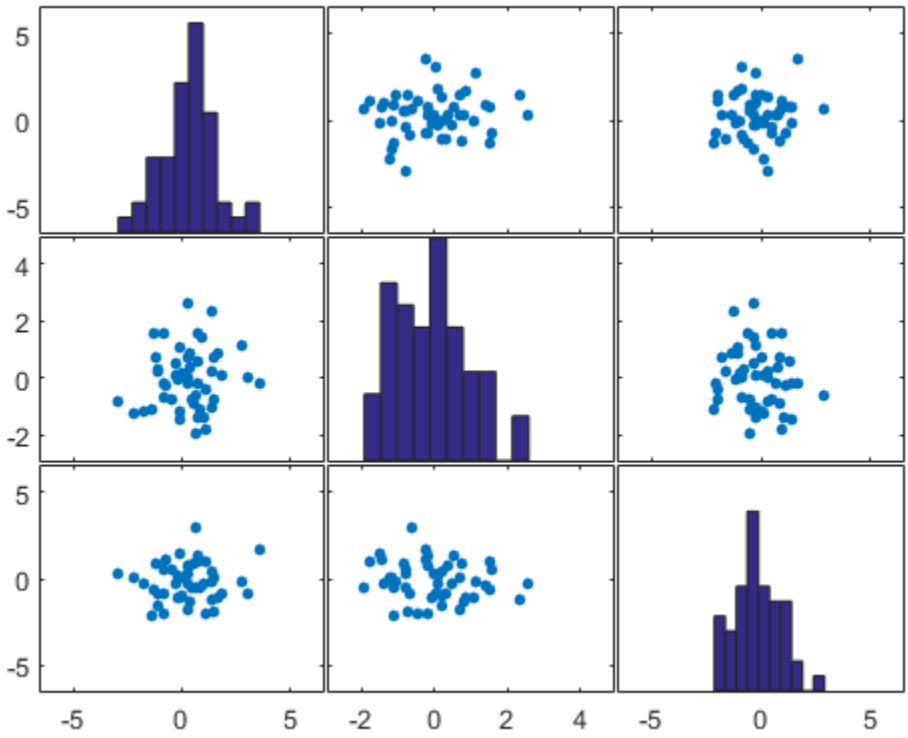
Create Scatter Plot Matrix with One Matrix Input

Initialize the random-number generator to make the output of `randn` repeatable and generate a matrix of normally distributed pseudorandom data

```
rng default
X = randn(50,3);
```

Create a scatter plot matrix.

```
figure
plotmatrix(X)
```



The subplot in the ith row, jth column of the matrix is a scatter plot of the ith column of `X` against the jth column of `X`. Along the diagonal, `plotmatrix` creates a histogram plot of each column of `X`.

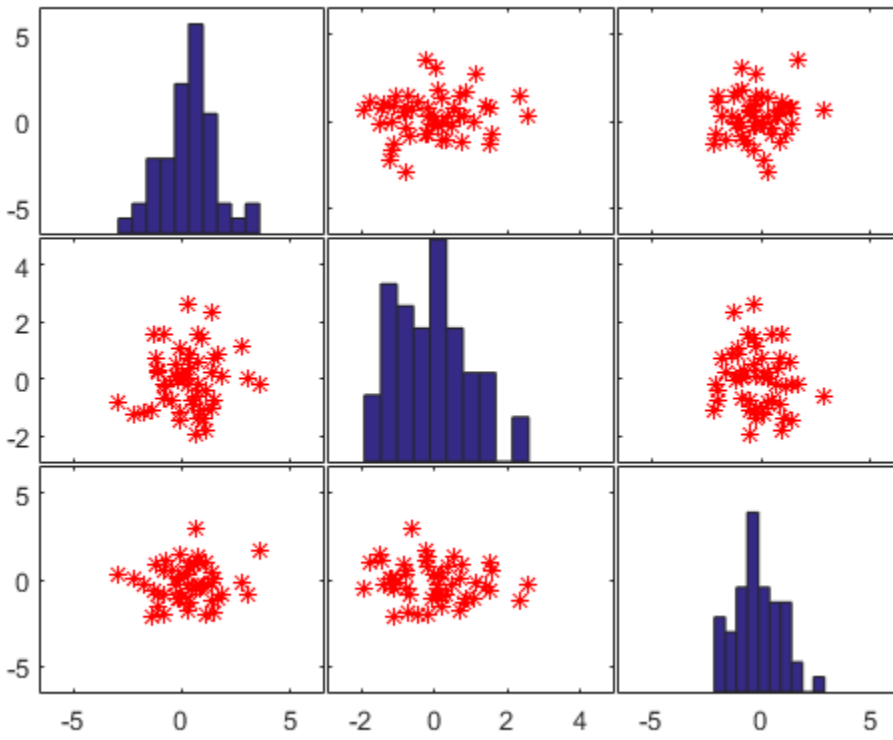
Specify Marker Type and Color

Initialize the random-number generator to make the output of `randn` repeatable. Generate a matrix of normally distributed pseudorandom data.

```
rng(0,'twister');
X = randn(50,3);
```

Create a scatter plot matrix and specify the marker type and the color for the scatter plots.

```
figure
plotmatrix(X,'*r')
```

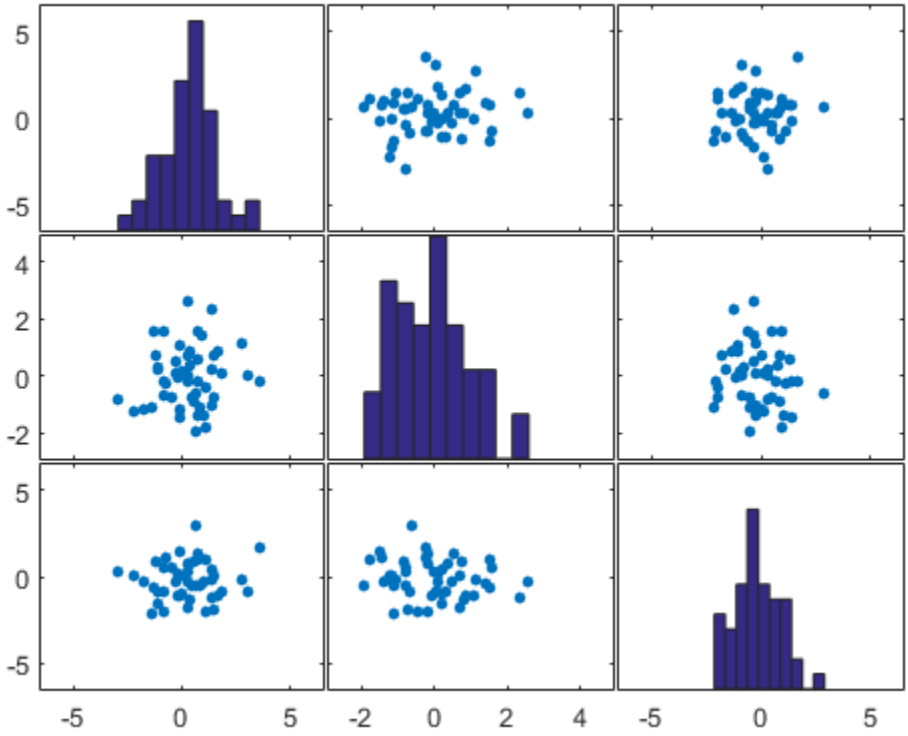


The `LineSpec` option sets properties for the scatter plots. To set properties for the histogram plots, use the patch object handles.

Set Plotmatrix Properties after Creation

Create a scatter plot matrix of random data.

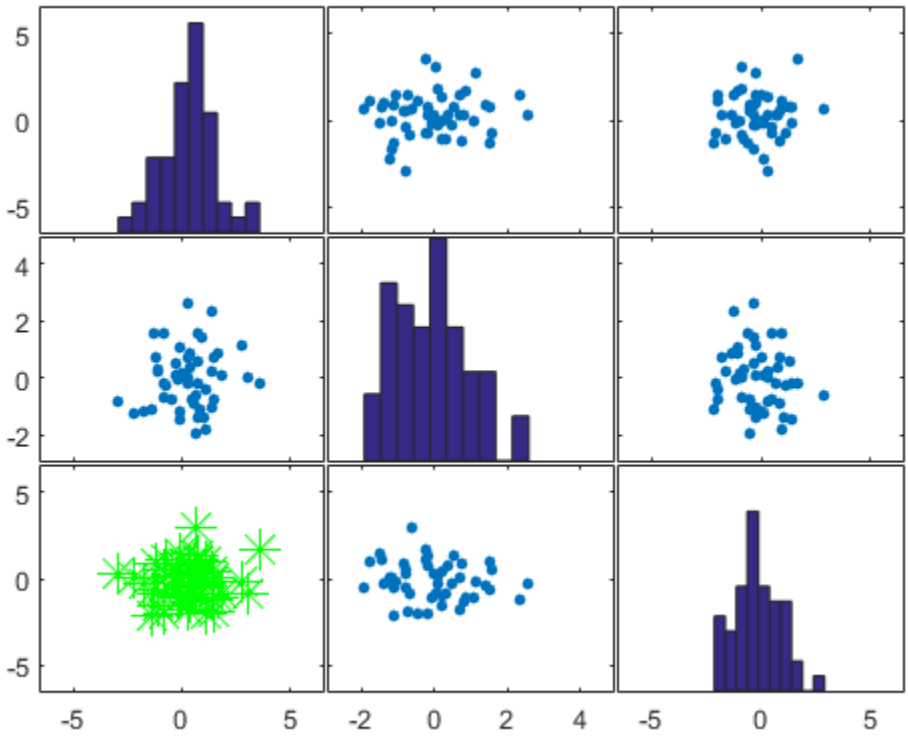
```
rng default
X = randn(50,3);
[H,AX,BigAx,P,PAx] = plotmatrix(X);
```



To set properties for the scatter plots, use `H`. To set properties for the histograms, use `P`. To set axes properties, use `AX`, `BigAx`, and `PAx`. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

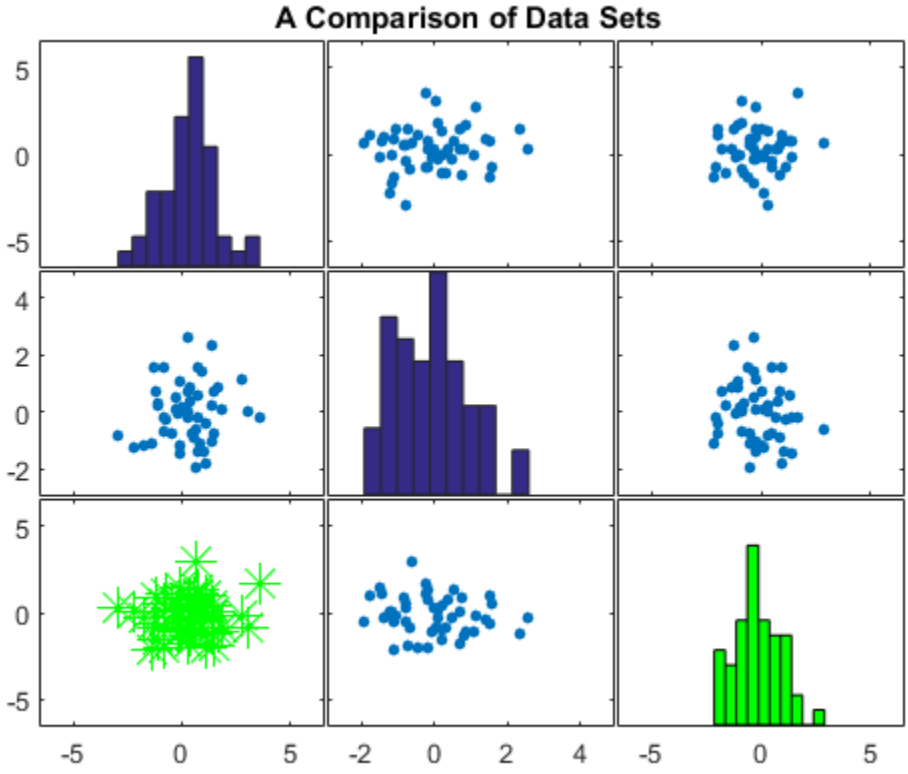
Set the color and marker type for the scatter plot in the lower left corner.

```
H(3).Color = 'g';
H(3).Marker = '*';
```



Set the color for the histogram plot in the lower right corner. Use the `title` command to title the figure.

```
P(3).EdgeColor = 'k';
P(3).FaceColor = 'g';
title(BigAx,'A Comparison of Data Sets')
```



Input Arguments

X — Data to display

matrix

Data to display, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Y — Data to plot against X

matrix

Data to plot against `X`, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`LineSpec` — Line style, marker symbol, and color for scatter plots

string

Line style, marker symbol, and color for the scatter plots, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

Data Types: `char`

Output Arguments

`H` — Line object handles

matrix

Line object handles, returned as a matrix. This is a unique identifier, which you can use to query and modify the properties of a specific line object. The line objects are used to create the scatter plots.

`AX` — Subaxes handles

matrix

Subaxes handles, returned as a matrix. This is a unique identifier, which you can use to query and modify the properties of a specific subaxes.

`BigAx` — Big axes handle

scalar

Big axes handle, returned as a scalar. This is a unique identifier, which you can use to query and modify properties of the big axes. `BigAx` is left as the current axes (`gca`) so that a subsequent `title`, `xlabel`, or `ylabel` command will center text with respect to the big axes.

`P` — Patch object handles

vector | []

Patch object handles, returned as a vector or `[]`. If histogram plots are created, then `P` is returned as a vector of patch object handles for the histogram plots. These are unique

identifiers, which you can use to query and modify the properties of a specific patch object. If no histogram plots are created, then `P` is returned as empty brackets.

`PAx` — Handle to invisible histogram axes

vector | []

Handle to invisible histogram axes, returned as a vector or `[ ]`. If histogram plots are created, then `PAx` is returned as a vector of histogram axes handles. These are unique identifiers, which you can use to query and modify the properties of a specific axes, such as the axes scale. If no histogram plots are created, then `PAx` is returned as empty brackets.

See Also

`scatter` | `scatter3`

Introduced before R2006a

plottools

Show or hide plot tools

Syntax

```
plottools('on')
plottools('off')
plottools
plottools(figure_handle,...)
plottools(...,'tool')
```

Description

`plottools('on')` displays the Figure Palette, Plot Browser, and Property Editor on the current figure, configured as you last used them.

`plottools('off')` hides the Figure Palette, Plot Browser, and Property Editor on the current figure.

`plottools` with no arguments, is the same as `plottools('on')`

`plottools(figure_handle,...)` displays or hides the plot tools on the specified figure instead of on the current figure.

`plottools(...,'tool')` operates on the specified tool only. *`tool`* can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

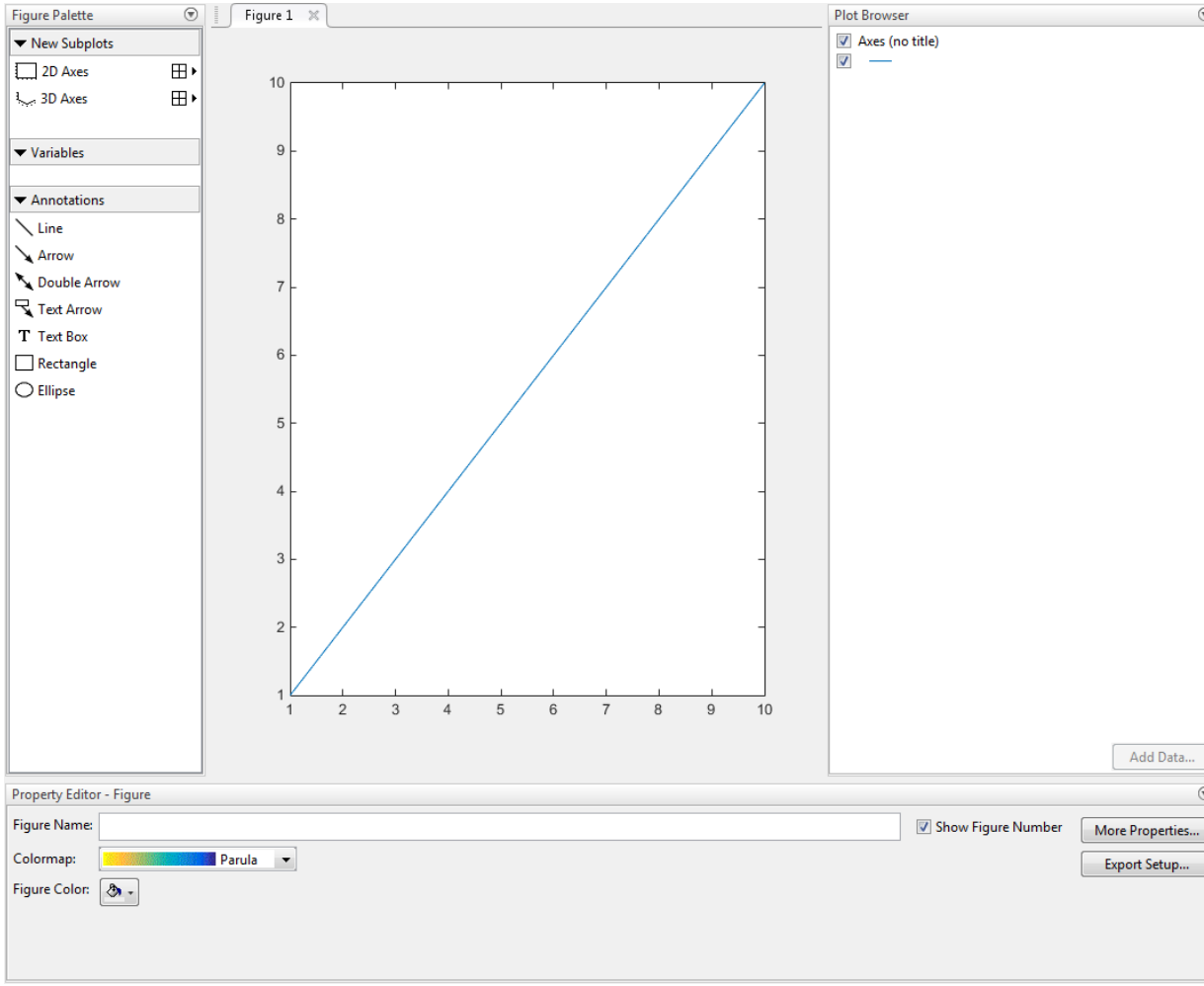
Note: The first time you open the plotting tools, all three of them appear, grouped around the current figure as shown above. If you close, move, or undock any of the tools, MATLAB remembers the configuration you left them in and restores it when you invoke the tools for subsequent figures, both within and across MATLAB sessions.

Examples

Open Plot Tools

Create a line plot and open the plot tools.

```
plot(1:10);
plottools('on')
```



Alternatives

Click the larger **Plotting Tools** icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Individually select

the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see “Customize Graph Using Plot Tools”.

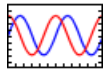
See Also

`figurepalette` | `plotbrowser` | `propertyeditor`

Introduced before R2006a

plotyy

2-D line plots with y-axes on both left and right side



Syntax

```
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,function)
plotyy(X1,Y1,X2,Y2,'function1','function2')
[AX,H1,H2] = plotyy(...)
```

Description

`plotyy(X1,Y1,X2,Y2)` plots `Y1` versus `X1` with *y*-axis labeling on the left and plots `Y2` versus `X2` with *y*-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,function)` uses the specified plotting function to produce the graph.

`function` can be either a function handle or a string specifying `plot`, `semilogx`, `semilogy`, `loglog`, `stem`, or any MATLAB function that accepts the syntax

```
h = function(x,y)
```

For example,

```
plotyy(x1,y1,x2,y2,@loglog) % function handle
plotyy(x1,y1,x2,y2,'loglog') % string
```

Function handles enable you to access user-defined local functions and can provide other advantages. See @ for more information on using function handles.

`plotyy(X1,Y1,X2,Y2,'function1','function2')` uses `function1(X1,Y1)` to plot the data for the left axis and `function2(X2,Y2)` to plot the data for the right axis.

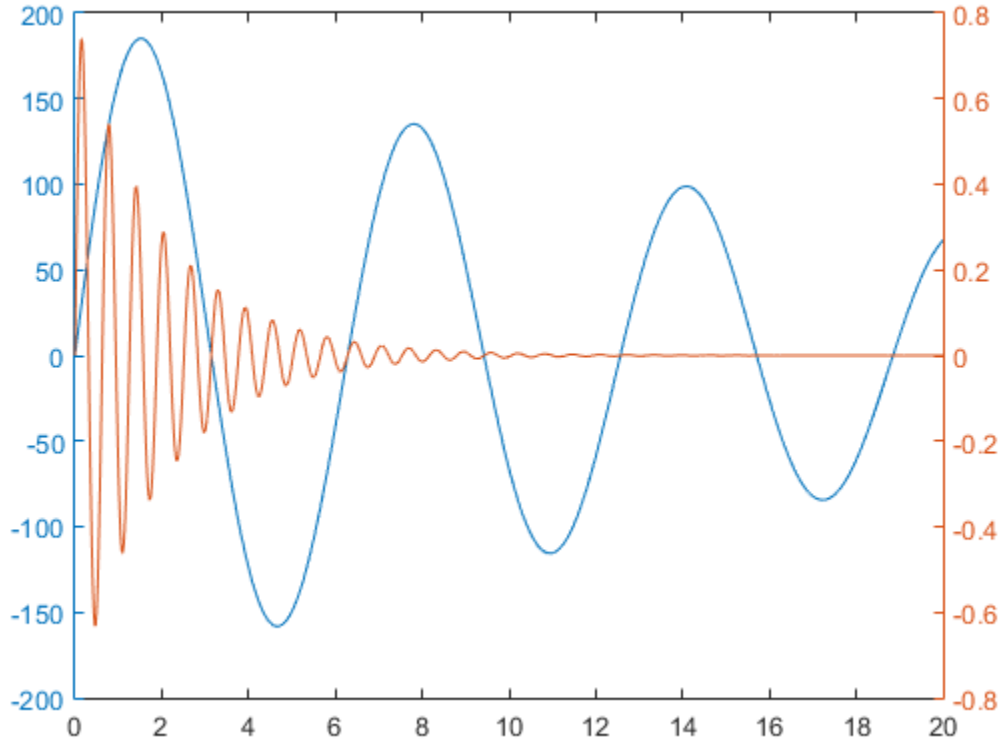
`[AX,H1,H2] = plotyy(...)` returns the handles of the two axes created in `AX` and the handles of the graphics objects from each plot in `H1` and `H2`. `AX(1)` is the left axes and `AX(2)` is the right axes.

Examples

Plot Two Data Sets with Different y-Axes

Plot two data sets on one graph using two *y*-axes.

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

figure % new figure
plotyy(x,y1,x,y2)
```

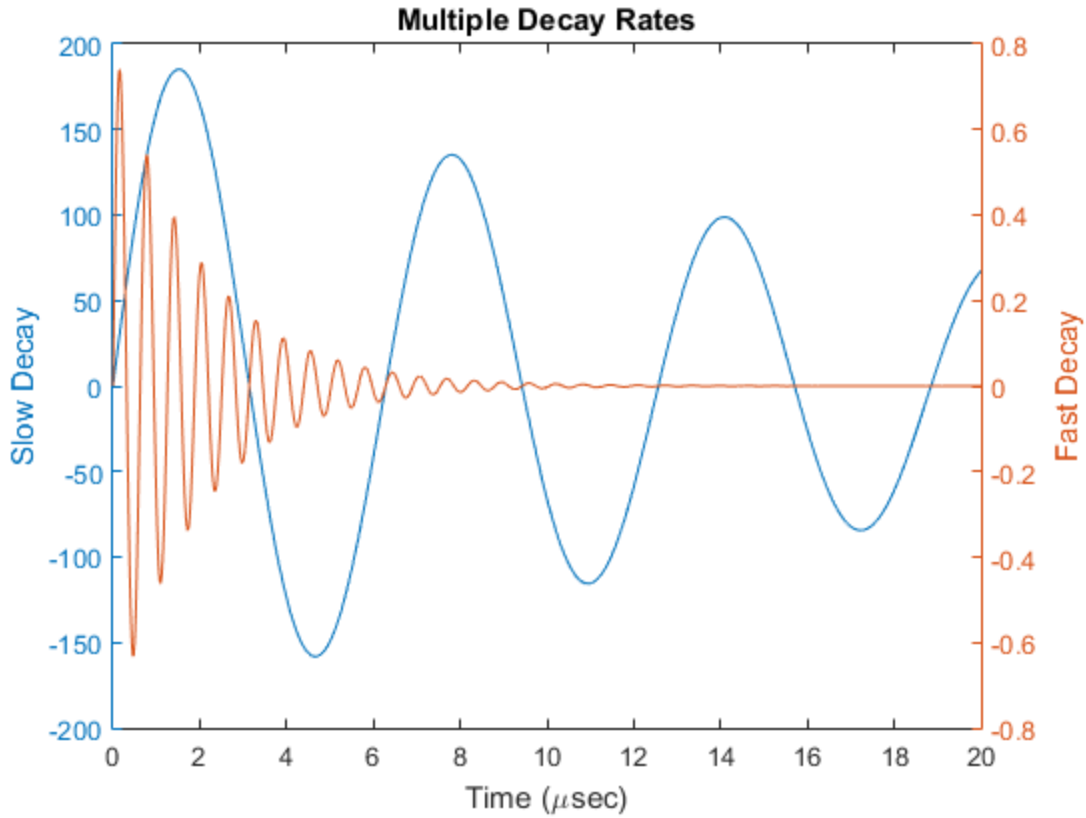


Add Title and Axis Labels

Plot two data sets using a graph with two *y*-axes. Add a title and axis labels.

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

figure % new figure
[hAx,hLine1,hLine2] = plotyy(x,y1,x,y2);

title('Multiple Decay Rates')
xlabel('Time (\musec)')
```

```
ylabel(hAx(1),'Slow Decay') % left y-axis
ylabel(hAx(2),'Fast Decay') % right y-axis
```

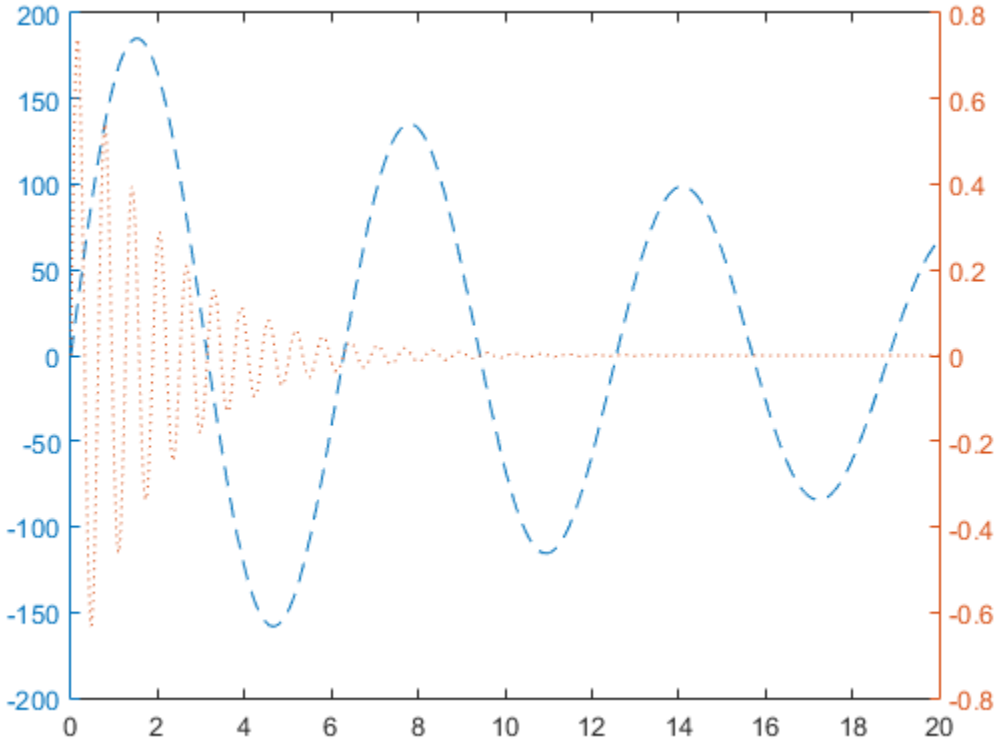


Change Line Styles

Plot two data sets using a graph with two *y*-axes. Change the line styles. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
```

```
[hAx,hLine1,hLine2] = plotyy(x,y1,x,y2);
hLine1.LineStyle = '--';
hLine2.LineStyle = ':';
```

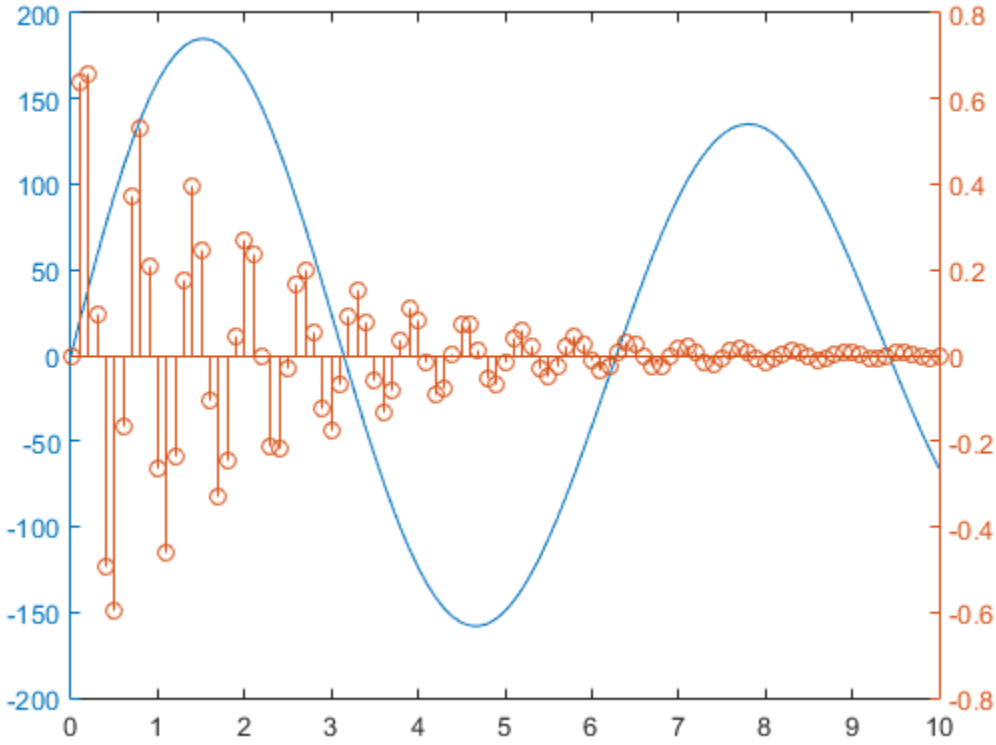


Combine Different Types of Plots

Plot two data sets using a graph with two y-axes. Use a line plot for the data associated with the left *y*-axes. Use a stem plot for the data associated with the right *y*-axes.

```
x = 0:0.1:10;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

figure
```

```
plotyy(x,y1,x,y2,'plot','stem')
```

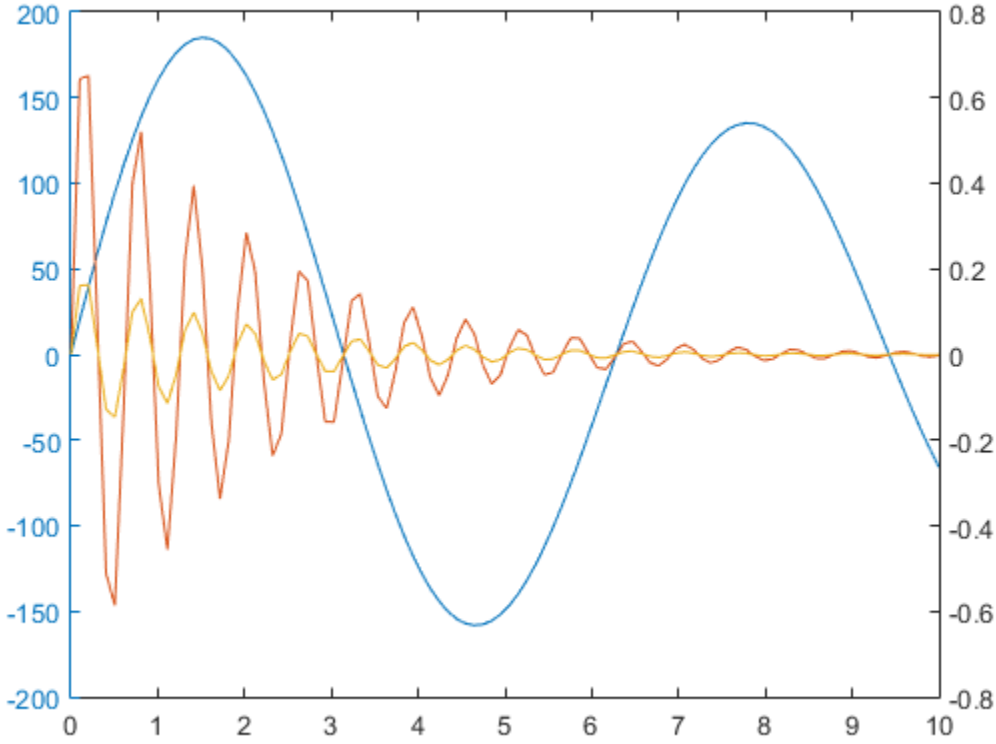


Use Right y-Axis for Two Data Sets

Plot three data sets using a graph with two *y*-axes. Plot one set of data associated with the left *y*-axis. Plot two sets of data associated with the right *y*-axis by using two-column matrices.

```
x = linspace(0,10);
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
y3 = 0.2*exp(-0.5*x).*sin(10*x);

figure
```

```
[hAx,hLine1,hLine2] = plotyy(x,y1,[x',x'],[y2',y3']);
```



More About

- "Create Graph with Two y-Axes"
- "Graph with Multiple x-Axes and y-Axes"

See Also

Functions

`linkaxes` | `linkprop` | `loglog` | `plot` | `semilogx` | `semilogy`

Properties

Chart Line Properties | Stem Series Properties

Introduced before R2006a

matlab.unittest.plugins

Summary of classes in MATLAB Plugins Interface

Description

Plugins customize a `TestRunner` object. The `matlab.unittest.plugins` package consists of the following customized MATLAB plugins.

matlab.unittest.plugins.CodeCoveragePlugin	Plugin that produces a code coverage report
matlab.unittest.plugins.DiagnosticsValidationPlugin	Plugin to help validate diagnostic code
matlab.unittest.plugins.FailureDiagnosticsPlugin	Plugin to show diagnostics on failure
matlab.unittest.plugins.LoggingPlugin	Plugin to report diagnostic messages
matlab.unittest.plugins.OutputStream	Interface that determines where to send text output
matlab.unittest.plugins.StopOnFailuresPlugin	Plugin to debug test failures
matlab.unittest.plugins.TAPPlugin	Plugin that produces Test Anything Protocol stream
matlab.unittest.plugins.ToFile	Output stream to write text output to file
matlab.unittest.plugins.ToStandardOutput	Output stream to display text information to screen
matlab.unittest.plugins.TestRunnerPlugin	Plugin interface for extending TestRunner
matlab.unittest.plugins.TestRunProgressPlugin	Plugin that reports test run progress

Related Examples

- “Add Plugin to Test Runner”
- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”
- “Write Plugin to Save Diagnostic Details”

matlab.unittest.plugins.CodeCoveragePlugin class

Package: matlab.unittest.plugins

Plugin that produces a code coverage report

Description

Add the `CodeCoveragePlugin` to the `TestRunner` to produce a line coverage report for MATLAB source code. The testing framework runs the tests, and the resulting coverage report indicates the executed lines of code. The coverage report is based on source code located in one or more folders or packages.

The `CodeCoveragePlugin` uses the MATLAB profiler to determine which lines of code the tests execute. The tests and source code should not interact with the profiler. Prior to running a suite of tests, the plugin clears any data collected by the profiler.

Construction

`matlab.unittest.plugins.CodeCoveragePlugin.forFolder(folder)` creates a plugin that produces a code coverage report for one or more folders. The plugin reports on the source code inside `folder`.

`matlab.unittest.plugins.CodeCoveragePlugin.forPackage(package)` creates a plugin that produces a code coverage report for one or more packages. The plugin reports on the source code contained in `package`.

Input Arguments

`folder` — Location(s) of folder containing source code

string | cell array of strings

Location(s) of folder containing source code, specified as a string or cell array of strings. `folder` is the absolute or relative path to one or more folders. If you specify multiple folders, MATLAB opens a profile coverage report for each folder.

Example: `'C:\projects\myproj'`

Example: `pwd`

Example: `{'C:\projects\myprojA','myprojB'}`

`package` — Name(s) of package containing source code

string | cell array of strings

Name(s) of package containing source code, specified as a string or cell array of strings. If you specify multiple packages, MATLAB opens a profile coverage report for each package.

Example: `'myproject.controller'`

Example: `{'myprojA','myprojB'}`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Generate Code Coverage Report

In an new file, `quadraticSolver.m`, in your working folder, create the following function.

```
function roots = quadraticSolver(a,b,c)
% quadraticSolver returns solutions to the
% quadratic equation a*x^2 + b*x + c = 0.

checkInputs

roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);

    function checkInputs
        if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
            error('quadraticSolver:InputMustBeNumeric', ...
                'Coefficients must be numeric.')
        end
```

```
    end
end
```

Create a test for the quadratic solver. In a `tests` subfolder, create `SolverTest.m` containing the following test class.

```
classdef SolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function testImaginarySolution(testCase)
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i, -1-3i];
            testCase.verifyEqual(actSolution,expSolution)
        end
    end

end
```

At the command prompt, create a test suite from the `tests` folder.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.CodeCoveragePlugin

suite = TestSuite.fromFolder('tests');
```

Create a test runner.

```
runner = TestRunner.withTextOutput;
```

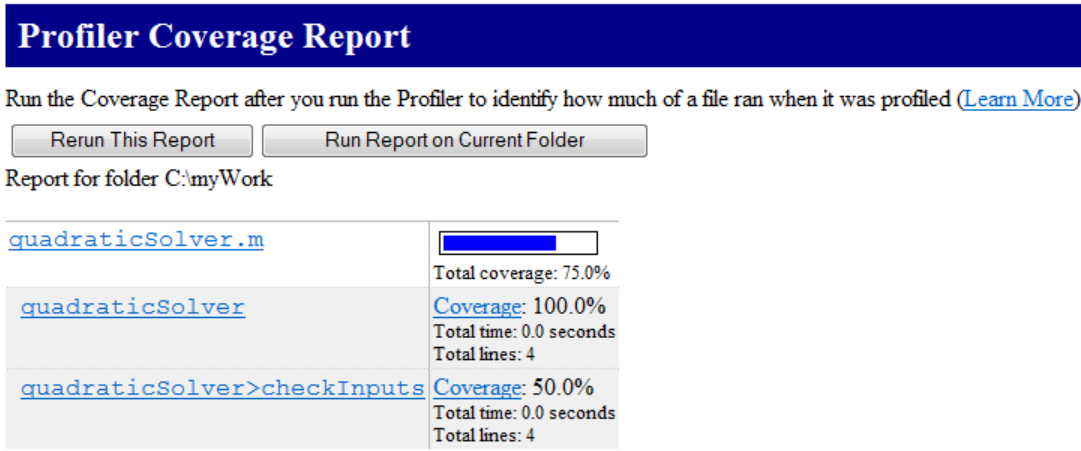
Add `CodeCoveragePlugin` to the runner and run the tests. Specify that the source code folder is your current working folder. If you have other source code files in your current working folder, they will show up in the coverage report.

```
runner.addPlugin(CodeCoveragePlugin.forFolder(pwd))
result = runner.run(suite);
```

```
Running SolverTest
```

```
..
Done SolverTest
__________
```

MATLAB opens a Profiler Coverage Report for the quadratic solver function.



The `checkinputs` nested function does not have complete code coverage. Since the tests in `SolverTest.m` do not pass nonnumeric input to `quadraticSolver`, MATLAB does not exercise the code that throws an error if the inputs are not numeric. To address the missing coverage, add a test method to test the error condition.

See Also

`matlab.unittest.plugins.TestRunnerPlugin` | `profile`

Introduced in R2014b

matlab.unittest.plugins.DiagnosticsValidationPlugin class

Package: matlab.unittest.plugins

Plugin to help validate diagnostic code

Description

The `DiagnosticsValidationPlugin` creates a plugin to help validate diagnostic code.

Add the `DiagnosticsValidationPlugin` to the `TestRunner` to confirm that user-supplied diagnostics execute correctly. This plugin is useful because typically tests do not encounter failure conditions. A failure can result in unexercised diagnostic code. If a programming error exists in this diagnostic code, the error is not evident unless the test fails. However, at this point in the testing process, the diagnostics for the failure condition are lost due to the error in the diagnostic code.

Use this plugin to unconditionally evaluate the diagnostics supplied by the test writer, regardless of whether the test results in a passing or failing condition. This approach helps you to confirm that all of the diagnostic code is free from programming errors.

The diagnostic analysis can reduce the test performance and can result in very verbose text output. Be aware of these impacts before using this plugin for routine testing.

Construction

`matlab.unittest.plugins.DiagnosticsValidationPlugin` creates a plugin to help validate diagnostic code.

`matlab.unittest.plugins.DiagnosticsValidationPlugin(stream)` redirects all the text output to the output stream, `stream`. If you do not specify the output stream, the plugin uses the default `ToStandardOutput` stream.

Input Arguments

`stream`

Location where the plugin directs text output, specified as an `OutputStream`.

Default: `ToStandardOutput`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Add Plugin to TestRunner

In your working folder, create a file, `ExampleTest.m`, containing the following test class. In this example, the `testThree` method has an intentional error. The method should use a function handle to the `dir` function as a `FunctionHandleDiagnostic`, but `dir` is misspelled.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            % test code
        end
        function testTwo(testCase)
            % test code
        end
        function testThree(testCase)
            % The following should use @dir as a function handle,
            % but there is a typo
            testCase.verifyEqual('myfile','myfile', @dri)
        end
    end
end
```

All of the tests in `ExampleTest.m` result in a passing condition, but there is an error in the diagnostic.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.DiagnosticsValidationPlugin

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins.

```
runner = TestRunner.withTextOutput;
```

Run the tests.

```
result1 = runner.run(suite);
```

```
Running ExampleTest
...
Done ExampleTest
__________
```

No diagnostic output is displayed because all the tests passed. The testing framework does not encounter the bug in the `FunctionHandleDiagnostic` of `testThree`.

Add `DiagnosticValidationPlugin` to the runner and run the tests.

```
runner.addPlugin(DiagnosticsValidationPlugin)
result2 = runner.run(suite);
```

```
Running ExampleTest
..
------------------------------
Validation of Test Diagnostic:
------------------------------
Error occurred while capturing diagnostics:
Error using evalc
Undefined function or variable 'dri'.

Error in ExampleTest/testThree (line 12)
            testCase.verifyEqual('myfile','myfile', @dri);

.
Done ExampleTest
__________
```

The framework executes the diagnostic provided by the `FunctionHandleDiagnostic`, even though none of the tests fails. Without this plugin, the test framework only encounters the bug if the test fails.

See Also

`matlab.unittest.diagnostics` | `matlab.unittest.plugins` | `OutputStream` | `ToStandardOutput`

matlab.unittest.plugins.FailureDiagnosticsPlugin class

Package: matlab.unittest.plugins

Plugin to show diagnostics on failure

Description

The `FailureDiagnosticsPlugin` creates a plugin to show diagnostics upon encountering a test failure. Add it to the `TestRunner` to output test failure diagnostics to the Command Window. This plugin is used by default when you construct a test runner using `TestRunner.withTextOutput`.

Construction

`matlab.unittest.plugins.FailureDiagnosticsPlugin` creates a plugin to show diagnostics upon encountering a test failure.

`matlab.unittest.plugins.FailureDiagnosticsPlugin(stream)` redirects all the text output to the output stream, `stream`. If you do not specify the output stream, the plugin uses the `ToStandardOutput` stream.

Input Arguments

`stream`

Location where the plugin directs text output, specified as an `OutputStream`.

Default: `ToStandardOutput`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Add Plugin to TestRunner

In your working folder, create a file, `ExampleTest.m`, containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4')
        end
         function testTwo(testCase)  % Test passes
             testCase.verifyEqual(5, 5, 'Testing 5==5')
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure.
The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.FailureDiagnosticsPlugin

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins.

```
runner = TestRunner.withNoPlugins;
```

Run the tests.

```
result1 = runner.run(suite);
```

No output is displayed, but `result1` contains information about the failed test.

Add `FailureDiagnosticsPlugin` to the runner and run the tests.

```
runner.addPlugin(FailureDiagnosticsPlugin)
result2 = runner.run(suite);
```

```
================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        5         4           1        0.25

    Actual Value:
             5
    Expected Value:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 7
================================================================================
Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                Failed by verification.
```

The framework displays the `DiagnosticResult` of the `StringDiagnostic` for failed tests only. It also displays additional framework diagnostics. The `TestResult` object, `result2`, is the same as `result1`.

See Also

`matlab.unittest.diagnostics` | `matlab.unittest.plugins` | `OutputStream` | `ToStandardOutput`

Introduced in R2013a

matlab.unittest.plugins.LoggingPlugin class

Package: matlab.unittest.plugins

Plugin to report diagnostic messages

Description

The `LoggingPlugin` creates a plugin to report diagnostic messages that are created by the `log` method of a `TestCase` or `Fixture`.

Construction

Instantiate a `LoggingPlugin` using one of its static methods.

Use the `withVerbosity` static method to configure a plugin to respond to messages of a particular verbosity. Also, the `withVerbosity` method accepts a number of name/value pairs to configure the format for reporting logged messages.

Properties

`Description` — Logged diagnostic message description

`'Diagnostic logged'` (default) | string

Logged diagnostic message description, specified as a string. The value of this property is printed alongside each logged diagnostic message. `Description` is read only, and its value is set during construction.

`HideLevel` — Indicator to display verbosity level

`false` (default) | `true`

Indicator to display the verbosity level alongside each logged diagnostic, specified as `false` (`logical(0)`) or `true` (`logical(1)`). By default, this property is `false` and the test framework displays the verbosity level. `HideLevel` is read only, and its value is set during construction.

`HideTimestamp` — Indicator to display timestamp

`false` (default) | `true`

Indicator to display the timestamp from when the test framework generates the logged message alongside each logged diagnostic, specified as `false` (`logical(0)`) or `true` (`logical(1)`). By default, this property is `false` and the test framework displays the timestamp. `HideTimestamp` is read only, and its value is set during construction.

`NumStackFrames` — Number of stack frames to display

`0` (default) | integer value | `Inf`

Number of stack frames to display after each logged diagnostic message, specified as an integer value. By default, this property is `0`, and the test framework does not display stack information. If `NumStackFrames` is `Inf`, the test framework displays all available stack frames. `NumStackFrames` is read only, and its value is set during construction.

`Verbosity` — Verbosity levels supported by plugin instance

array of `matlab.unittest.Verbosity` instances

Verbosity levels supported by plugin instance, specified as an array of `matlab.unittest.Verbosity` instances. The plugin reacts to diagnostics that are logged at a verbosity level listed in this array. `Verbosity` is read only, and its value is set during construction.

Methods

withVerbosity	Construct LoggingPlugin for messages of specified verbosity

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`matlab.unittest.Verbosity` | `matlab.unittest.fixtures.Fixture.log` | `matlab.unittest.TestCase.log`

Introduced in R2014b

matlab.unittest.plugins.LoggingPlugin.withVerbosity

Class: matlab.unittest.plugins.LoggingPlugin
Package: matlab.unittest.plugins

Construct LoggingPlugin for messages of specified verbosity

Syntax

```
matlab.unittest.plugins.LoggingPlugin.withVerbosity(v)
matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,stream)
matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,Name,Value)
```

Description

`matlab.unittest.plugins.LoggingPlugin.withVerbosity(v)` constructs a `LoggingPlugin` for messages of the specified verbosity.

`matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,stream)` redirects the text output to the output stream.

`matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,Name,Value)` includes additional options specified by one or more Name,Value pair arguments.

Input Arguments

v — Verbosity levels supported by plugin instance

`1 | 2 | 3 | 4 | matlab.unittest.Verbosity` enumeration

Verbosity levels supported by the plugin instance, specified as an integer value between 1 and 4 or a `matlab.unittest.Verbosity` enumeration object. The plugin reacts to diagnostics that are logged at this level and lower. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Corresponding Enumeration Object	Verbosity Description
1	`matlab.unittest.Verbosity.Ter`	minimal amount of information

Numeric Representation	Corresponding Enumeration Object	Verbosity Description
2	`matlab.unittest.Verbosity.Con`	typical amount of information
3	`matlab.unittest.Verbosity.Det`	supplemental amount of information
4	`matlab.unittest.Verbosity.Ver`	surplus of information

`stream` — Location where plugin directs text output
`ToStandardOutput` instance (default) | `OutputStream` instance

Location where the plugin directs text output, specified as an `OutputStream` instance. By default, the plugin uses the `OutputStream` subclass `ToStandardOutput` as the stream.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Description'` — Logged diagnostic message description
`'Diagnostic logged'` (default) | string

Logged diagnostic message description, specified as a string. This value is printed alongside each logged diagnostic message. If the value is an empty string, the test framework does not display a description.

`'ExcludingLowerLevels'` — Indicator to display messages logged at levels lower than the verbosity level
`false` (default) | `true`

Indicator to display messages logged at levels lower than the verbosity level, v, specified as `false` or `true` (`logical(0)` or `logical(1)`). By default, the value is `false` and the plugin reacts to all messages logged at level v or lower. If the value is `true`, the plugin reacts only to messages logged at level `v`.

`'HideLevel'` — Indicator to display verbosity level
`false` (default) | `true`

Indicator to display the verbosity level alongside each logged diagnostic, specified as `false` or `true` (`logical(0)` or `logical(1)`). By default, the value is `false` and the test framework displays the verbosity level.

`'HideTimestamp'` — Indicator to display timestamp

`false` (default) | `true`

Indicator to display the timestamp from when the test framework generates the logged message alongside each logged diagnostic, specified as `false` or `true` (`logical(0)` or `logical(1)`). By default, the value is `false` and the test framework displays the timestamp.

`'NumStackFrames'` — Number of stack frames to display

`0` (default) | integer value | `Inf`

Number of stack frames to display after each logged diagnostic message, specified as an integer value. By default, the value is `0`, and the test framework does not display stack information. If `NumStackFrames` is `Inf`, the test framework displays all available stack frames.

Examples

Create Logging Plugin

Create a function-based test in a file, `sampleLogTest.m`, in your working folder.

```
function tests = sampleLogTest
tests = functiontests(localfunctions);

function svdTest(testCase)
import matlab.unittest.Verbosity

log(testCase,'Generating matrix.');
m = rand(1000);

log(testCase,1,'About to call SVD.');
[U,S,V] = svd(m);

log(testCase,Verbosity.Terse,'SVD finished.');

verifyEqual(testCase,U*S*V',m,'AbsTol',1e-6)
```

At the command prompt, run the test.

```
results = run(sampleLogTest);
```

```
Running sampleLogTest
   [Terse] Diagnostic logged (2014-04-14T14:20:59): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:20:59): SVD finished.
.
Done sampleLogTest
__________
```

The default runner reports the diagnostics at level 1 (`Terse`).

Create a test runner to report the diagnostics at levels 1 and 2, and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(2);
runner.addPlugin(p);

results = runner.run(sampleLogTest);
```

```
 [Concise] Diagnostic logged (2014-04-14T14:28:14): Generating matrix.
   [Terse] Diagnostic logged (2014-04-14T14:28:14): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:28:15): SVD finished.
```

Configure Logged Message Output

Create the following class In a file in your current working folder, `ExampleLogTest.m`.

```
classdef ExampleLogTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            log(testCase,3,'Starting Test')
            log(testCase,'Testing 5==4')
            testCase.verifyEqual(5,4)
            log(testCase,4,'Test Complete')
        end
        function testTwo(testCase)  % Test passes
            log(testCase,matlab.unittest.Verbosity.Detailed,'Starting Test')
            log(testCase,'Testing 5==5')
            testCase.verifyEqual(5,5)
            log(testCase,matlab.unittest.Verbosity.Verbose,'Test Complete')
```

```
        end
    end
end
```

The log messages in `testTwo` uses `Verbosity` enumerations instead of the corresponding integers.

At the command prompt, create the test suite and a runner at verbosity level 4, and then run the test.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin
suite   = TestSuite.fromClass(?ExampleLogTest);

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(4);
runner.addPlugin(p);

results = runner.run(suite);
```

```
[Detailed] Diagnostic logged (2014-04-14T15:24:03): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:24:03): Testing 5==4
 [Verbose] Diagnostic logged (2014-04-14T15:24:03): Test Complete
[Detailed] Diagnostic logged (2014-04-14T15:24:03): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:24:03): Testing 5==5
 [Verbose] Diagnostic logged (2014-04-14T15:24:03): Test Complete
```

Create a new plugin to direct the output to a file, `myOutput.log`, and rerun the tests.

```
import matlab.unittest.plugins.ToFile
outFile = 'myOutput.log';

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(4,ToFile(outFile));
runner.addPlugin(p);

results = runner.run(suite);
```

Observe the contents in the file created by the plugin.

```
disp(fileread(outFile))
```

```
[Detailed] Diagnostic logged (2014-04-14T15:27:44): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:27:44): Testing 5==4
```

```
 [Verbose] Diagnostic logged (2014-04-14T15:27:44): Test Complete
[Detailed] Diagnostic logged (2014-04-14T15:27:44): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:27:44): Testing 5==5
 [Verbose] Diagnostic logged (2014-04-14T15:27:44): Test Complete
```

Create a new plugin that does not display level 4 messages. Do not display the verbosity level or timestamp. Rerun the tests.

```
runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(matlab.unittest.Verbosity.Detailed,...
    'HideLevel',true,'HideTimestamp',true);
runner.addPlugin(p);

results = runner.run(suite);
```

```
Diagnostic logged: Starting Test
Diagnostic logged: Testing 5==4
Diagnostic logged: Starting Test
Diagnostic logged: Testing 5==5
```

See Also

`matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.ToStandardOutput` | `matlab.unittest.Verbosity` | `matlab.unittest.fixtures.Fixture.log` | `matlab.unittest.TestCase.log`

Introduced in R2014b

matlab.unittest.plugins.OutputStream class

Package: matlab.unittest.plugins

Interface that determines where to send text output

Description

The `OutputStream` interface is an abstract interface class that you can use as a base class to specify where plugins direct their text output. To create a custom output stream, implement a `print` method that correctly handles the formatted text information the testing framework passes to it. Many text-oriented plugins accept an `OutputStream` to redirect the text they produce in a configurable manner.

Methods

print	Print string to output stream

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Custom Output Stream

In a file in your working directory, create a new output stream class in the file `ToFigure.m`. This class allows plugin output to be redirected to a figure.

```
classdef ToFigure < matlab.unittest.plugins.OutputStream

    properties(SetAccess=private)
```

```
    Figure
end
properties(Access=private)
    ListBox
end
```

This class uses two properties. `Figure` is the figure that receives and displays the output. `ListBox` is a handle to the list box that displays the text.

In the same file, add the following `methods` block.

```
methods
    function print(stream,formatSpec,varargin)
        % Create the figure
        if isempty(stream.Figure) || ~ishghandle(stream.Figure)
            stream.createFigure
        end
        newStr = sprintf(formatSpec,varargin{:});
        oldStr = strjoin(stream.ListBox.String', '\n');

        % Create the full string
        fullStr = [oldStr,newStr];
        fullStrCell = strsplit(fullStr,'\n','CollapseDelimiters',false);

        % Set the string and selection
        stream.ListBox.String = fullStrCell';
        stream.ListBox.Value = numel(fullStrCell);
        drawnow
    end
end
```

You must implement the `print` method for any subclass of `OutputStream`. In this example, the method creates a new figure (if necessary), formats the incoming text, and then adds it to the output stream.

In the same file, add the following `methods` block containing a helper function to create the figure.

```
methods(Access=private)
    function createFigure(stream)
        stream.Figure = figure(...
            'Name',         'Unit Test Output', ...
            'WindowStyle',  'docked');
```

```
            stream.ListBox = uicontrol( ...
                'Parent',       stream.Figure, ...
                'Style',        'listbox', ...
                'String',       {}, ...
                'Units',        'normalized', ...
                'Position',     [.05 .05 .9 .9], ...
                'Max',          2, ...
                'FontName',     'Monospaced', ...
                'FontSize',      13);
        end
    end
end
```

In an new file in your working folder, create `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4' );
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5' );
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure.
The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.DiagnosticsValidationPlugin

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window.

```
runner = TestRunner.withTextOutput;
```

Create a `DiagnosticsValidationPlugin` that explicitly specifies that its output should go to a figure via the `ToFigure` output stream.

```
plugin = DiagnosticsValidationPlugin(ToFigure);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        5         4           1        0.25

    Actual Value:
             5
    Expected Value:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
...
Done ExampleTest
__________
```

```
Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Only the test failures produce output to the screen. By default, `TestRunner.withTextOutput` uses a `FailureDiagnosticsPlugin` to display output on the screen.

In addition to the default text output being displayed on the screen, the `DiagnosticsValidationPlugin` output is directed to a docked figure. The figure shows the following text.

```
------------------------------
Validation of Test Diagnostic:
------------------------------
Testing 5==4
------------------------------
Validation of Test Diagnostic:
------------------------------
Testing 5==5
```

The `DiagnosticsValidationPlugin` displays the diagnostic information regardless of whether the tests encounter failure conditions.

See Also

`matlab.unittest.plugins.OutputStream` | `fprintf` | `matlab.unittest.plugins`

Introduced in R2014a

print

Class: matlab.unittest.plugins.OutputStream
Package: matlab.unittest.plugins

Print string to output stream

Syntax

```
print(stream,formatSpec,A1,...,An)
```

Description

`print(stream,formatSpec,A1,...,An)` formats the data in arrays `A1,...,An` according to `formatSpec`, and sends the result to the output stream, `stream`. Assign `formatSpec` and `A1,...,An` using the same interface that you use for `sprintf` and `fprintf`.

Input Arguments

`stream`

Output stream, specified as an instance of the `OutputStream` class

`formatSpec`

Format of text in output stream, specified as a string. For information on the construction of `formatSpec` string, see the input argument entry on the `fprintf` or `sprint` reference pages.

`A`

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array.

See Also

`fprintf` | `sprintf`

Introduced in R2014a

matlab.unittest.plugins.StopOnFailuresPlugin class

Package: matlab.unittest.plugins

Plugin to debug test failures

Description

The `StopOnFailuresPlugin` class provides a plugin to help debug test failures. Adding `StopOnFailuresPlugin` to the test runner pauses execution of a test if it encounters a qualification failure or uncaught error and puts MATLAB into debug mode.

If `StopOnFailuresPlugin` encounters a qualification failure or uncaught error in a test, you can use MATLAB debugging commands, such as `dpbup`, `dbstep`, `dbcont`, and `dbquit`, to investigate the cause of the test failure.

If `StopOnFailuresPlugin` encounters an uncaught error in a test, you cannot use `dbup` to shift context to the source of the error because the error disrupts the stack.

Construction

`matlab.unittest.plugins.StopOnFailuresPlugin` creates a plugin to debug test failures.

`matlab.unittest.plugins.StopOnFailuresPlugin('IncludingAssumptionFailures', tf)` indicates whether to react to assumption failures. By default, `StopOnFailuresPlugin` reacts to only uncaught errors and verification, assertion, and fatal assertion qualification errors. However, when `'IncludingAssumptionFailures'` is set to `true`, the plugin also reacts to assumption failures.

Input Arguments

`tf` — Indicator to react to assumption failures

FALSE (default) | TRUE

Indicator to react to assumption failures, specified as logical `false` or `true`. When this value is `true`, the test runner reacts to assumption failures. When the value is `false`, the plugin ignores assumption failures.

Properties

`IncludeAssumptionFailures`

When this property value is `true`, the instance reacts to assumption failures. When the value is `false`, the instance ignores assumption failures. The `IncludeAssumptionFailures` property is `false` by default. To specify the property as `true`, use the `InlcudingAssumptionFailures` input when you construct the instance.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Add Plugin to TestRunner

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            act = 3.1416;
            exp = pi;
            testCase.verifyEqual(act, exp)
        end
        function testTwo(testCase)  % Test does not complete
            testCase.assumeEqual(5, 4)
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class and a test runner.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.StopOnFailuresPlugin
```

```
suite = TestSuite.fromClass(?ExampleTest);
runner = TestRunner.withTextOutput;
```

Run the tests.

```
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual        Expected              Error               Relat
                _____    ______    _________________    ____________________    _____

                1        3.1416    3.14159265358979    7.34641020683213e-06    2.33843

    Actual Value:
           3.141600000000000
    Expected Value:
           3.141592653589793

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 6
================================================================================
.
================================================================================
ExampleTest/testTwo was filtered.
    Details
================================================================================
.
Done ExampleTest
__________

Failure Summary:
```

```
 Name                 Failed  Incomplete  Reason(s)
=================================================================
 ExampleTest/testOne    X                 Failed by verification.
-----------------------------------------------------------------
 ExampleTest/testTwo              X       Filtered by assumption.
```

As a result of the qualifications in the test class, the first test fails, and the second test does not complete.

Add the `StopOnFailuresPlugin` to the runner and run the tests.

```
runner.addPlugin(StopOnFailuresPlugin)
result = runner.run(suite);
```

```
Running ExampleTest
Test execution paused due to failure. Either click here or execute DBUP 9 times to shi
```

During the test execution, when the failure occurs, MATLAB enters debug mode.

Click on the hyperlinked word `'here'` to shift debug context to your work source. If necessary, make the command window your current window.

```
In workspace belonging to ExampleTest>ExampleTest.testOne at 6
```

Examine the variables in the workspace.

```
whos
```

```
  Name          Size            Bytes  Class           Attributes

  act           1x1                 8  double
  exp           1x1                 8  double
  testCase      1x1               112  ExampleTest
```

Now, you can investigate the cause of the test failure.

For example, see if the test passes when you specify a relative tolerance of `100*eps`.

```
testCase.verifyEqual(act,exp,'RelTol',100*eps)
```

```
================================================================================
Verification failed in ExampleTest/testOne.

    ---------------------
    Framework Diagnostic:
```

```
    ---------------------
    IsEqualTo failed.
    --> NumericComparator failed.
        --> The values are not equal using "isequaln".
        --> RelativeTolerance failed.
            --> The error was not within relative tolerance.
            --> Failure table:
                        Index    Actual        Expected               Error
                        _____    ______    _________________    ____________________

                        1        3.1416    3.14159265358979    7.34641020683213e-06

    Actual Value:
           3.141600000000000
    Expected Value:
           3.141592653589793

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 6
================================================================================
```

The test fails even with the specified tolerance.

Exit out of debug mode.

```
dbquit
```

```
================================================================================
Verification failed in ExampleTest/testOne.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual        Expected               Error                Relat
                _____    ______    _________________    ____________________    _______

                1        3.1416    3.14159265358979    7.34641020683213e-06    2.33843

    Actual Value:
           3.141600000000000
```

```
    Expected Value:
           3.141592653589793

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 6
================================================================================
.
================================================================================
ExampleTest/testTwo was filtered.
    Details
================================================================================
.
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    =====================================================================
     ExampleTest/testOne    X                 Failed by verification.
    ---------------------------------------------------------------------
     ExampleTest/testTwo              X       Filtered by assumption.
```

To enter debug mode for tests that fail by assumption, such as `testTwo` in the `ExampleTest` class, include `'IncludingAssumptionFailures'` option for the plugin.

```
runner = TestRunner.withTextOutput;
runner.addPlugin(StopOnFailuresPlugin(...
    'IncludingAssumptionFailures', true))
```

If you run the test runner, you enter debug mode for both `testOne` and `testTwo`.

See Also

`dbcont` | `dbquit` | `dbstep` | `dbup` | `matlab.unittest.plugins`

matlab.unittest.plugins.TAPPlugin class

Package: matlab.unittest.plugins

Plugin that produces Test Anything Protocol stream

Description

The `TAPPlugin` creates a plugin that produces a Test Anything Protocol (TAP) stream. Using this plugin, you can integrate MATLAB Unit Test results into third-party systems that recognize the TAP protocol. For example, you can integrate test results with continuous integration systems like Jenkins™ or TeamCity®.

Construction

`matlab.unittest.plugins.TAPPlugin.producingOriginalFormat` creates a plugin that produces output in the form of the original TAP format (version 12). By default, the plugin displays the output to the screen. In this case, other output sent to the screen can invalidate the TAP stream. To avoid this, redirect the output to a different output stream, such as the `ToFile` stream.

`matlab.unittest.plugins.TAPPlugin.producingOriginalFormat(stream)` redirects all the text output to the output stream, `stream`. If you do not specify the output stream, the plugin uses the `ToStandardOutput` stream.

Input Arguments

`stream`

Location where the plugin directs text output, specified as an `OutputStream`.

Default: `ToStandardOutput`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create TAP Plugin

In an new file in your working folder, create `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4')
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5')
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.TAPPlugin
import matlab.unittest.plugins.ToFile

suite   = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window using the default plugin.

```
runner = TestRunner.withTextOutput;
```

Create a `TAPPlugin` that explicitly specifies that its output should go to the file `MyTapOutput.tap`.

```
tapFile = 'MyTAPOutput.tap';
plugin = TAPPlugin.producingOriginalFormat(ToFile(tapFile));
```

Add the plugin to the TestRunner and run the suite.

```
runner.addPlugin(plugin)
```

```
result = runner.run(suite);

Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        5         4           1        0.25

    Actual Value:
             5
    Expected Value:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
...
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Observe contents in the file created by the plugin.

```
disp(fileread(tapFile))
```

```
1..3
not ok 1 - ExampleTest/testOne
ok 2 - ExampleTest/testTwo
ok 3 - ExampleTest/testThree
```

You can use the `TAPPlugin` directed to standard output. However, any other text displayed to standard output (such as failed test information) interrupts the stream and has the potential to invalidate it.

See Also

`matlab.unittest.plugins.TestRunnerPlugin` | `matlab.unittest.plugins.ToFile` | `matlab.unittest.plugins.OutputStream`

External Web Sites

- Jenkins
- TeamCity
- testanything.org

Introduced in R2014a

matlab.unittest.plugins.ToFile class

Package: matlab.unittest.plugins
Superclasses: matlab.unittest.plugins.OutputStream

Output stream to write text output to file

Description

The `ToFile` class creates an output stream that writes text output to a file. Whenever text prints to this stream, the output stream opens the file, appends the text, and closes the file.

Construction

`matlab.unittest.plugins.ToFile(fname)` creates an `OutputStream` that writes text output to the file, `fname`.

Input Arguments

`fname`

Name of file to write the output text, specified as a string. If `fname` exists, the text from the stream is appended to the file.

Properties

`Filename`

Name of file to redirect text output from the plugin, specified in the input argument, `fname`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Direct TAPPlugin Output Text to Separate File

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4')
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5')
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure.
The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.TAPPlugin
import matlab.unittest.plugins.ToFile

suite   = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window.

```
runner = TestRunner.withTextOutput;
```

Create a `TAPPlugin` that explicitly specifies that its output should go to the file, `MyTapOutput.tap`.

```
filename = 'MyTapOutput.tap';
plugin = TAPPlugin.producingOriginalFormat(ToFile(filename));
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        5         4           1        0.25

    Actual Value:
             5
    Expected Value:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
...
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Only the test failures produce output to the screen. By default, `TestRunner.withTextOutput` uses a `FailureDiagnosticsPlugin` to display output on the screen.

Observe contents in the file created by the plugin.

```
disp(fileread(filename))
```

```
1..3
not ok 1 - ExampleTest/testOne
ok 2 - ExampleTest/testTwo
ok 3 - ExampleTest/testThree
```

See Also

`matlab.unittest.plugins.OutputStream` | `fopen` | `fprintf` | `matlab.unittest.plugins`

Introduced in R2014a

matlab.unittest.plugins.ToStandardOutput class

Package: matlab.unittest.plugins
Superclasses: matlab.unittest.plugins.OutputStream

Output stream to display text information to screen

Description

The `ToStandardOutput` class creates an output stream to display text output to the screen. Many plugins that accept an output stream use `ToStandardOutput` as their default stream.

Construction

`matlab.unittest.plugins.ToStandardOutput` creates an `OutputStream` that prints text output to the screen.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Direct Plugin Output Text to Standard Output

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4')
        end
```

```
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5')
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure. The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.FailureDiagnosticsPlugin
import matlab.unittest.plugins.ToStandardOutput

suite   = TestSuite.fromClass(?ExampleTest);
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins.

```
runner = TestRunner.withNoPlugins;
```

Create a `FailureDiagnosticsPlugin` that explicitly specifies that its output should go to the screen.

```
plugin = FailureDiagnosticsPlugin(ToStandardOutput);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4
```

```
    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                1        5         4           1        0.25

    Actual Value:
             5
    Expected Value:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                Failed by verification.
```

Only the test failures produce output to the screen.

See Also

`matlab.unittest.plugins.OutputStream` | `fprintf` | `matlab.unittest.plugins`

Introduced in R2014a

matlab.unittest.plugins.TestRunnerPlugin class

Package: matlab.unittest.plugins

Plugin interface for extending TestRunner

Description

The `TestRunnerPlugin` interface enables extension of the `matlab.unittest.TestRunner`. To customize a test run, create a subclass of `TestRunnerPlugin` and override select methods. `TestRunnerPlugin` provides you with a default implementation, so override only methods necessary to achieve your required customization. Every method you implement must invoke its corresponding superclass method, passing along the same instance of `pluginData` that it receives.

To run tests with this extension, add the custom `TestRunnerPlugin` to the `TestRunner` using the `addPlugin` method of `TestRunner`.

Methods

runTestSuite	Extend running of TestSuite array
createSharedTestFixture	Extend creation of shared test fixture instances
setupSharedTestFixture	Extend setting up shared test fixture
runTestClass	Extend running of TestSuite array from same class or function
createTestClassInstance	Extend creation of class-level TestCase instances
setupTestClass	Extend setting up test class

runTest	Extend running of single TestSuite element
createTestMethodInstance	Extend creation of method-level TestCase instances
setupTestMethod	Extend setting up of test method
runTestMethod	Extend running of single Test method
teardownTestMethod	Extend tearing down of test method
teardownTestClass	Extend tearing down of test class
teardownSharedTestFixture	Extend tearing down shared test fixture

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata`

Related Examples

- "Write Plugins to Extend TestRunner"
- "Create Custom Plugin"

Introduced in R2014a

runTestSuite

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend running of TestSuite array

Syntax

```
runTestSuite(plugin,pluginData)
```

Description

`runTestSuite(plugin,pluginData)` extends the running of the original `TestSuite` array that the test framework hands to the `TestRunner`.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test suite information, specified as an instance of `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement runTestSuite Method

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTestSuite(plugin, pluginData)
```

```
            % Introspect into pluginData to get TestSuite size
            suiteSize = numel(pluginData.TestSuite);
            fprintf('### Running a total of %d tests\n', suiteSize)

            % Invoke the super class method
            runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData)
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` |
`matlab.unittest.plugins.plugindata.TestSuiteRunPluginData` |
`matlab.unittest.TestResult`

Introduced in R2014a

createSharedTestFixture

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend creation of shared test fixture instances

Syntax

```
f = createSharedTestFixture(plugin,pluginData)
```

Description

`f = createSharedTestFixture(plugin,pluginData)` extends the creation of shared test fixtures and returns the modified `Fixture` instance, `f`. The testing framework uses the fixture instance to customize running tests that use shared fixtures. The testing framework evaluates this method within the scope of the `runTestSuite` method of the `TestRunnerPlugin` for each shared test fixture it needs to set up. A typical implementation of this method is to add listeners to various events originating from the shared test fixture instance. Since the `Fixture` inherits from the `handle` class, add listeners by calling the `addlistener` method from within the `createSharedTestFixture` method.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Shared test fixture creation information, specified as an instance of `matlab.unittest.plugins.plugindata.PluginData`. The test framework uses this information to introspect into the test content.

Examples

Extend Creation of Shared Test Fixture Instances

Extend the running of tests to count the number of shared test fixture assertion failures.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    properties (SetAccess = private)
        FixtureAssertionFailureData = {};
    end

    methods (Access = protected)
        function fixture = createSharedTestFixture(plugin, pluginData)
            % Invoke the super class method
            fixture = createSharedTestFixture@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

            % Get the fixture name
            fixtureName = pluginData.Name;

            % Add a listener to fixture assertion failures
            % and capture the qualification failure information
            fixture.addlistener('AssertionFailed', @(~,evd) ...
                plugin.captureFixtureAssertionFailureData(evd, fixtureName))
        end
    end

    methods (Access = private)
        function captureFixtureAssertionFailureData(plugin, eventData, fixtureName)
            plugin.FixtureAssertionFailureData{end+1} = struct(...
                'FixtureName', fixtureName, ...
                'ActualValue', eventData.ActualValue, ...
                'Constraint' , eventData.Constraint, ...
                'Stack'      , eventData.Stack);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.fixtures.Fixture` | `matlab.unittest.plugins.plugindata.PluginData` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.qualifications.QualificationEventData`

Introduced in R2014a

setupSharedTestFixture

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend setting up shared test fixture

Syntax

```
setupSharedTestFixture(plugin,pluginData)
```

Description

`setupSharedTestFixture(plugin,pluginData)` extends the setting up of a shared test fixture. This method defines how the `TestRunner` performs shared fixture setup. The test framework evaluates this method one time for each shared test fixture, within the scope of the `runTestSuite` method of the `TestRunnerPlugin`.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Shared test fixture setup information, specified as an instance of `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement setupSharedTestFixture Method

Display the shared test fixture name at setup time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function setupSharedTestFixture(plugin, pluginData)
            fprintf('### Setting up: %s\n', pluginData.Name)
            setupSharedTestFixture@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData` | `createSharedTestFixture`

Introduced in R2014a

runTestClass

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend running of TestSuite array from same class or function

Syntax

```
runTestClass(plugin,pluginData)
```

Description

`runTestClass(plugin,pluginData)` extends the running of tests that belong to the same test class, function, or script. This method applies to a subset of the full `TestSuite` being run by the `TestRunner`. The test framework evaluates this method within the scope of the `runTestSuite` method of the `TestRunnerPlugin`. It evaluates this method between setting up and tearing down the shared test fixture (`setupSharedTestFixture` and `teardownSharedTestFixture`). Provided the test framework completes shared test fixture setup, it invokes this method one time per test class.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test suite information, specified as an instance of `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData`. The test framework uses this information to introspect into the test content.

Examples

Extend runTestClass method

Print the label of the test content element at run time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTestClass(plugin, pluginData)
            fprintf('### Running test class: %s\n', pluginData.Name)

            runTestClass@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData` | `matlab.unittest.TestResult`

Introduced in R2014a

createTestClassInstance

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend creation of class-level TestCase instances

Syntax

```
tc = createTestClassInstance(plugin,pluginData)
```

Description

`tc = createTestClassInstance(plugin,pluginData)` extends the creation of class-level `TestCase` instances, and returns the modified `TestCase` instance, `tc`. The test framework uses the `TestCase` instance to customize running tests that belong to the same test class. The test framework evaluates this method within the scope of the `runTestClass` method of the `TestRunnerPlugin`. A typical implementation of this method is to add listeners to various events originating from the class level instance. Since the `TestCase` inherits from the `handle` class, add listeners by calling the `addlistener` method from within the `createTestClassInstance` method. For each class, the test framework passes the instance to any method with the `TestClassSetup` or `TestClassTeardown` attribute.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Class-level `TestCase` creation information, specified as an instance of `matlab.unittest.plugins.plugindata.PluginData`. The test framework uses this information to introspect into the test content.

Examples

Extend Creation of Class-Level TestCase Instances

Extend the running of tests to count the number of class-level assumption failures.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    properties (SetAccess = private)
        TestClassAssumptionFailureData = {};
    end

    methods (Access = protected)
        function testCase = createTestClassInstance(plugin,pluginData)
            % Invoke super class method
            testCase = createTestClassInstance@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);

            % Get the test class name
            instanceName = pluginData.Name;

            % Add a listener to capture assumption failures
            testCase.addlistener('AssumptionFailed', @(~,evd) ...
                plugin.captureClassLevelAssumptionFailureData(evd,instanceName))
        end
    end

    methods (Access = private)
        function captureClassLevelAssumptionFailureData(plugin,eventData,instanceName)
            plugin.TestClassAssumptionFailureData{end+1} = struct(...
                'InstanceName', instanceName, ...
                'ActualValue'   , eventData.ActualValue, ...
                'Constraint'    , eventData.Constraint, ...
                'Stack'         , eventData.Stack);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.TestCase` | `matlab.unittest.plugins.plugindata.PluginData` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.qualifications.QualificationEventData`

Introduced in R2014a

setupTestClass

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend setting up test class

Syntax

```
setupTestClass(plugin,pluginData)
```

Description

`setupTestClass(plugin,pluginData)` extends the setting up of a test class. This method defines how the `TestRunner` performs test class setup. The test framework evaluates this method within the scope of the `runTestClass` method of the `TestRunnerPlugin`. If the test class contains properties with the `ClassSetupParameter` attribute, the test framework evaluates the `setupTestClass` method as many times as the class setup parameterization dictates.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test class setup information, specified as an instance of `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement setupTestClass Method

Display the test class name at setup time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function setupTestClass(plugin, pluginData)
            fprintf('### Setting up: %s\n', pluginData.Name)
            setupTestClass@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

- "Write Plugins to Extend TestRunner"
- "Create Custom Plugin"

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` | `createTestClassInstance`

Introduced in R2014a

runTest

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend running of single TestSuite element

Syntax

```
runTest(plugin,pluginData)
```

Description

`runTest(plugin,pluginData)` extends the running of a single `TestSuite` element. This method allows the test author to override the method that runs a scalar test element in the `TestSuite` array, including the creation of the `TestCase`, and the `TestMethodSetup` and `TestMethodTeardown` routines. Provided the test framework completes all fixture setup, it invokes this method one time per test element.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test element information, specified as an instance of `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData`. The test framework uses this information to introspect into the test content.

Examples

Extend runTest method

Print the label of the test content element at run time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTest(plugin, pluginData)
            fprintf('### Running test: %s\n', pluginData.Name)
            runTest@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData` | `matlab.unittest.TestResult`

Introduced in R2014a

createTestMethodInstance

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend creation of method-level TestCase instances

Syntax

```
tc = createTestMethodInstance(plugin,pluginData)
```

Description

`tc = createTestMethodInstance(plugin,pluginData)` extends the creation of method-level `TestCase` instances, and returns the modified `TestCase` instance, `tc`. The test framework evaluates this method within the scope of the `runTest` method of the `TestRunnerPlugin`. A typical implementation of this method is to add listeners to various events originating from the method level instance. Since the `TestCase` inherits from the `handle` class, add listeners by calling the `addlistener` method from within the `createTestMethodInstance` method. The test framework creates instances for every element of the `matlab.unittest.Test` array and passes each instance to its corresponding `Test` methods and to any method with the `TestMethodSetup` or `TestMethodTeardown` attribute.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Method-level `TestCase` creation information, specified as an instance of `matlab.unittest.plugins.plugindata.PluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement createMethodInstance Method

Add a listener to listen for assumption failures. Use the helper function, `captureMethodLevelAssumptionFailureData`, to populate the `TestMethodAssumptionFailureData` property.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    properties (SetAccess = private)
        TestMethodAssumptionFailureData = {};
    end

    methods (Access = protected)
        function testCase = createTestMethodInstance(plugin, pluginData)
            testCase = createTestMethodInstance@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

            instanceName = pluginData.Name;
            testCase.addlistener('AssumptionFailed', @(~,evd) ...
                plugin.captureMethodLevelAssumptionFailureData(evd,instanceName))
        end
    end

    methods (Access = private)
        function captureMethodLevelAssumptionFailureData(...
                plugin, eventData, instanceName)
            plugin.TestMethodAssumptionFailureData{end+1} = struct(...
                'InstanceName', instanceName, ...
                'ActualValue'     , eventData.ActualValue, ...
                'Constraint'      , eventData.Constraint, ...
                'Stack'           , eventData.Stack);
        end
    end

end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.TestCase` | `matlab.unittest.plugins.plugindata.PluginData` | `matlab.unittest.plugins.TestRunnerPlugin.createTestClassInstance` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.qualifications.QualificationEventData`

Introduced in R2014a

setupTestMethod

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend setting up of test method

Syntax

```
setupTestMethod(plugin,pluginData)
```

Description

`setupTestMethod(plugin,pluginData)` extends the setting up of a test method. This method defines how the `TestRunner` performs test method setup for the single test suite element. The test framework evaluates this method within the scope of the `runTest` method of the `TestRunnerPlugin`.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test method setup information, specified as an instance of `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement setupTestMethod

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
```

```
    methods (Access = protected)
        function setupTestMethod(plugin, pluginData)
            fprintf('### Setting up: %s\n', pluginData.Name)
            setupTestMethod@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`

Introduced in R2014a

runTestMethod

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend running of single Test method

Syntax

```
runTestMethod(plugin,pluginData)
```

Description

`runTestMethod(plugin,pluginData)` extends the running of a single `Test` method. The test framework evaluates this method within the scope of the `runTest` method of the `TestRunnerPlugin`. It evaluates this method between setting up and tearing down the scalar `TestSuite` element (`setupTestMethod` and `teardownTestMethod`).

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test method information, specified as an instance of `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData`. The test framework uses this information to introspect into the test content.

Examples

Extend runTestMethod method

Print the time taken to evaluate the test method.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTestMethod(plugin, pluginData)
            tic

            runTestMethod@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);

            fprintf('### %s ran in %f seconds excluding fixture time.',...
                pluginData.Name, toc)
        end
    end
end
```

- "Write Plugins to Extend TestRunner"
- "Create Custom Plugin"

See Also

`matlab.unittest.TestRunner` |
`matlab.unittest.plugins.plugindata.TestSuiteRunPluginData` |
`matlab.unittest.TestResult` |
`matlab.unittest.plugins.TestRunnerPlugin.runTest` |
`matlab.unittest.plugins.TestRunnerPlugin.runTestClass`

Introduced in R2014a

teardownTestMethod

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend tearing down of test method

Syntax

```
teardownTestMethod(plugin,pluginData)
```

Description

`teardownTestMethod(plugin,pluginData)` extends the tearing down of a test method. This method defines how the `TestRunner` performs test method teardown for the single test suite element. The test framework evaluates this method within the scope of the `runTest` method of the `TestRunnerPlugin`.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test method teardown information, specified as an instance of `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement teardownTestMethod Method

Display the test method name at teardown time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function teardownTestMethod(plugin, pluginData)
            fprintf('### Tearing down: %s\n', pluginData.Name)
            teardownTestMethod@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` |
`matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` |
`matlab.unittest.plugins.TestRunnerPlugin.setupTestMethod`

Introduced in R2014a

teardownTestClass

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend tearing down of test class

Syntax

```
teardownTestClass(plugin,pluginData)
```

Description

`teardownTestClass(plugin,pluginData)` extends the tearing down of a test class. This method defines how the `TestRunner` performs test class teardown. The test framework evaluates this method within the scope of the `runTestClass` method of the `TestRunnerPlugin`. If the test class contains properties with the `ClassSetupParameter` attribute, the test framework evaluates the `teardownTestClass` method as many times as the class setup parameterization dictates.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Test class teardown information, specified as an instance of `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement teardownTestClass Method

Display the test class name at teardown time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function teardownTestClass(plugin, pluginData)
            fprintf('### Tearing down: %s\n', pluginData.Name)
            teardownTestClass@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` | `createTestClassInstance`

Introduced in R2014a

teardownSharedTestFixture

Class: matlab.unittest.plugins.TestRunnerPlugin
Package: matlab.unittest.plugins

Extend tearing down shared test fixture

Syntax

```
teardownSharedTestFixture(plugin,pluginData)
```

Description

`teardownSharedTestFixture(plugin,pluginData)` extends the tearing down of a shared test fixture. This method defines how the `TestRunner` performs shared fixture teardown. The test framework evaluates this method one time for each shared test fixture, within the scope of the `runTestSuite` method of the `TestRunnerPlugin`.

Input Arguments

`plugin`

Instance of `matlab.unittest.plugins.TestRunnerPlugin`.

`pluginData`

Shared test fixture teardown information, specified as an instance of `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData`. The test framework uses this information to introspect into the test content.

Examples

Implement teardownSharedTestFixture Method

Display the shared test fixture name at teardown time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function teardownSharedTestFixture(plugin, pluginData)
            fprintf('### Setting up: %s\n', pluginData.Name)
            teardownSharedTestFixturee@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

- “Write Plugins to Extend TestRunner”
- “Create Custom Plugin”

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData` | `createSharedTestFixture`

Introduced in R2014a

matlab.unittest.plugins.TestRunProgressPlugin class

Package: matlab.unittest.plugins

Plugin that reports test run progress

Description

The `TestRunProgressPlugin` creates a plugin that reports on test run progress.

Construction

`matlab.unittest.plugins.TestRunProgressPlugin.withVerbosity(v)` constructs a `TestRunProgressPlugin` for the specified verbosity.

`matlab.unittest.plugins.TestRunProgressPlugin.withVerbosity(v, stream)` redirects the text output to the output stream.

Input Arguments

v — Verbosity level

`1 | 2 | 3 | 4 | matlab.unittest.Verbosity` enumeration

Verbosity level, specified as an integer value between 1 and 4 or a `matlab.unittest.Verbosity` enumeration object. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Corresponding Enumeration Object	Verbosity Description
1	`matlab.unittest.Verbosity.Ter`	Minimal amount of information
2	`matlab.unittest.Verbosity.Con`	Typical amount of information
3	`matlab.unittest.Verbosity.Det`	Supplemental amount of information
4	`matlab.unittest.Verbosity.Ver`	Surplus of information

`stream` — Location where plugin directs text output

`ToStandardOutput` instance (default) | `OutputStream` instance

Location where the plugin directs text output, specified as an `OutputStream` instance. By default, the plugin uses the `OutputStream` subclass `ToStandardOutput` as the stream.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Test Run Progress Plugin

Create a function-based test called `cylinderPlotTest` in a file in your working folder.

```
function tests = cylinderPlotTest
tests = functiontests(localfunctions);
end

function setupOnce(testCase)
testCase.TestData.Figure = figure;
addTeardown(testCase,@close,testCase.TestData.Figure)
end

function setup(testCase)
testCase.TestData.Axes = axes('Parent',testCase.TestData.Figure);
addTeardown(testCase,@clf,testCase.TestData.Figure)
cylinder(testCase.TestData.Axes,10)
end

function testXLim(testCase)
xlim = testCase.TestData.Axes.XLim;
verifyLessThanOrEqual(testCase,xlim(1),-10,'Minimum x-limit too large')
verifyGreaterThanOrEqual(testCase,xlim(2),10,'Maximum x-limit too small')
end
```

```
function zdataTest(testCase)
s = findobj(testCase.TestData.Axes,'Type','surface');
verifyEqual(testCase,min(s.ZData(:)),0,'Min cylinder value is incorrect')
verifyEqual(testCase,max(s.ZData(:)),1,'Max cylinder value is incorrect')
end
```

At the command prompt, run the test.

```
results = run(cylinderPlotTest);
```

```
Running cylinderPlotTest
..
Done cylinderPlotTest
__________
```

By default, the test runner uses verbosity level 2.

Create a test runner to report the diagnostics at level 1, and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.TestRunProgressPlugin

runner = TestRunner.withNoPlugins;
p = TestRunProgressPlugin.withVerbosity(1);
runner.addPlugin(p);

results = runner.run(cylinderPlotTest);
```

```
..
```

Create a test runner to report the diagnostics at level 4, and rerun the test.

```
runner = TestRunner.withNoPlugins;
p = TestRunProgressPlugin.withVerbosity(4);
runner.addPlugin(p);

results = runner.run(cylinderPlotTest);
```

```
 Running cylinderPlotTest
  Setting up cylinderPlotTest
    Evaluating TestClassSetup: setupOnce
  Done setting up cylinderPlotTest in 0.067649 seconds
   Running cylinderPlotTest/testXLim
    Evaluating TestMethodSetup: setup
    Evaluating Test: testXLim
    Evaluating TestMethodTeardown: teardown
```

```
    Evaluating addTeardown function: clf
   Done cylinderPlotTest/testXLim in 0.053834 seconds
   Running cylinderPlotTest/zdataTest
    Evaluating TestMethodSetup: setup
    Evaluating Test: zdataTest
    Evaluating TestMethodTeardown: teardown
    Evaluating addTeardown function: clf
   Done cylinderPlotTest/zdataTest in 0.037715 seconds
  Tearing down cylinderPlotTest
    Evaluating TestClassTeardown: teardownOnce
    Evaluating addTeardown function: close
  Done tearing down cylinderPlotTest in 0.022783 seconds
 Done cylinderPlotTest in 0.18198 seconds
```

Configure Progress Message Output

Create a class named `ExampleProgressTest` in a file in your current working folder.

```
classdef ExampleProgressTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5,4)
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5,5)
        end
    end
end
```

At the command prompt, create the test suite and a runner at verbosity level 3, and then run the test.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.TestRunProgressPlugin

suite = TestSuite.fromClass(?ExampleProgressTest);

runner = TestRunner.withNoPlugins;
p = TestRunProgressPlugin.withVerbosity(3);
runner.addPlugin(p);
results = runner.run(suite);
```

```
 Running ExampleProgressTest
```

```
  Setting up ExampleProgressTest
  Done setting up ExampleProgressTest in 0 seconds
   Running ExampleProgressTest/testOne
   Done ExampleProgressTest/testOne in 0.0049988 seconds
   Running ExampleProgressTest/testTwo
   Done ExampleProgressTest/testTwo in 0.0044541 seconds
  Tearing down ExampleProgressTest
  Done tearing down ExampleProgressTest in 0 seconds
 Done ExampleProgressTest in 0.0094529 seconds
__________
```

Create a new plugin to direct the output to a file named `myOutput.log`, and rerun the tests.

```
import matlab.unittest.plugins.ToFile
outFile = 'myOutput.log';

runner = TestRunner.withNoPlugins;
p = TestRunProgressPlugin.withVerbosity(3,ToFile(outFile));
runner.addPlugin(p);

results = runner.run(suite);
```

Observe the contents of the file created by the plugin.

```
disp(fileread(outFile))
```

```
 Running ExampleProgressTest
  Setting up ExampleProgressTest
  Done setting up ExampleProgressTest in 0 seconds
   Running ExampleProgressTest/testOne
   Done ExampleProgressTest/testOne in 0.0050172 seconds
   Running ExampleProgressTest/testTwo
   Done ExampleProgressTest/testTwo in 0.0049449 seconds
  Tearing down ExampleProgressTest
  Done tearing down ExampleProgressTest in 0 seconds
 Done ExampleProgressTest in 0.009962 seconds
__________
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.TestRunnerPlugin` | `matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.ToStandardOutput` | `matlab.unittest.Verbosity`

Introduced in R2014b

matlab.unittest.plugins.plugindata

Summary of classes in MATLAB Plugin Data Interface

Description

The `plugindata` classes store information about test content for use by the plugins and plugin methods. The `TestRunner` passes instances of these classes to various plugin methods. The `matlab.unittest.plugins.plugindata` package consists of the following MATLAB plugin data classes.

matlab.unittest.plugins.plugindata.ImplicitFixturePluginData	Plugin data containing test setup and teardown information
matlab.unittest.plugins.plugindata.PluginData	Data object passed to TestRunnerPlugin methods
matlab.unittest.plugins.plugindata.SharedTestFixturePluginData	Plugin data containing shared test fixture information
matlab.unittest.plugins.plugindata.TestSuiteRunPluginData	Plugin data containing selected test information

See Also

`matlab.unittest.plugins` | `TestRunnerPlugin`

matlab.unittest.plugins.plugindata.ImplicitFixturePluginData class

Package: matlab.unittest.plugins.plugindata

Plugin data containing test setup and teardown information

Description

The `ImplicitFixturePluginData` class defines the data the `TestRunner` passes to plugin methods related to setting up and tearing down tests. The `TestRunner` creates this class, so there is no need for test plugin authors to construct this class directly.

Properties

`Name` — Label of test content test runner sets up or tears down

string

Label of test content test runner sets up or tears down, represented as a string. Use the `Name` property for informational, labeling, and display purposes. Do not use `Name` programmatically to introspect into the content.

Data Types: `char`

See Also

`TestRunnerPlugin` | `TestRunnerPlugin.setupTestClass` | `TestRunnerPlugin.setupTestMethod` | `TestRunnerPlugin.teardownTestClass` | `TestRunnerPlugin.teardownTestMethod`

Introduced in R2015a

matlab.unittest.plugins.plugindata.PluginData class

Package: matlab.unittest.plugins.plugindata

Data object passed to TestRunnerPlugin methods

Description

The `PluginData` class defines the data the `TestRunner` passes to various plugin methods. It is created by the `TestRunner`, so there is no need for test plugin authors to construct this class directly.

Properties

`Name`

Label of test content executed by the test runner within the scope of a plugin method, represented as a string. Use the `Name` property for informational, labeling, and display purposes. Do not use `Name`programmatically to introspect into the content.

See Also

`TestRunnerPlugin`

Introduced in R2014a

matlab.unittest.plugins.plugindata.SharedTestFixturePluginDa class

Package: matlab.unittest.plugins.plugindata

Plugin data containing shared test fixture information

Description

The `SharedTestFixturePluginData` defines the data the `TestRunner` passes to plugin methods related to shared test fixtures. The `TestRunner` creates this, so there is no need for test plugin authors to construct this class directly.

Properties

`Name`

Label of shared test fixture, represented as a string. Use the `Name` property for informational, labeling, and display purposes. Do not use `Name`programmatically to introspect into the content.

`Description`

Description of action performed during setup and teardown of a shared text fixture, represented as a string

See Also

`matlab.unittest.fixtures.Fixture` | `TestRunnerPlugin`

Introduced in R2014a

matlab.unittest.plugins.plugindata.TestSuiteRunPluginData class

Package: matlab.unittest.plugins.plugindata

Plugin data containing selected test information

Description

The `TestSuiteRunPluginData` defines the data the `TestRunner` passes to plugin methods related to running tests from the suite. The `TestRunner` creates this, so there is no need for test plugin authors to construct this class directly.

Properties

`Name`

Name corresponding to the portion of the test suite the runner executes within a plugin method, represented as a string. Use the `Name` property for informational, labeling, and display purposes. Do not use `Name`programmatically to introspect into the content.

`TestSuite`

Select test methods, represented as a `matlab.unittest.TestSuite` instance

`TestResult`

Results from running select test methods listed in `TestSuite`, represented as a `matlab.unittest.TestResult` array

See Also

`matlab.unittest.TestSuite` | `matlab.unittest.TestResult` | `TestRunnerPlugin`

Introduced in R2014a

plus, +

Addition

Syntax

```
C = A + B
C = plus(A,B)
```

Description

`C = A + B` adds arrays `A` and `B` and returns the result in `C`.

`C = plus(A,B)` is an alternate way to execute `A + B`, but is rarely used. It enables operator overloading for classes.

Examples

Add Scalar to Array

Create an array, `A`, and add a scalar value to it.

```
A = [0 1; 1 0];
C = A + 2
```

```
C =

     2     3
     3     2
```

The scalar value is added to each entry of `A`.

Add Two Arrays

Create two arrays, `A` and `B`, and add them together.

```
A = [1 0; 2 4];
B = [5 9; 2 1];
```

```
C = A + B

C =

     6     9
     4     5
```

The elements of A are added to the corresponding elements of B.

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. A can be a numeric array, logical array, character array, datetime array, duration array, or calendar duration array. Inputs A and B must be the same size unless one is a scalar. You can add a scalar value to any other value.

If one input is a datetime array, duration array, or calendar duration array, then numeric values in the other input are treated as a number of 24-hour days.
Complex Number Support: Yes

B — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. B can be a numeric array, logical array, character array, datetime array, duration array, or calendar duration array. Inputs A and B must be the same size unless one is a scalar. You can add a scalar value to any other value.

If one input is a datetime array, duration array, or calendar duration array, then numeric values in the other input are treated as a number of 24-hour days.
Complex Number Support: Yes

More About

- “Array vs. Matrix Operations”
- “Operator Precedence”

See Also

`cumsum` | `minus` | `sum` | `uplus`

pointLocation

Class: DelaunayTri

(Will be removed) Simplex containing specified location

Note: `pointLocation(DelaunayTri)` will be removed in a future release. Use `pointLocation(triangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
SI = pointLocation(DT,QX)
SI = pointLocation(DT,QX,QY)
SI = pointLocation(DT,QX,QY,QZ)
[SI, BC] = pointLocation(DT,...)
```

Description

`SI = pointLocation(DT,QX)` returns the indices `SI` of the enclosing simplex (triangle/tetrahedron) for each query point location in `QX`. The enclosing simplex for point `QX(k,:)` is `SI(k)`. `pointLocation` returns `NaN` for all points outside the convex hull.

`SI = pointLocation(DT,QX,QY)` and `SI = pointLocation(DT,QX,QY,QZ)` allow the query point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`[SI, BC] = pointLocation(DT,...)` returns the barycentric coordinates `BC`.

Input Arguments

`DT`	Delaunay triangulation.

`QX`	Matrix of size `mpts`-by-`ndim`, `mpts` being the number of query points.

Output Arguments

`SI`	Column vector of length `mpts` containing the indices of the enclosing simplex for each query point. `mpts` is the number of query points.
`BC`	`BC` is a `mpts`-by-`ndim` matrix, each row `BC(i,:)` represents the barycentric coordinates of `QX(i,:)` with respect to the enclosing simplex `SI(i)`.

Examples

Example 1

Create a 2-D Delaunay triangulation:

```
X = rand(10,2);
dt = DelaunayTri(X);
```

Find the triangles that contain specified query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
triids = pointLocation(dt, qrypts)
```

Example 2

Create a 3-D Delaunay triangulation:

```
x = rand(10,1);
y = rand(10,1);
z = rand(10,1);
dt = DelaunayTri(x,y,z);
```

Find the triangles that contain specified query points and evaluate the barycentric coordinates:

```
qrypts = [0.25 0.25 0.25; 0.5 0.5 0.5];
```

```
[tetids, bcs] = pointLocation(dt, qrypts)
```

See Also

delaunayTriangulation | nearestNeighbor | triangulation

pol2cart

Transform polar or cylindrical coordinates to Cartesian

Syntax

```
[X,Y] = pol2cart(THETA,RHO)
[X,Y,Z] = pol2cart(THETA,RHO,Z)
```

Description

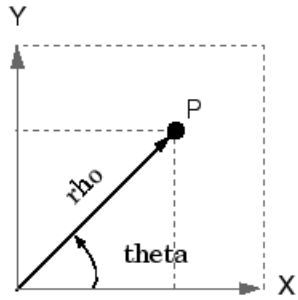
`[X,Y] = pol2cart(THETA,RHO)` transforms the polar coordinate data stored in corresponding elements of `THETA` and `RHO` to two-dimensional Cartesian, or *xy*, coordinates. The arrays `THETA` and `RHO` must be the same size (or either can be scalar). The values in `THETA` must be in radians.

`[X,Y,Z] = pol2cart(THETA,RHO,Z)` transforms the cylindrical coordinate data stored in corresponding elements of `THETA`, `RHO`, and `Z` to three-dimensional Cartesian, or *xyz* coordinates. The arrays `THETA`, `RHO`, and `Z` must be the same size (or any can be scalar). The values in `THETA` must be in radians.

More About

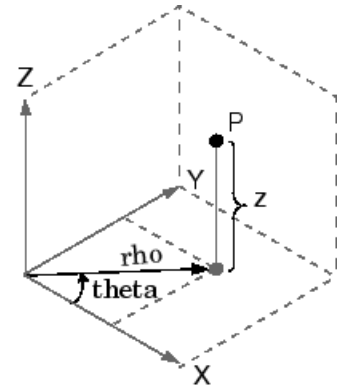
Algorithms

The mapping from polar and cylindrical coordinates to Cartesian coordinates is:



Polar to Cartesian Mapping

```
x = rho*cos(theta)
y = rho*sin(theta)
```



Polar to Cartesian Mapping

```
x = rho*cos(theta)
y = rho*sin(theta)
      z = z
```

See Also

`cart2pol` | `cart2sph` | `sph2cart`

Introduced before R2006a

polar

Polar coordinate plot

Syntax

```
polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axes_handle,...)
h = polar(...)
```

Description

The `polar` function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

`polar(theta,rho)` creates a polar coordinate plot of the angle `theta` versus the radius `rho`. `theta` is the angle from the *x*-axis to the radius vector specified in radians; `rho` is the length of the radius vector specified in dataspace units.

`polar(theta,rho,LineSpec)` `LineSpec` specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

`polar(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

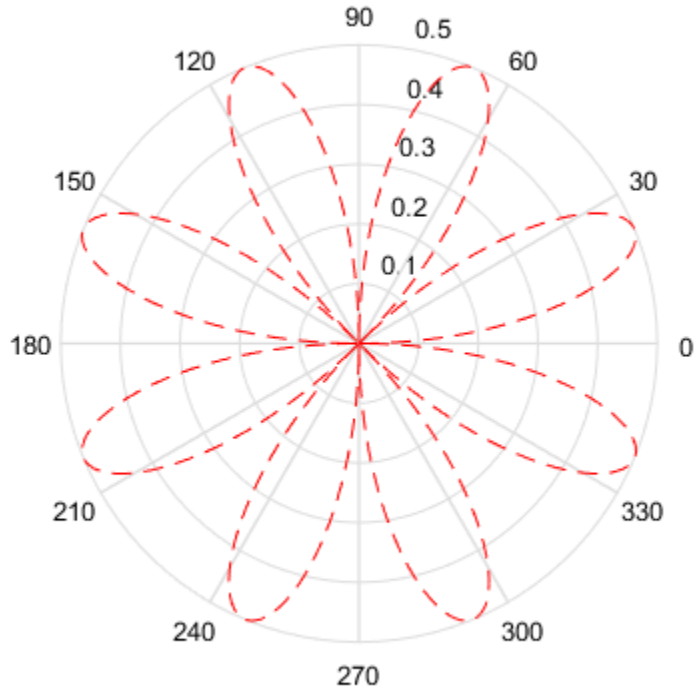
`h = polar(...)` returns the handle of a line object in `h`.

Examples

Simple Polar Plot

Create a simple polar plot using a dashed red line.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);

figure
polar(theta,rho,'--r')
```



More About

Tips

Negative `r` values reflect through the origin, rotating by `pi` (since `(theta,r)` transforms to `(r*cos(theta), r*sin(theta))`). If you want different behavior, you can

manipulate `r` prior to plotting. For example, you can make `r` equal to `max(0,r)` or `abs(r)`.

See Also

`cart2pol` | `compass` | `LineSpec` | `plot` | `pol2cart` | `rose`

Introduced before R2006a

poly

Polynomial with specified roots

Syntax

```
p = poly(A)
p = poly(r)
```

Description

`p = poly(A)` where `A` is an `n`-by-`n` matrix returns an `n+1` element row vector whose elements are the coefficients of the characteristic polynomial, $\det(\lambda I - A)$. The coefficients are ordered in descending powers: if a vector `c` has `n+1` components, the polynomial it represents is $c_1\lambda^n + c_2\lambda^{n-1} + \ldots + c_n\lambda + c_{n+1}$

`p = poly(r)` where `r` is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of `r`.

Examples

MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

```
A =

    1    2    3
    4    5    6
    7    8    0
```

is returned in a row vector by `poly`:

```
p = poly(A)

p =
    1    -6    -72    -27
```

The roots of this polynomial (eigenvalues of matrix A) are returned in a column vector by roots:

```
r = roots(p)

r =

    12.1229
    -5.7345
    -0.3884
```

More About

Tips

Note the relationship of this command to

```
r = roots(p)
```

which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector p. For vectors, roots and poly are inverse functions of each other, up to ordering, scaling, and roundoff error.

Algorithms

The algorithms employed for poly and roots illustrate an interesting aspect of the modern approach to eigenvalue computation. poly(A) generates the characteristic polynomial of A, and roots(poly(A)) finds the roots of that polynomial, which are the eigenvalues of A. But both poly and roots use eig, which is based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If A is an n-by-n matrix, poly(A) produces the coefficients c(1) through c(n+1), with c(1) = 1, in

$$\det(\lambda I - A) = c_1\lambda^n + \ldots + c_n\lambda + c_{n+1}$$

The algorithm is

```
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
```

```
for j = 1:n
    c(2:j+1) = c(2:j+1)-z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product.

$$(\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n)$$

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of `A`. This is true even if the eigenvalues of `A` are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.

See Also

`conv` | `polyval` | `roots` | `residue`

Introduced before R2006a

polyarea

Area of polygon

Syntax

```
A = polyarea(X,Y)
A = polyarea(X,Y,dim)
```

Description

`A = polyarea(X,Y)` returns the area of the polygon specified by the vertices in the vectors `X` and `Y`.

If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

If `X` and `Y` are multidimensional arrays, `polyarea` returns the area of the polygons in the first nonsingleton dimension of `X` and `Y`.

`A = polyarea(X,Y,dim)` operates along the dimension specified by scalar `dim`.

Examples

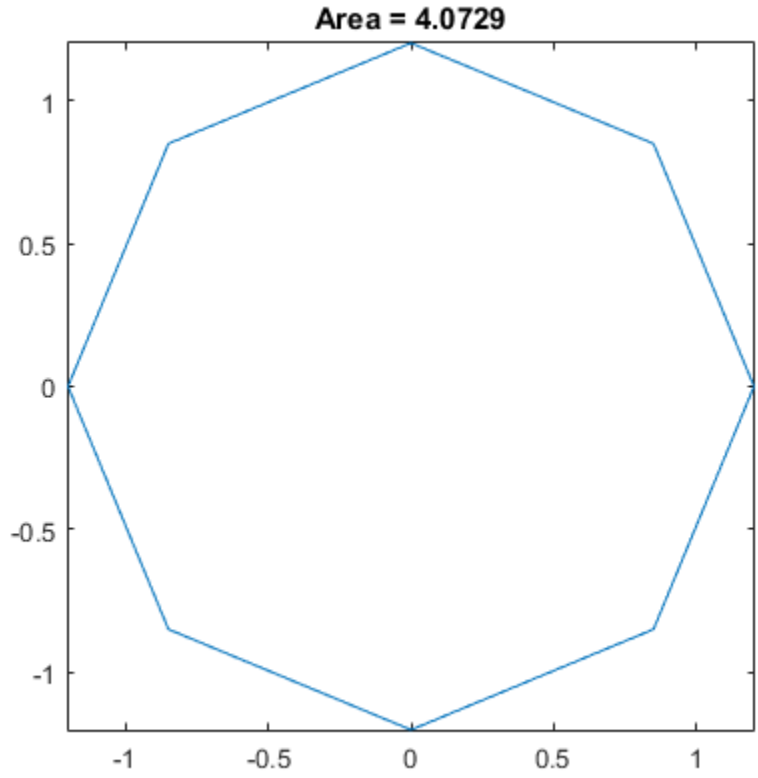
Find Area of Polygon

```
L = linspace(0,2.*pi,9);
xv = 1.2*cos(L)';
yv = 1.2*sin(L)';

xv = [xv ; xv(1)];
yv = [yv ; yv(1)];

A = polyarea(xv,yv)
```

```
A =
```

```
    4.0729
```

```
plot(xv,yv);
title(['Area = ' num2str(A)])
axis image
```



See Also

convhull | inpolygon | rectint

Introduced before R2006a

polyder

Polynomial derivative

Syntax

```
k = polyder(p)
k = polyder(a,b)
[q,d] = polyder(b,a)
```

Description

The `polyder` function calculates the derivative of polynomials, polynomial products, and polynomial quotients. The operands `a`, `b`, and `p` are vectors whose elements are the coefficients of a polynomial in descending powers.

`k = polyder(p)` returns the derivative of the polynomial `p`.

`k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`.

`[q,d] = polyder(b,a)` returns the numerator `q` and denominator `d` of the derivative of the polynomial quotient `b/a`.

Examples

The derivative of the product

$$\left(3x^2+6x+9\right)\left(x^2+2x\right)$$

is obtained with

```
a = [3 6 9];
b = [1 2 0];
k = polyder(a,b)
k =
    12    36    42    18
```

This result represents the polynomial

$$12x^3 + 36x^2 + 42x + 18$$

See Also

conv | deconv

Introduced before R2006a

polyeig

Polynomial eigenvalue problem

Syntax

```
[X,e] = polyeig(A0,A1,...Ap)
e = polyeig(A0,A1,..,Ap)
[X, e, s] = polyeig(A0,A1,..,AP)
```

Description

`[X,e] = polyeig(A0,A1,...Ap)` solves the polynomial eigenvalue problem of degree `p`

$$\left(A_0 + \lambda A_1 + \ldots + \lambda^P A_p\right)x = 0$$

where polynomial degree `p` is a non-negative integer, and `A0,A1,...Ap` are input matrices of order `n`. The output consists of a matrix `X` of size `n`-by-`n*p` whose columns are the eigenvectors, and a vector `e` of length `n*p` containing the eigenvalues.

If `lambda` is the `j`th eigenvalue in `e`, and `x` is the `j`th column of eigenvectors in `X`, then `(A0 + lambda*A1 + ... + lambda^p*Ap)*x` is approximately `0`.

`e = polyeig(A0,A1,..,Ap)` is a vector of length `n*p` whose elements are the eigenvalues of the polynomial eigenvalue problem.

`[X, e, s] = polyeig(A0,A1,..,AP)` also returns a vector `s` of length `p*n` containing condition numbers for the eigenvalues. At least one of `A0` and `AP` must be nonsingular. Large condition numbers imply that the problem is close to a problem with multiple eigenvalues.

More About

Tips

Based on the values of `p` and `n`, `polyeig` handles several special cases:

- `p = 0`, or `polyeig(A)` is the standard eigenvalue problem: `eig(A)`.
- `p = 1`, or `polyeig(A,B)` is the generalized eigenvalue problem: `eig(A,-B)`.
- `n = 1`, or `polyeig(a0,a1,...,ap)` for scalars `a0,a1,...,ap` is the standard polynomial problem: `roots([ap ... a1 a0])`.

If both `A0` and `Ap` are singular the problem is potentially ill-posed. Theoretically, the solutions might not exist or might not be unique. Computationally, the computed solutions might be inaccurate. If one, but not both, of `A0` and `Ap` is singular, the problem is well posed, but some of the eigenvalues might be zero or infinite.

Note that scaling `A0,A1,..,Ap` to have `norm(Ai)` roughly equal 1 may increase the accuracy of `polyeig`. In general, however, this cannot be achieved. (See Tisseur [3] for more detail.)

Algorithms

The `polyeig` function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of `eig` and `qz` for more on this.

References

[1] Dedieu, Jean-Pierre Dedieu and Francoise Tisseur, “Perturbation theory for homogeneous polynomial eigenvalue problems,” *Linear Algebra Appl.*, Vol. 358, pp. 71-94, 2003.

[2] Tisseur, Francoise and Karl Meerbergen, “The quadratic eigenvalue problem,” *SIAM Rev.*, Vol. 43, Number 2, pp. 235-286, 2001.

[3] Francoise Tisseur, “Backward error and condition of polynomial eigenvalue problems” *Linear Algebra Appl.*, Vol. 309, pp. 339-361, 2000.

See Also

`condeig` | `eig` | `qz`

Introduced before R2006a

polyfit

Polynomial curve fitting

Syntax

```
p = polyfit(x,y,n)
[p,S] = polyfit(x,y,n)
[p,S,mu] = polyfit(x,y,n)
```

Description

`p = polyfit(x,y,n)` returns the coefficients for a polynomial `p(x)` of degree `n` that is a best fit (in a least-squares sense) for the data in `y`. The coefficients in `p` are in descending powers, and the length of `p` is `n+1`

$$p(x) = p_1 x^n + p_2 x^{n-1} + ... + p_n x + p_{n+1}.$$

`[p,S] = polyfit(x,y,n)` also returns a structure `S` that can be used as an input to `polyval` to obtain error estimates.

`[p,S,mu] = polyfit(x,y,n)` also returns `mu`, which is a two-element vector with centering and scaling values. `mu(1)` is `mean(x)`, and `mu(2)` is `std(x)`. Using these values, `polyfit` centers `x` at zero and scales it to have unit standard deviation

$$\hat{x} = \frac{x - \bar{x}}{\sigma_x}.$$

This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

Examples

Fit Polynomial to Trigonometric Function

Generate 10 points equally spaced along a sine curve in the interval `[0,4*pi]`.

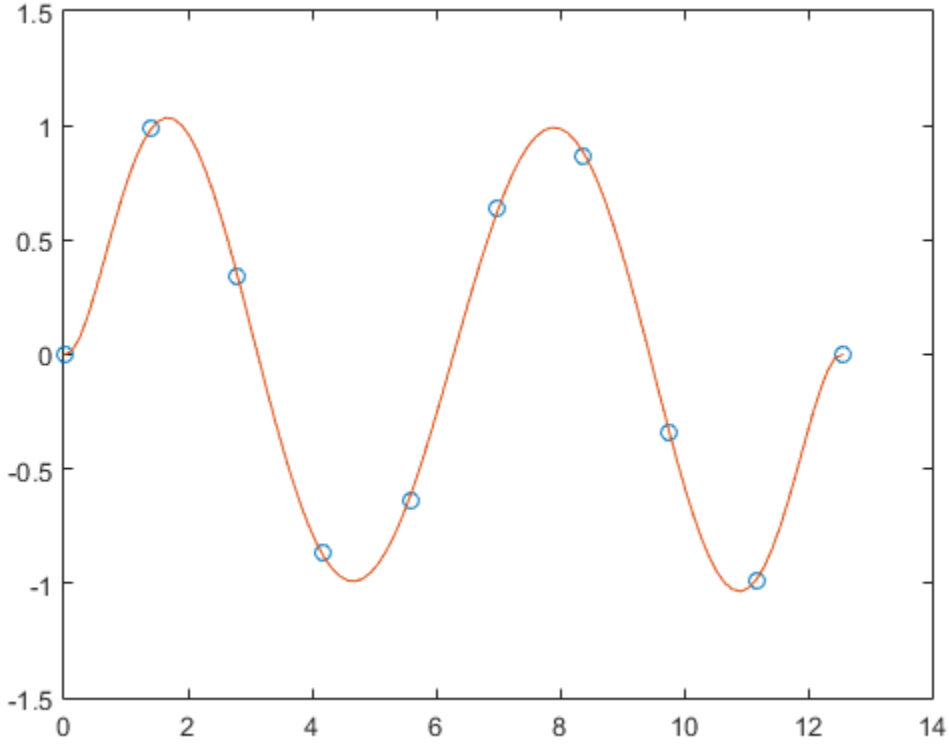
```
x = linspace(0,4*pi,10);
y = sin(x);
```

Use `polyfit` to fit a 7th-degree polynomial to the points.

```
p = polyfit(x,y,7);
```

Evaluate the polynomial on a finer grid and plot the results.

```
x1 = linspace(0,4*pi);
y1 = polyval(p,x1);
figure
plot(x,y,'o')
hold on
plot(x1,y1)
hold off
```



Fit Polynomial to Set of Points

Create a vector of 5 equally spaced points in the interval `[0,1]`, and evaluate $y(x) = (1+x)^{-1}$ at those points.

```
x = linspace(0,1,5);
y = 1./(1+x);
```

Fit a polynomial of degree 4 to the 5 points. In general, for `n` points, you can fit a polynomial of degree `n-1` to exactly pass through the points.

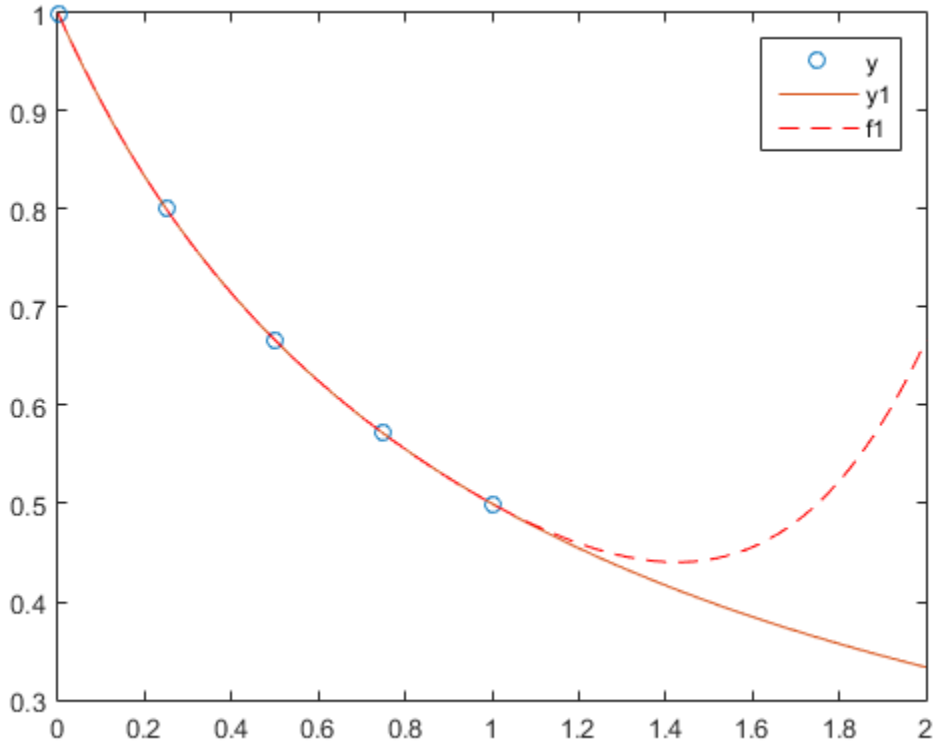
```
p = polyfit(x,y,4);
```

Evaluate the original function and the polynomial fit on a finer grid of points between 0 and 2.

```
x1 = linspace(0,2);
y1 = 1./(1+x1);
f1 = polyval(p,x1);
```

Plot the function values and the polynomial fit in the wider interval `[0,2]`, with the points used to obtain the polynomial fit highlighted as circles. The polynomial fit is good in the original `[0,1]` interval, but quickly diverges from the fitted function outside of that interval.

```
figure
plot(x,y,'o')
hold on
plot(x1,y1)
plot(x1,f1,'r--')
legend('y','y1','f1')
```



Fit Polynomial to Error Function

First generate a vector of `x` points, equally spaced in the interval `[0,2.5]`, and then evaluate `erf(x)` at those points.

```
x = (0:0.1:2.5)';
y = erf(x);
```

Determine the coefficients of the approximating polynomial of degree 6.

```
p = polyfit(x,y,6)
```

```
p =
```

```
    0.0084   -0.0983    0.4217   -0.7435    0.1471    1.1064    0.0004
```

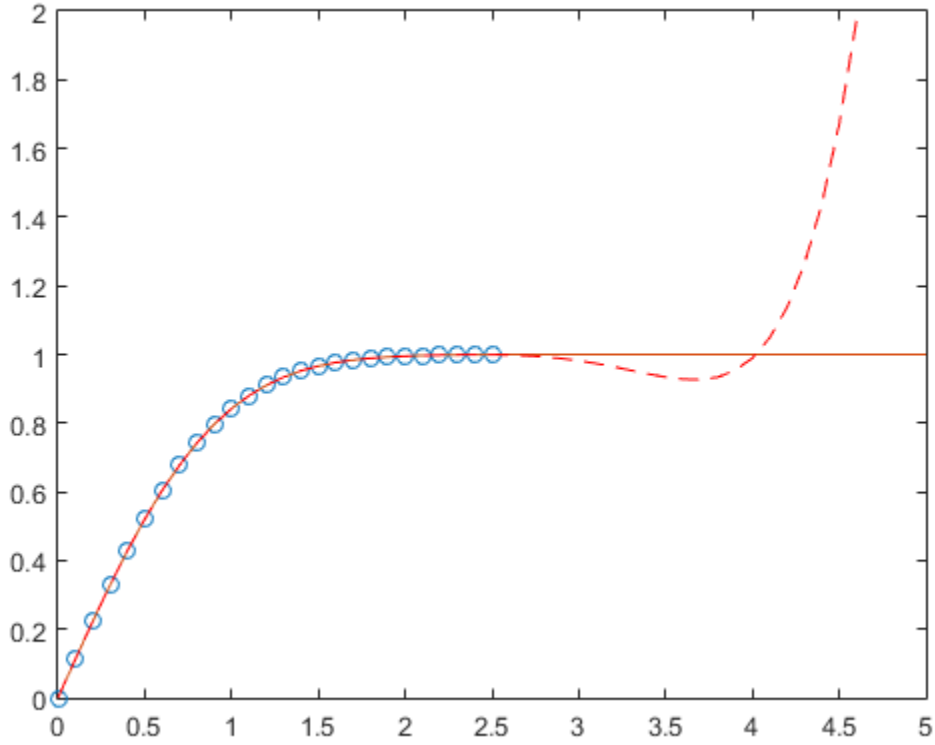
To see how good the fit is, evaluate the polynomial at the data points and generate a table showing the data, fit, and error.

```
f = polyval(p,x);
T = table(x,y,f,y-f,'VariableNames',{'X','Y','Fit','FitError'})
```

```
T =

     X         Y          Fit          FitError
    ___    _______    __________    ___________

      0          0    0.00044117    -0.00044117
    0.1    0.11246       0.11185     0.00060836
    0.2     0.2227       0.22231     0.00039189
    0.3    0.32863       0.32872    -9.7429e-05
    0.4    0.42839        0.4288    -0.00040661
    0.5     0.5205       0.52093    -0.00042568
    0.6    0.60386       0.60408    -0.00022824
    0.7     0.6778       0.67775     4.6383e-05
    0.8     0.7421       0.74183     0.00026992
    0.9    0.79691       0.79654     0.00036515
      1     0.8427       0.84238      0.0003164
    1.1    0.88021       0.88005     0.00015948
    1.2    0.91031       0.91035    -3.9919e-05
    1.3    0.93401       0.93422      -0.000211
    1.4    0.95229       0.95258    -0.00029933
    1.5    0.96611       0.96639    -0.00028097
    1.6    0.97635       0.97652    -0.00016704
    1.7    0.98379       0.98379     8.3306e-07
    1.8    0.98909       0.98893     0.00016278
    1.9    0.99279       0.99253     0.00025791
      2    0.99532       0.99508     0.00024347
    2.1    0.99702       0.99691      0.0001131
    2.2    0.99814       0.99823    -8.8548e-05
    2.3    0.99886       0.99911    -0.00025673
    2.4    0.99931       0.99954    -0.00022451
    2.5    0.99959       0.99936     0.00023151
```

In this interval, the interpolated values and the actual values agree fairly closely. Create a plot to show how outside this interval, the extrapolated values quickly diverge from the actual data.

```
x1 = (0:0.1:5)';
y1 = erf(x1);
f1 = polyval(p,x1);
figure
plot(x,y,'o')
hold on
plot(x1,y1,'-')
plot(x1,f1,'r--')
axis([0  5  0  2])
hold off
```



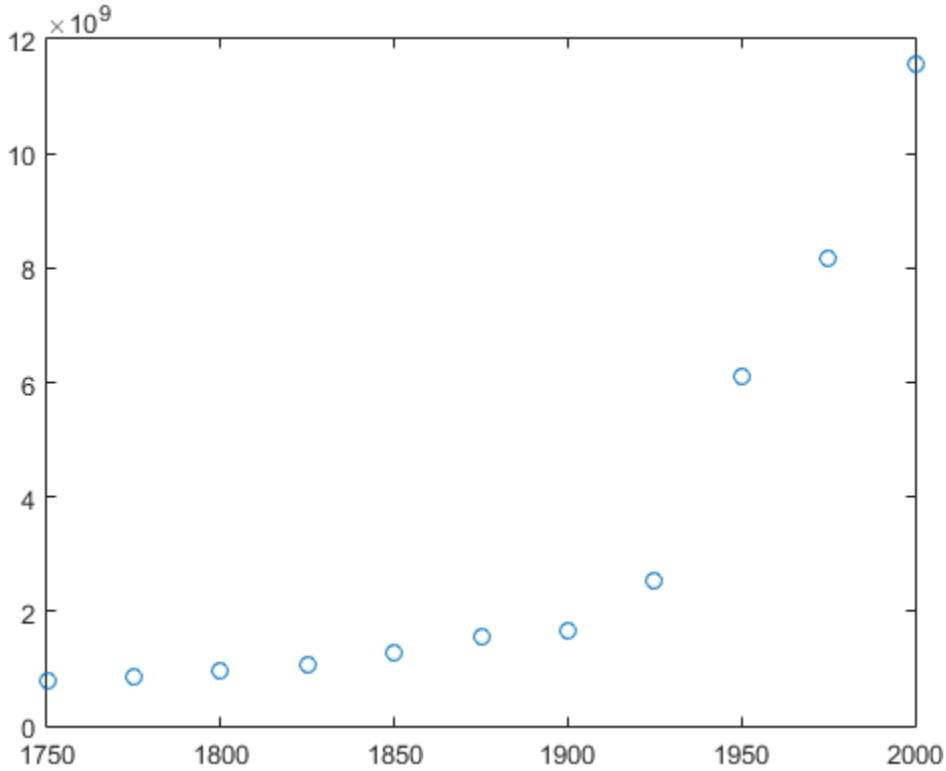
Use Centering and Scaling to Improve Numerical Properties

Create a table of population data for the years 1750 - 2000 and plot the data points.

```
year = (1750:25:2000)';
pop = 1e6*[791 856 978 1050 1262 1544 1650 2532 6122 8170 11560]';
T = table(year, pop)
plot(year,pop,'o')
```

```
T =

    year       pop
    ____    _________
```

```
1750      7.91e+08
1775      8.56e+08
1800      9.78e+08
1825       1.05e+09
1850     1.262e+09
1875     1.544e+09
1900      1.65e+09
1925     2.532e+09
1950     6.122e+09
1975      8.17e+09
2000     1.156e+10
```

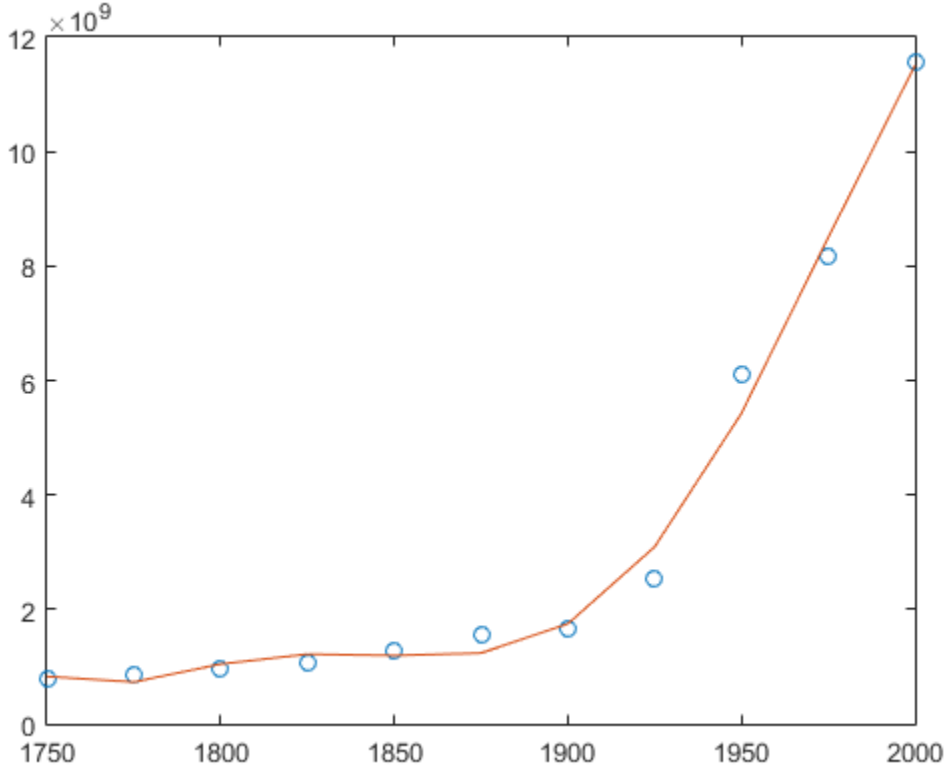


Use `polyfit` with three outputs to fit a 5th-degree polynomial using centering and scaling, which improves the numerical properties of the problem. `polyfit` centers the data in `year` at 0 and scales it to have a standard deviation of 1, which avoids an ill-conditioned Vandermonde matrix in the fit calculation.

```
[p,~,mu] = polyfit(T.year, T.pop, 5);
```

Use `polyval` with four inputs to evaluate `p` with the scaled years, `(year-mu(1))/mu(2)`. Plot the results against the original years.

```
f = polyval(p,year,[],mu);
hold on
plot(year,f)
hold off
```



Input Arguments

x — Query points

vector

Query points, specified as a vector. The points in `x` correspond to the fitted function values contained in `y`.

Warning messages result when `x` has repeated (or nearly repeated) points or if `x` might need centering and scaling.

Data Types: `single` | `double`

Complex Number Support: Yes

y — Fitted values at query points

vector

Fitted values at query points, specified as a vector. The values in `y` correspond to the query points contained in `x`.

Data Types: `single | double`
Complex Number Support: Yes

n — Degree of polynomial fit

positive integer scalar

Degree of polynomial fit, specified as a positive integer scalar. `n` specifies the polynomial power of the left-most coefficient in `p`.

A warning message results if `n` is greater than or equal to `length(x)`.

Output Arguments

p — Least-squares fit polynomial coefficients

vector

Least-squares fit polynomial coefficients, returned as a vector. `p` has length `n+1` and contains the polynomial coefficients in descending powers with the highest power being `n`.

Use `polyval` to evaluate `p` at query points.

S — Error estimation structure

structure

Error estimation structure. This optional output structure is primarily used as an input to the `polyval` function to obtain error estimates. `S` contains the following fields:

Field	Description
`R`	Triangular factor from a QR decomposition of the Vandermonde matrix of `x`
`df`	Degrees of freedom

Field	Description
`normr`	Norm of the residuals

If the data in `y` is random, then an estimate of the covariance matrix of `p` is `(Rinv*Rinv')*normr^2/df`, where `Rinv` is the inverse of `R`.

If the errors in the data in `y` are independent and normal with constant variance, then `[y,delta] = polyval(...)` produces error bounds that contain at least 50% of the predictions. That is, `y ± delta` contains at least 50% of the predictions of future observations at `x`.

`mu` — Centering and scaling values

two element vector

Centering and scaling values, returned as a two element vector. `mu(1)` is `mean(x)`, and `mu(2)` is `std(x)`. These values center the query points in `x` at zero with unit standard deviation.

Use `mu` as the fourth input to `polyval` to evaluate `p` at the scaled points, `(x - mu(1))/mu(2)`.

Limitations

- In problems with many points, increasing the degree of the polynomial fit using `polyfit` does not always result in a better fit. High-order polynomials can be oscillatory between the data points, leading to a *poorer* fit to the data. In those cases, you might use a low-order polynomial fit (which tends to be smoother between points) or a different technique, depending on the problem.
- Polynomials are unbounded, oscillatory functions by nature. Therefore, they are not well-suited to extrapolating bounded data or monotonic (increasing or decreasing) data.

More About

Algorithms

`polyfit` uses `x` to form Vandermonde matrix `V` with `n+1` columns, resulting in the linear system

$$\begin{pmatrix} x_1^{n+1} & x_1^n & \cdots & 1 \\ x_2^{n+1} & x_2^n & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{n+1} & x_n^n & \cdots & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix},$$

which `polyfit` solves with `p = V\y`. Since the columns in the Vandermonde matrix are powers of the vector `x`, the condition number of `V` is often large for high-order fits, resulting in a singular coefficient matrix. In those cases centering and scaling can improve the numerical properties of the system to produce a more reliable fit.

- "Programmatic Fitting"

See Also

`cov` | `lscov` | `poly` | `polyder` | `polyint` | `polyval` | `roots`

Introduced before R2006a

polyint

Integrate polynomial analytically

Syntax

```
polyint(p,k)
polyint(p)
```

Description

`polyint(p,k)` returns a polynomial representing the integral of polynomial `p`, using a scalar constant of integration `k`. Specify `p` and `k` as type `double` or `single`.

`polyint(p)` assumes a constant of integration `k=0`.

See Also

`polyder` | `polyval` | `polyvalm` | `polyfit`

Introduced before R2006a

polyval

Polynomial evaluation

Syntax

```
y = polyval(p,x)
[y,delta] = polyval(p,x,S)
y = polyval(p,x,[],mu)
[y,delta] = polyval(p,x,S,mu)
```

Description

`y = polyval(p,x)` returns the value of a polynomial of degree `n` evaluated at `x`. The input argument `p` is a vector of length `n+1` whose elements are the coefficients in descending powers of the polynomial to be evaluated.

$y = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1}$

`x` can be a matrix or a vector. In either case, `polyval` evaluates `p` at each element of `x`.

`[y,delta] = polyval(p,x,S)` uses the optional output structure `S` generated by `polyfit` to generate error estimates `delta`. `delta` is an estimate of the standard deviation of the error in predicting a future observation at `x` by `p(x)`. If the coefficients in `p` are least squares estimates computed by `polyfit`, and the errors in the data input to `polyfit` are independent, normal, and have constant variance, then `y±delta` contains at least 50% of the predictions of future observations at `x`.

`y = polyval(p,x,[],mu)` or `[y,delta] = polyval(p,x,S,mu)` use $\hat{x} = (x - \mu_1)/\mu_2$ in place of `x`. In this equation, $\mu_1 = \text{mean}(x)$ and $\mu_2 = \text{std}(x)$. The centering and scaling parameters `mu` = [μ_1,μ_2] are optional output computed by `polyfit`.

Examples

The polynomial $p(x) = 3x^2 + 2x + 1$ is evaluated at $x = 5$, 7, and 9 with

```
p = [3 2 1];
polyval(p,[5 7 9])
```

which results in

```
ans =

    86   162   262
```

For another example, see `polyfit`.

More About

Tips

The `polyvalm(p,x)` function, with `x` a matrix, evaluates the polynomial in a matrix sense. See `polyvalm` for more information.

See Also

`polyfit` | `polyvalm` | `polyder` | `polyint`

Introduced before R2006a

polyvalm

Matrix polynomial evaluation

Syntax

```
Y = polyvalm(p,X)
```

Description

`Y = polyvalm(p,X)` evaluates a polynomial in a matrix sense. This is the same as substituting matrix `X` in the polynomial `p`.

Polynomial `p` is a vector whose elements are the coefficients of a polynomial in descending powers, and `X` must be a square matrix.

Examples

The Pascal matrices are formed from Pascal's triangle of binomial coefficients. Here is the Pascal matrix of order 4.

```
X = pascal(4)
X =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

Its characteristic polynomial can be generated with the `poly` function.

```
 p = poly(X)
 p =
     1    -29     72    -29     1
```

This represents the polynomial $x^4 - 29x^3 + 72x^2 - 29x + 1$.

Pascal matrices have the curious property that the vector of coefficients of the characteristic polynomial is palindromic; it is the same forward and backward.

Evaluating this polynomial at each element is not very interesting.

```
polyval(p,X)
ans =
   16      16      16      16
   16      15    -140    -563
   16    -140   -2549  -12089
   16    -563  -12089  -43779
```

But evaluating it in a matrix sense is interesting.

```
polyvalm(p,X)
ans =
   0    0    0    0
   0    0    0    0
   0    0    0    0
   0    0    0    0
```

The result is the zero matrix. This is an instance of the Cayley-Hamilton theorem: a matrix satisfies its own characteristic equation.

See Also

`polyfit` | `polyval`

Introduced before R2006a

posixtime

Convert MATLAB datetime to POSIX time

Syntax

```
p = posixtime(t)
```

Description

`p = posixtime(t)` returns the POSIX® time equivalent to the datetime values in `t`. The POSIX time is the number of seconds (including fractional seconds) elapsed since 00:00:00 1-Jan-1970 UTC, ignoring leap seconds. `p` is a `double` array.

Examples

Convert Datetime Array to POSIX Time

Create a `datetime` array. Then, convert the dates to the equivalent POSIX® time.

```
t = datetime('now') + calmonths(1:3)
```

```
t =

   23-Mar-2015 10:12:07   23-Apr-2015 10:12:07   23-May-2015 10:12:07
```

```
format longG
p = posixtime(t)
```

```
p =

            1427105527.525            1429783927.525            1432375927.525
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

`datenum` | `datetime` | `exceltime` | `juliandate` | `yyyymmdd`

Introduced in R2014b

pow2

Base 2 power and scale floating-point numbers

Syntax

```
X = pow2(Y)
X = pow2(F,E)
```

Description

`X = pow2(Y)` returns an array `X` whose elements are 2 raised to the power `Y`.

`X = pow2(F,E)` computes $x = f * 2^e$ for corresponding elements of `F` and `E`. The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. Arguments `F` and `E` are real and integer arrays, respectively.

Examples

For IEEE arithmetic, the statement `X = pow2(F,E)` yields the values:

```
F          E         X
1/2        1         1
pi/4       2         pi
-3/4       2         -3
1/2        -51       eps
1-eps/2    1024      realmax
1/2        -1021     realmin
```

More About

Tips

This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

See Also

`exp` | `hex2num` | `log2` | `mpower` | `power` | `realmax` | `realmin`

Introduced before R2006a

power, .^

Element-wise power

Syntax

```
C = A.^B
C = power(A,B)
```

Description

`C = A.^B` raises each element of `A` to the corresponding power in `B`.

`C = power(A,B)` is an alternate way to execute `A.^B`, but is rarely used. It enables operator overloading for classes.

Examples

Square Each Element of Vector

Create a vector, `A`, and square each element.

```
A = 1:5;
C = A.^2
```

```
C =

     1     4     9    16    25
```

Find Inverse of Each Matrix Element

Create a matrix, `A`, and take the inverse of each element.

```
A = [1 2 3; 4 5 6; 7 8 9];
C = A.^-1
```

```
C =
```

```
    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
    0.1429    0.1250    0.1111
```

An inversion of the elements is not equal to the inverse of the matrix, which is instead written `A^-1` or `inv(A)`.

Find Roots of Number

Calculate the roots of `-1` to the `1/3` power.

```
A = -1;
B = 1/3;
C = A.^B
```

```
C =

   0.5000 + 0.8660i
```

For negative base `A` and noninteger `B`, if `abs(B)` is less than `1`, the `power` function returns the complex roots of `A`.

Use the `nthroot` function to obtain the real roots.

```
C = nthroot(A,3)
```

```
C =

    -1
```

Input Arguments

A — Base

scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

B — Exponent

scalar | vector | matrix | multidimensional array

Exponent, specified as a scalar, vector, matrix, or multidimensional array. Inputs A and B must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

More About

- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`mpower` | `nthroot` | `realpow`

Introduced before R2006a

ppval

Evaluate piecewise polynomial

Syntax

```
v = ppval(pp,xx)
```

Description

`v = ppval(pp,xx)` returns the value of the piecewise polynomial *f*, contained in `pp`, at the entries of `xx`. You can construct `pp` using the functions `pchip`, `spline`, or the spline utility `mkpp`.

`v` is obtained by replacing each entry of `xx` by the value of *f* there. If *f* is scalar-valued, `v` is of the same size as `xx`. `xx` may be N-dimensional.

If `pp` was constructed by `pchip`, `spline`, or `mkpp` using the orientation of non-scalar function values specified for those functions, then:

If `f` is `[D1,..,Dr]`-valued, and `xx` is a vector of length `N`, then `V` has size `[D1,...,Dr, N]`, with `V(:,...,:,J)` the value of `f` at `xx(J)`.

If `f` is `[D1,..,Dr]`-valued, and `xx` has size `[N1,...,Ns]`, then `V` has size `[D1,...,Dr, N1,...,Ns]`, with `V(:,...,:, J1,...,Js)` the value of `f` at `xx(J1,...,Js)`.

Examples

Compare the result of integrating *cos*(*x*) between 0 and 10 to the result of integrating a piece-wise polynomial approximation of the same function.

```
a = 0; b = 10;
int1 = integral(@cos,a,b)

int1 =
   -0.5440
```

Create a piece-wise polynomial approximation of *cos*(*x*) and integrate over the same interval.

```
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = integral(@(x)ppval(pp,x),a,b)

int2 =
   -0.5485
```

See Also

`mkpp` | `spline` | `unmkpp`

Introduced before R2006a

prefdir

Folder containing preferences, history, and layout files

Syntax

```
prefdir
folder = prefdir
folder = prefdir(1)
```

Description

`prefdir` returns the folder that contains

- Preferences for MATLAB and related products (`matlab.prf`)
- Command history file (`History.xml`)
- MATLAB shortcuts (`shortcuts_2.xml`)
- MATLAB desktop layout files (`MATLABDesktop.xml` and `Your_Saved_LayoutMATLABLayout.xml`)
- Other related files

`folder = prefdir` assigns to `folder` the name of the folder containing preferences and related files.

`folder = prefdir(1)` creates a folder for preferences and related files if one does not exist. If the folder does exist, the name is assigned to `folder`.

Examples

View the location of the preferences folder:

```
prefdir
```

Make the preferences folder become the current folder:

```
cd(prefdir)
```

```
% Then, view the files for customizing MathWorks products:
dir
```

On Windows platforms, go directly to the preferences folder in Microsoft Windows Explorer

```
winopen(prefdir)
```

More About

Tips

- You must have write access to the preferences folder. Otherwise, MATLAB generates an error in the Command Window when you try to change preferences. This can happen if the folder is hidden, for example: `myname/.matlab/R2009a`.
- "Preferences Folder and Files MATLAB Uses When Multiple MATLAB Releases Are Installed"

See Also

`preferences` | `getpref` | `setpref`

Introduced before R2006a

preferences

Open Preferences dialog box

Syntax

```
preferences
```

Description

`preferences` displays the Preferences dialog box, from which you can make changes to options for MATLAB and related products.

More About

- "Preferences"

See Also

`prefdir`

Introduced before R2006a

primes

Prime numbers less than or equal to input value

Syntax

```
p = primes(n)
```

Description

`p = primes(n)` returns a row vector containing all the prime numbers less than or equal to `n`. The data type of `p` is the same as that of `n`.

Examples

Primes Less Than or Equal to 25

```
p = primes(25)
```

```
p =

    2     3     5     7    11    13    17    19    23
```

Primes Less Than or Equal to an Unsigned Integer

```
n = uint16(12);
p = primes(n)
```

```
p =

    2     3     5     7    11
```

Input Arguments

n — Input value

scalar, real integer value

Input value, specified as a scalar that is a real integer value.

Example: `10`

Example: `int16(32)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`factor` | `isprime`

Introduced before R2006a

print

Print figure or save to specific file format

Syntax

```
print(filename,formattype)
print(filename,formattype,formatoptions)

print
print(printer)
print(driver)
print(printer,driver)

print('-clipboard',clipboardformat)

print(resolution, ___ )
print(renderer, ___ )
print('-noui', ___ )
print(fig, ___ )

cdata = print('-RGBImage');
```

Description

`print(filename,formattype)` saves the current figure to a file using the specified file format. If the file name does not include an extension, then `print` appends the appropriate one.

`print(filename,formattype,formatoptions)` specifies additional options that are available for some formats.

`print` prints the current figure to the default printer.

`print(printer)` specifies the printer. Specify the printer as a string containing the printer name preceded by `-P`, for example, `'-Pmy printer'`. The printer must be setup on your system.

`print(driver)` specifies the driver for the temporary file that `print` generates and sends to the printer. Specify a driver if you want to ensure that the printed output is either black and white or color.

`print(printer,driver)` specifies the printer and the driver.

`print('-clipboard',clipboardformat)` copies the current figure to the clipboard using the specified format. You can paste the copied figure into other applications.

`print(resolution, ___ )` uses the specified resolution. Specify the resolution as a string containing an integer value preceded by `-r`, for example, `'-r200'`. Use this option with any of the input arguments from the previous syntaxes.

`print(renderer, ___ )` uses the specified renderer. Specify the renderer as either `'-painters'` or `'-opengl'`.

`print('-noui', ___ )` excludes user interface controls, such as push buttons and sliders, from the saved or printed output. It does not exclude user interface objects that can contain an axes, such as a uitab or uipanel.

`print(fig, ___ )` saves or prints the figure or Simulink block diagram specified by `fig` instead of the current figure.

`cdata = print('-RGBImage');` returns the RGB image data for the current figure. This option differs from screen captures in that all printing features apply to the output. You can additionally specify the resolution, renderer, '-noui', and fig options with this syntax. However, you cannot specify a Simulink block diagram.

Examples

Print Paper Copy of Figure

Create a bar chart and print it to your system default printer. If you do not specify the figure to print, then `print` uses the current figure.

```
bar(1:10)
print
```

Copy Figure to Clipboard

Create a plot and copy it to the system clipboard.

```
plot(1:10)
print('-clipboard','-dmeta')
```

You can paste the copied plot into other applications.

Save Figure as Image File

Create a plot and save it as a PNG image file.

```
bar(1:10)
print('BarPlot','-dpng')
```

`print` saves the plot as `BarPlot.png`.

Save Figure as Vector Graphics File

Create a plot and save it as an Encapsulated PostScript file.

```
bar(1:10)
print('BarPlot','-depsc')
```

`print` saves the plot as `BarPlot.eps`.

Add TIFF Preview to EPS File

Save the current figure as an Encapsulated PostScript File and add a TIFF preview.

```
surf(peaks)
print('SurfacePlot','-depsc','-tiff')
```

Specify Figure to Save

Save a specific figure by passing its object variable to `print`.

```
fig = figure;
plot(1:10)
print(fig,'MySavedPlot','-dpng')
```

Alternatively, refer to a figure using the value of its `Number` property, which is the integer value that displays in the figure window title bar. For example, save the figure with `Figure 2` displayed in the title bar. Precede the integer value by `-f`.

```
figure(2);
plot(1:10)
```

```
print('-f2','MySavedPlot','-dpng')
```

Save Figure at Screen Size and Resolution

Save a surface plot to a PNG file. Set the `PaperPositionMode` property for the figure to `'auto'` so that it saves at the size displayed on the screen. Use `'-r0'` to save it with screen resolution.

```
surf(peaks)
set(gcf,'PaperPositionMode','auto')
print('PeaksSurface','-dpng','-r0')
```

Save Figure Without Saving UIControls

Create a figure with a push button that clears the axes. Save the figure to a JPEG file without saving the push button.

```
surf(peaks)
uicontrol('Style','pushbutton','String','Clear',...
    'Position',[20 20 50 20],'Callback','cla');
print('SurfacePlot','-djpeg','-noui')
```

Return RGB Image Data for Figure

Return the RGB image data for a figure.

```
surf(peaks)
cdata = print('-RGBImage');
```

Display the image data at full resolution using `imshow`.

```
figure
imshow(cdata)
```

Input Arguments

`filename` — File name

string

File name, specified as a string containing the desired file name and path.

Example: `'My Saved Chart'`

Example: `'Folder\My Saved Chart'`

The maximum file name length, including the path, is operating system and file format specific. Typically, the file name should be no more than 126 characters, or if you include the path, then no more than 128 characters.

`formattype` — File format
`'-djpeg' | '-dpng' | '-dtiff' | '-dpdf' | '-deps' | ...`

File format, specified as one of the options in these tables.

Bitmap Image File

Bitmap images contain a pixel-based representation of the figure. The size of the generated file depends on the figure and the format used. Bitmap images are widely used by Web browsers and other applications that display graphics. However, they do not scale well and you cannot modify individual graphics objects, such as lines and text, in other graphics applications.

This table lists the supported bitmap image formats.

Option	Bitmap Image Format	Corresponding File Extension
`'jpeg'`	JPEG 24-bit	`.jpg`
`'png'`	PNG 24-bit	`.png`
`'tiff'`	TIFF 24-bit (compressed)	`.tif`
`'tiffn'`	TIFF 24-bit (not compressed)	`.tif`
`'meta'`	Enhanced metafile (Windows only)	`.emf`
`'bmpmono'`	BMP Monochrome	`.bmp`
`'bmp'`	BMP 24-bit	`.bmp`
`'bmp16m'`	BMP 24-bit	`.bmp`
`'bmp256'`	BMP 8-bit (256 color, uses a fixed colormap)	`.bmp`
`'hdf'`	HDF 24-bit	`.hdf`
`'pbm'`	PBM (plain format) 1-bit	`.pbm`

Option	Bitmap Image Format	Corresponding File Extension
'pbmraw'	PBM (raw format) 1-bit	.pbm
'pcxmono'	PCX 1-bit	.pcx
'pcx24b'	PCX 24-bit color (three 8-bit planes)	.pcx
'pcx256'	PCX 8-bit newer color (256 color)	.pcx
'pcx16'	PCX older color (EGA/VGA 16-color)	.pcx
'pgm'	PGM (plain format)	.pgm
'pgmraw'	PGM (raw format)	.pgm
'ppm'	PPM (plain format)	.ppm
'ppmraw'	PPM (raw format)	.ppm

Vector Graphics File

Vector graphics files store commands that redraw the figure. This type of format scales well, but can result in a large file. Additionally, it might not produce the correct 3-D arrangement of objects in certain cases. Some applications support extensive editing of vector graphics formats. However, some applications do not support editing beyond resizing the graphic. In general, you should try to make all the necessary changes while your figure is still in MATLAB.

If you set the `Renderer` property for the figure, then `print` uses that renderer when generating output. Otherwise, `print` chooses the appropriate renderer. Typically, `print` uses the Painters renderer when generating vector graphics files. For some complex figures, `print` uses the OpenGL renderer instead. If it uses the OpenGL renderer, then the vector graphics file contains an embedded image, which might limit the extent to which you can edit the image in other applications. To ensure that `print` uses the Painters renderer, set the `Renderer` property for the figure to `'painters'` or specify `'-painters'` as an input argument to `print`.

Note: The default figure renderer is OpenGL. If the figure renderer differs from the renderer used when generating output, some details of the saved figure can differ from

the figure on the display. If necessary, you can make the displayed figure and the saved figure use the same renderer. Set the `Renderer` property for the figure or specify the renderer input argument to the `print` function.

This table lists the supported vector graphics formats.

Option	Vector Graphics Format	Corresponding File Extension
`'pdf'`	Full page Portable Document Format (PDF) color	`.pdf`
`'eps'`	Encapsulated PostScript (EPS) Level 3 black and white	`.eps`
`'epsc'`	Encapsulated PostScript (EPS) Level 3 color	`.eps`
`'eps2'`	Encapsulated PostScript (EPS) Level 2 black and white	`.eps`
`'epsc2'`	Encapsulated PostScript (EPS) Level 2 color	`.eps`
`'meta'`	Enhanced Metafile (Windows only)	`.emf`
`'svg'`	SVG (scalable vector graphics)	`.svg`
`'ps'`	Full-page PostScript (PS) Level 3 black and white	`.ps`
`'psc'`	Full-page PostScript (PS) Level 3 color	`.ps`
`'ps2'`	Full-page PostScript (PS) Level 2 black and white	`.ps`
`'psc2'`	Full-page PostScript (PS) Level 2 color	`.ps`

Note: Only PDF and PS formats use the `PaperOrientation` property of the figure and the `left` and `bottom` elements of the `PaperPosition` property. Other formats ignore these values.

`formatoptions` — Additional formatting options

`'-tiff' | '-loose' | '-cmyk' | '-append'`

Additional formatting options supported by some file formats, specified as one or more of these values:

- `'-tiff'` — Include a TIFF preview. EPS files only.
- `'-loose'` — Use a loose bounding box. EPS and PS files only.
- `'-cmyk'` — Use CMYK colors instead of RGB colors. EPS and PS files only.
- `'-append'` — Append the figure to an existing PS file. PS files only.

Example: `print('my file','-deps','-tiff','loose')` saves the current figure to the file `my file.eps` using a loose bounding box and includes a TIFF preview.

`printer` — Printer name

string containing `-P` and printer name

Printer name, specified as a string containing `-P` and printer name.

Example: `'-Pmy local printer'`

To view a list of available printers, use this command:

```
[~,printers] = findprinters
```

If you do not specify a printer, then `print` uses the system default printer. If you want to set up a new printer or select a different default printer, use the operating system printer management utilities. Restart MATLAB if you do not see a printer that is already setup.

`driver` — Printer driver

`'-dwin' | '-dwinc' | '-dps' | '-dpsc' | '-dps2' | '-dpsc2'`

Printer driver, specified as `'-dwin'`, `'-dwinc'`, `'-dps'`, `'-dpsc'`, `'-dps2'`, or `'-dpsc2'`. If you do not specify a driver, then `print` relies on your operating system to choose the driver.

The option you use depends on your system, for example:

System	Driver	Output
Windows	`'-dwin'`	Black and white
	`'-dwinc'`	Color
Linux or Mac OS X	`'-dps'` or `'-dps2'`	Black and white
	`'-dpsc'` or `'-dpsc2'`	Color

`clipboardformat` — Format copied to clipboard

`-dmeta | -dbitmap | -dpdf`

Format copied to clipboard, specified as one of these options:

- `'-dmeta'` — Enhanced metafile (Windows only)
- `'-dbitmap'` — Bitmap image (Windows and Mac OS X)
- `'-dpdf'` — PDF file (Windows and Mac OS X)

`resolution` — Resolution

string containing `-r` and integer

Resolution, specified as a string containing `-r` and an integer value indicating the resolution in dots per inch. For example, `'-r300'` sets the output resolution to 300 dots per inch. To specify screen resolution, use `'-r0'`.

In general, using a higher resolution value yields higher quality output, but at the cost of higher memory use and larger output files. The higher the resolution setting, the longer it takes to render your figure. By default, the resolution is 90 dpi for Simulink models and 150 dpi for figures. For typical laser-printer output, the default resolution of 150 dpi for figures is normally adequate. However, if you are preparing figures for high-quality printing, such as a textbook or color brochures, you might want to use 200 or 300 dpi. The resolution can be limited by the printer's capabilities.

Specifying the resolution is useful if you want to scale a bitmap image or a vector graphics file that contains an embedded image. At ordinary magnification, the effect is subtle. The effect is most apparent when viewing the output at a higher magnification or when printed.

`renderer` — Graphics renderer

`'-opengl' | '-painters'`

Graphics renderer, specified as `'-opengl'` or `'-painters'`.

- `'-opengl'` — OpenGL renderer. Use this renderer when saving bitmap images. OpenGL produces a bitmap image even with vector formats, which might limit the extent to which you can edit the image in other applications.
- `'-painters'` — Painters renderer. Use this renderer when saving vector graphics files. If you save to a vector graphics file and if the figure `RendererMode` property is set to `'auto'`, then `print` automatically attempts to use the Painters renderer. If you want to ensure that your output format is a true vector graphics file, then specify the Painters renderer. For example:

  ```
  print('-painters','-deps','myVectorFile')
  ```

Note: In some cases, saving a file with the `'-painters'` option can cause longer rendering times and, in rare cases, might not accurately arrange graphics objects in 3-D views. Additionally, the Painters renderer cannot print or save lines thinner than 1 pixel.

If you do not specify the renderer, then `print` automatically uses the appropriate renderer to produce the output format that you request. However, if you set the `Renderer` property for the figure, then `print` uses that renderer when generating output.

`fig` — Figure or Simulink block diagram

figure object | Simulink block diagram

Figure object or Simulink block diagram. You can refer to a figure using either its object variable name or using the figure number preceded by `-f`. For example, `-f2` refers to the figure with a `Number` property value of `2`. When specifying a Simulink block diagram, precede the model name with `-s`. Specify the current model using `'-s'`.

Output Arguments

`cdata` — Image data

n-by-m-by-3 array

Image data, returned as an n-by-m-by-3 array.

Limitations

- `print` does not support capturing ActiveX controls.
- Starting MATLAB in no display mode on Linux or using the `-noFigureWindows` startup option on any platform has these limitations for `print`:
  - Does not print or save uicontrols.
  - Always uses the painters renderer, even if you specify the `'-opengl'` option.

More About

Tips

- You can set properties of the figure to control some printing and saving parameters. This table lists properties of the figure related to printing and saving.

Figure Property	Description
`PaperPosition`	Size of the printed or saved figure. If printing to a printer or a full-page output format, then this property also determines the figure location on the page.
`PaperPositionMode`	Specifies whether to use the `PaperPosition` property or the size of the figure on the screen to set the size of the printed or saved figure.
`InvertHardcopy`	Specifies whether to use the current background color of the figure or to change the background color to white when printing or saving the figure.
`PaperOrientation`	Figure orientation on printed page.
`PaperType`	Standard printer paper size.
`PaperSize`	Custom width and height of printer paper.
`PaperUnits`	Units for the `PaperSize` and `PaperPosition` properties.

- The `print` function returns a warning message when you print a figure that has a callback defined for the `SizeChangedFcn` property. To avoid the warning, set the `PaperPositionMode` property for the figure to `'auto'`.

See Also

`getframe` | `saveas` | `savefig`

Introduced before R2006a

printopt

Configure printer defaults

Syntax

```
[pcmd,dev] = printopt
```

Description

`[pcmd,dev] = printopt` returns strings containing the current system-dependent printing command and output device. `printopt` is a file used by `print` to produce the hard-copy output. You can edit the file `printopt.m` to set your default printer type and destination.

`pcmd` and `dev` are platform-dependent strings. `pcmd` contains the command that `print` uses to send a file to the printer. `dev` contains the printer driver or graphics format option for the `print` command. Their defaults are platform dependent.

Platform	Print Command	Driver or Format
Mac and UNIX	`lpr -r`	`-dps2`
Windows	COPY /B %s LPT1:	`-dwin`

See Also

`printdlg` | `print`

Introduced before R2006a

printdlg

Open figure Print dialog box

Note: The `-crossplatform` and `-setup` input arguments have been removed in R2014b. They no longer have any effect.

Syntax

```
printdlg
printdlg(fig)
```

Description

`printdlg` prints the current figure.

`printdlg(fig)` creates a modal dialog box from which you can print the figure window identified by the handle `fig`. Uimenus do not print.

More About

Tips

If you want to set up a new printer, use the operating system printer management utilities. Restart MATLAB if you do not see the printer which is already setup.

See Also

`print` | `printopt` | `printpreview`

Introduced before R2006a

printpreview

Open figure Print Preview dialog box

Syntax

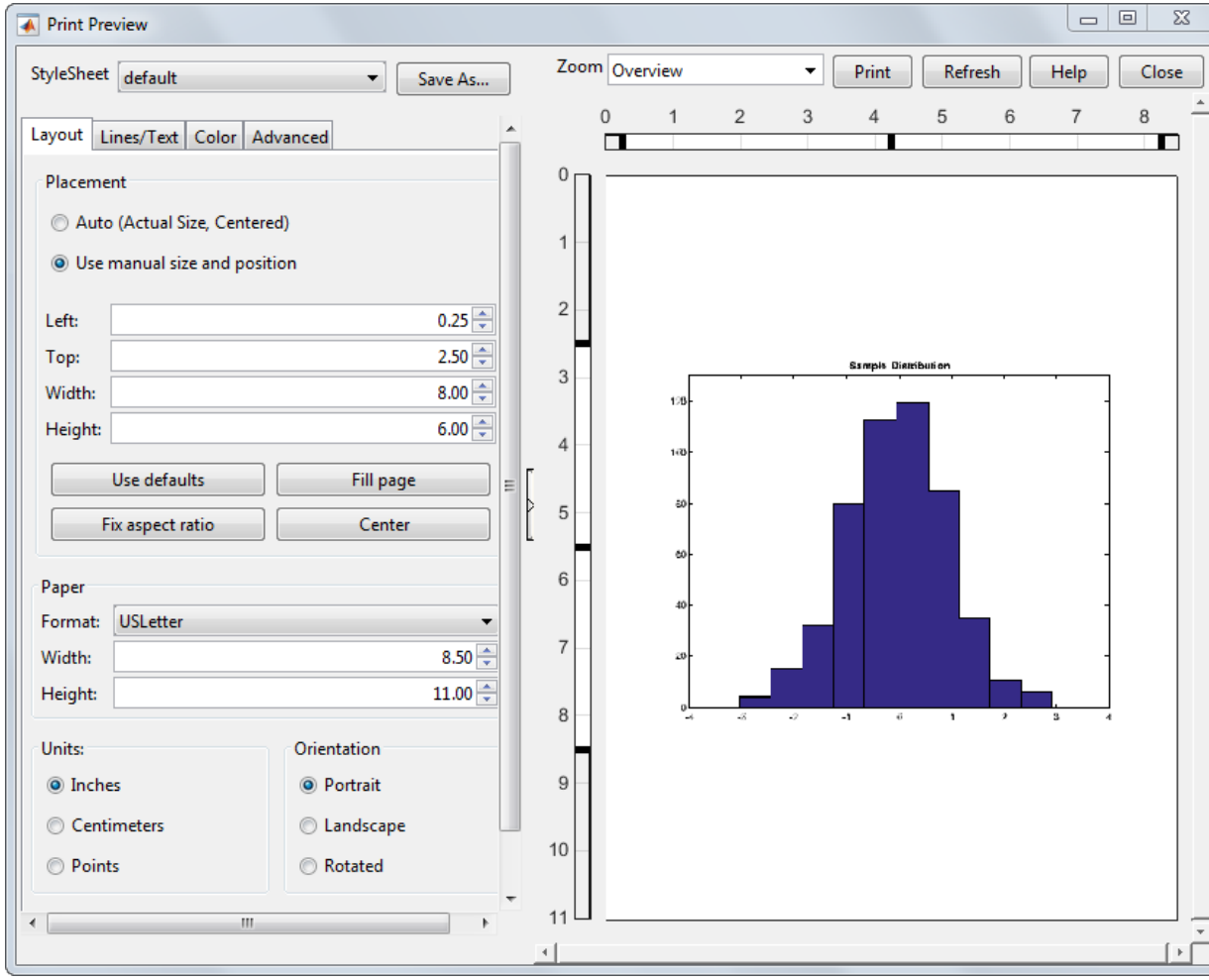
```
printpreview
printpreview(f)
```

Description

`printpreview` displays a dialog box showing the figure in the currently active figure window as it will print. A scaled version of the figure displays in the right-hand pane of the dialog box.

`printpreview(f)` displays a dialog box showing the figure having the handle `f` as it will print.

Use the Print Preview dialog box, shown below, to control the layout and appearance of figures before sending them to a printer or print file. Controls are grouped into four tabbed panes: **Layout**, **Lines/Text**, **Color**, and **Advanced**.



Right Pane Controls

You can position and scale plots on the printed page using the rulers in the right-hand pane of the Print Preview dialog. Use the outer ruler handlebars to change margins. Moving them changes plot proportions. Use the center ruler handlebars to change the position of the plot on the page. Plot proportions do not change, but you can move portions of the plot off the paper. The buttons on that pane let you refresh the plot, close the dialog (preserving all current settings), print the page immediately, or obtain context-

sensitive help. Use the **Zoom** box and scroll bars to view and position page elements more precisely.

The Layout Tab

Use the **Layout** tab, shown above, to control the paper format and placement of the plot on printed pages. The following table summarizes the **Layout** options:

Group	Option	Description
Placement	**Auto**	Let MATLAB decide placement of plot on page*
	Use manual...	Specify position parameters for plot on page*
	Top, Left, Width, Height	Standard position parameters in current units
	Use defaults	Revert to default position
	Fill page	Expand figure to fill printable area (see note below)
	Fix aspect ratio	Correct height/width ratio
	Center	Center plot on printed page
Paper	**Format**	U.S. and ISO® sheet size selector
	Width, Height	Sheet size in current units
Units	**Inches**	Use inches as units for dimensions and positions
	Centimeters	Use centimeters as units for dimensions and positions
	Points	Use points as units for dimensions and positions
Orientation	**Portrait**	Upright paper orientation
	Landscape	Sideways paper orientation
	Rotated	Currently the same as **Landscape**

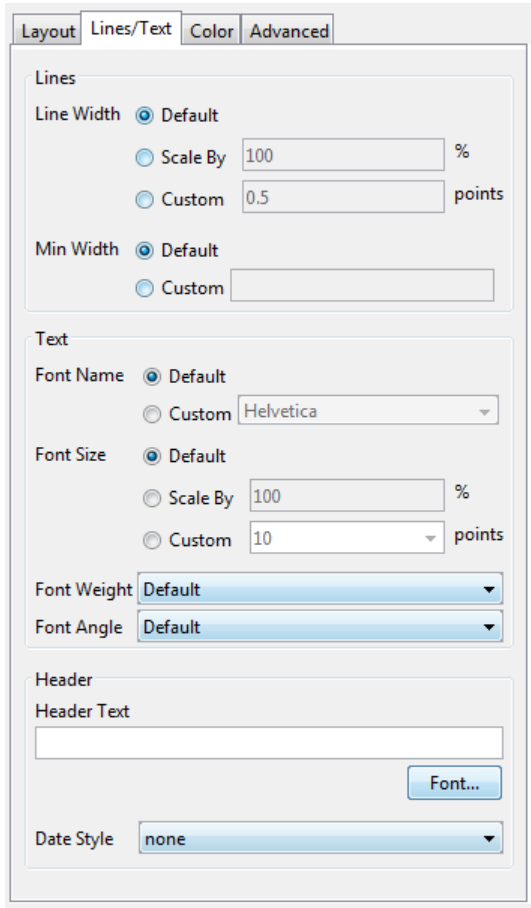
* Selecting **Auto** in the Placement group sets the figure `PaperPositionMode` to `'auto'` and disables the controls in that panel. Selecting **Use manual size and position** sets the figure `PaperPositionMode` to `'manual'` and enables the controls. If you set

`PaperPositionMode` programmatically, the print preview Placement controls respond accordingly.

Note: Selecting the **Fill page** option changes the `PaperPosition` property to fill the page, allowing objects in normalized units to expand to fill the space. If an object within the figure has an absolute size, for example a table, it can overflow the page when objects with normalized units expand. To avoid having objects fall off the page, do not use **Fill page** under such circumstances.

The Lines/Text Tab

Use the **Lines/Text** tab, shown below, to control the line weights, font characteristics, and headers for printed pages. The following table summarizes the **Lines/Text** options:

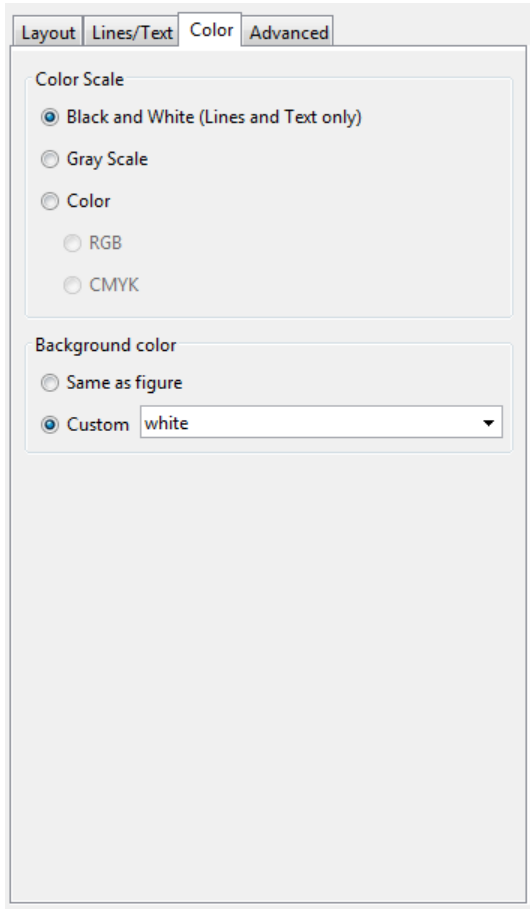


Group	Option	Description
Lines	**Line Width**	Scale all lines by a percentage from 0 upward (100 being no change), print lines at a specified point size, or default line widths used on the plot
	Min Width	Smallest line width (in points) to use when printing; defaults to 0.5 point
Text	**Font Name**	Select a system font for all text on plot, or default to fonts currently used on the plot

Group	Option	Description
	Font Size	Scale all text by a percentage from 0 upward (100 being no change), print text at a specified point size, or default to this used on the plot
	Font Weight	Select `Normal` ... `Bold` font styling for all text from drop-down menu or default to the font weights used on the plot
	Font Angle	Select `Normal`, `Italic` or `Oblique` font styling for all text from drop-down menu or default to the font angles used on the plot
Header	**Header Text**	Type the text to appear on the header at the upper left of printed pages, or leave blank for no header
	Date Style	Select a date format to have today's date appear at the upper left of printed pages, or `none` for no date

The Color Tab

Use the **Color** tab, shown below, to control how colors are printed for lines and backgrounds. The following table summarizes the **Color** options:

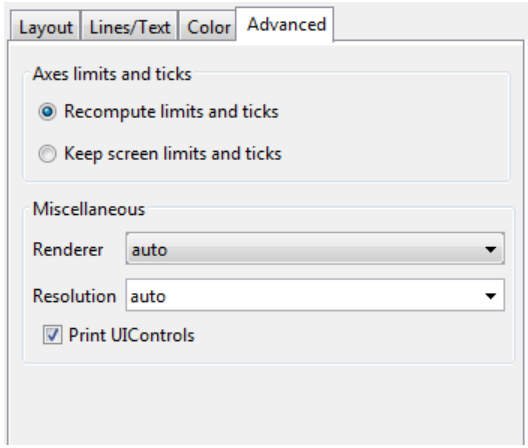


Group	Option	Description
Color Scale	**Black and White**	Select to print lines and text in black and white, but use color for patches and other objects
	Gray Scale	Convert colors to shades of gray on printed pages
	Color	Print everything in color, matching colors on plot; select `RGB` (default) or `CMYK` color model for printing

Group	Option	Description
Background Color	**Same as figure**	Print the figure's background color as it is
	Custom	Select a color name, or type a colorspec for the background; `white` (default) implies no background color, even on colored paper.

The Advanced Tab

Use the **Advanced** tab, shown below, to control finer details of printing, such as limits and ticks, renderer, resolution, and the printing of UIControls. The following table summarizes the **Advanced** options:



Group	Option	Description
Axes limits and ticks	**Recompute limits and ticks**	Redraw *x*- and *y*-axes ticks and limits based on printed plot size (default)
	Keep limits and ticks	Use the *x*- and *y*-axes ticks and limits shown on the plot when printing the previewed figure
Miscellaneous	**Renderer**	Select a rendering algorithm for printing: `painters`, `opengl`, or `auto` (default)

Group	Option	Description
	Resolution	Select resolution to print at in dots per inch: `150`, `300`, `600`, or `auto` (default), or type in any other positive value
	Print UIControls	Print all visible UIControls in the figure (default), or uncheck to exclude them from being printed

Alternatives

Use **File > Print Preview** on the figure window menu to access the Print Preview dialog box, described below. For details, see "Print Figure from File Menu".

More About

- "Print Figure from File Menu"

See Also

`print`

Introduced before R2006a

prod

Product of array elements

Syntax

```
B = prod(A)
B = prod(A,dim)
B = prod( ___ ,type)
```

Description

`B = prod(A)` returns the product of the array elements of `A`.

- If `A` is a vector, then `prod(A)` returns the product of the elements.
- If `A` is a nonempty matrix, then `prod(A)` treats the columns of `A` as vectors and returns a row vector of the products of each column.
- If `A` is an empty 0-by-0 matrix, `prod(A)` returns `1`.
- If `A` is a multidimensional array, then `prod(A)` acts along the first nonsingleton dimension and returns an array of products. The size of this dimension reduces to `1` while the sizes of all other dimensions remain the same.

`prod` computes and returns `B` as `single` when the input, `A`, is `single`. For all other numeric and logical data types, `prod` computes and returns `B` as `double`.

`B = prod(A,dim)` returns the products along dimension `dim`. For example, if `A` is a matrix, `prod(A,2)` is a column vector containing the products of each row.

`B = prod( ___ ,type)` returns an array in the class specified by `type`, using any of the input arguments in the previous syntaxes. `type` can be `'double'`, `'native'`, or `'default'`.

Examples

Product of Elements in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9]
```

```
A =

     1     4     7
     2     5     8
     3     6     9
```

Find the product of the elements in each column.

```
B = prod(A)
```

```
B =

     6   120   504
```

The length of the first dimension is `1`, and the length of the second dimension matches `size(A,2)`.

Logical Input with Double Output

Create an array of logical values.

```
A = [true false; true true]
```

```
A =

     1     0
     1     1
```

Find the product of the elements in each column.

```
B = prod(A)
```

```
B =

     1     0
```

The output is `double`.

```
class(B)
```

```
ans =
```

```
double
```

Product of Elements in Each Row

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9]
```

```
A =

     1     4     7
     2     5     8
     3     6     9
```

Find the product of the elements in each row and reduce the length of the second dimension to 1.

```
dim = 2;
B = prod(A,dim)
```

```
B =

    28
    80
   162
```

The length of the first dimension matches `size(A,1)`, and the length of the second dimension is 1.

Product of Elements in Each Plane

Create a 3-by-3-by-2 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9];
A(:,:,2)=[10:3:16;11:3:17;12:3:18]
```

```
A(:,:,1) =

     1     4     7
     2     5     8
     3     6     9


A(:,:,2) =
```

```
    10    13    16
    11    14    17
    12    15    18
```

Find the product of each element in the first plane with its corresponding element in the second plane.

```
dim = 3;
B = prod(A,dim)
```

```
B =

    10    52   112
    22    70   136
    36    90   162
```

The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is `1`.

Single-Precision Input Treated as Double

Create a 3-by-3 array of single-precision values.

```
A = single([1200 1500 1800; 1300 1600 1900; 1400 1700 2000])
```

```
A =

        1200        1500        1800
        1300        1600        1900
        1400        1700        2000
```

Find the product of the elements in each row by multiplying in double precision.

```
B = prod(A,2,'double')
```

```
B =

   1.0e+09 *

    3.2400
    3.9520
    4.7600
```

The output is double precision.

```
class(B)
```

```
ans =

double
```

Integer Data Type for Input and Output

Create a 3-by-3 array of 8-bit unsigned integers.

```
A = uint8([1:3:7;2:3:8;3:3:9])
```

```
A =

    1    4    7
    2    5    8
    3    6    9
```

Find the product of the elements in each column natively in `uint8`.

```
B = prod(A,'native')
```

```
B =

    6  120  255
```

The result is an array of 8-bit unsigned integers.

```
class(B)
```

```
ans =

uint8
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Complex Number Support: Yes

`dim` — Dimension to operate along

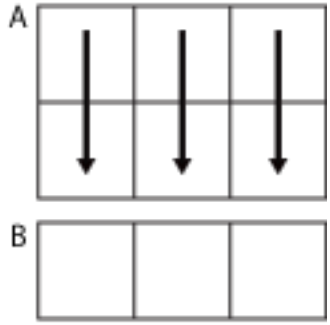
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(B,dim)` is `1`, while the sizes of all other dimensions remain the same.

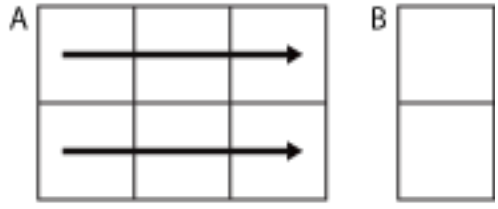
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `prod(A,1)` returns a row vector containing the product of the elements in each column.



prod(A,1)

- If `dim = 2`, then `prod(A,2)` returns a column vector containing the product of the elements in each row.



prod(A,2)

`prod` returns `A` when `dim` is greater than `ndims(A)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`type` — Output class

`'default'` (default) | `'double'` | `'native'`

Output class, specified as `'default'`, `'double'`, or `'native'`, and which defines the data type of the output, `B`.

`type`	Output data type
`'default'`	`double`, unless the input data type is `single`. In which case, the output data type is `single`.
`'double'`	`double`
`'native'`	same data type as the input array, `A`

Data Types: `char`

Output Arguments

`B` — Product array

scalar | vector | matrix | multidimensional array

Product array, returned as a scalar, vector, matrix, or multidimensional array.

The class of `B` is as follows:

- If the `type` argument specifies `'default'` or is not used
  - and the input is not `single`, then the output is `double`.
  - and the input is `single`, then the output is `single`.
- If the `type` argument specifies `'double'`, then the output is `double` regardless of the input data type.
- If the `type` argument specifies `'native'`, then the output is the same data type as the input.

More About

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of `X`.

See Also

`cumprod` | `diff` | `ndims` | `sum`

Introduced before R2006a

profile

Profile execution time for function

Syntax

```
profile on
profile -history
profile -nohistory
profile -history -historysize integer
profile -timer clock
profile -history -historysize integer -timer clock
profile off
profile resume
profile clear
profile viewer
S = profile('status')
stats = profile('info')
```

Description

The `profile` function helps you debug and optimize MATLAB code files by tracking their execution time. For each MATLAB function, MATLAB local function, or MEX-function in the file, `profile` records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Some people use `profile` simply to see the child functions; see also `matlab.codetools.requiredFilesAndProducts` for that purpose. To open the Profiler graphical user interface, use the `profile viewer` syntax. By default, Profiler time is CPU time. The total time reported by the Profiler is not the same as the time reported using the `tic` and `toc` functions or the time you would observe using a stopwatch.

Note: If your system uses Intel multi-core chips, you may want to restrict the active number of CPUs to 1 for the most accurate and efficient profiling. See "Multi-Core Processors — Setting for Most Accurate Profiling on Windows Systems" or "Multi-Core

Processors — Setting for Most Accurate Profiling on Linux Systems" for details on how to do this.

`profile on` starts the Profiler, clearing previously recorded profile statistics. Note the following:

- You can specify all, none, or a subset, of the `–history`, `–historysize` and `–timer` options with the `profile on` syntax.
- You can specify options in any order, including before or after `on`.
- If the Profiler is currently on and you specify `profile` with one of the options, MATLAB software returns an error message and the option has no effect. For example, if you specify `profile –timer real`, MATLAB returns the following error: `The profiler has already been started. TIMER cannot be changed.`
- To change options, first specify `profile off`, and then specify `profile on` or `profile resume` with new options.

`profile -history` records the exact sequence of function calls. The `profile` function records, by default, up to 1,000,000 function entry and exit events. For more than 1,000,000 events, `profile` continues to record other profile statistics, but not the sequence of calls. To change the number of function entry and exit events that the `profile` function records, use the `–historysize` option. By default, the `history` option is not enabled.

`profile -nohistory` disables further recording of the history (exact sequence of function calls). Use the `-nohistory` option after having previously set the `-history` option. All other profiling statistics continue to be collected.

`profile -history -historysize` *`integer`* specifies the number of function entry and exit events to record. By default, `historysize` is set to 1,000,000.

`profile -timer` *`clock`* specifies the type of time to use. Valid values for *`clock`* are:

- `'cpu'` — The Profiler uses computer time (the default).
- `'real'` — The Profiler uses wall-clock time.

For example, `cpu` time for the `pause` function is typically small, but `real` time accounts for the actual time paused, and therefore would be larger.

`profile -history -historysize integer -timer clock` specifies all of the options. Any order is acceptable, as is a subset.

`profile off` stops the Profiler.

`profile resume` restarts the Profiler without clearing previously recorded statistics.

`profile clear` clears the statistics recorded by `profile`.

`profile viewer` stops the Profiler and displays the results in the Profiler window. For more information, see Profiling for Improving Performance in the Desktop Tools and Development Environment documentation.

`S = profile('status')` returns a structure containing information about the current status of the Profiler. The table lists the fields in the order that they appear in the structure.

Field	Values	Default Value
`ProfilerStatus`	`'on'` or `'off'`	off
`DetailLevel`	`'mmex'`	'mmex'
`Timer`	`'cpu'` or `'real'`	'cpu'
`HistoryTracking`	`'on'` or `'off'`	'off'
`HistorySize`	integer	1000000

`stats = profile('info')` stops the Profiler and displays a structure containing the results. Use this function to access the data generated by `profile`. The table lists the fields in the order that they appear in the structure.

Field	Description
`FunctionTable`	Structure array containing statistics about each function called
`FunctionHistory`	Array containing function call history
`ClockPrecision`	Precision of the `profile` function's time measurement
`ClockSpeed`	Estimated clock speed of the CPU
`Name`	Name of the profiler

The `FunctionTable` field is an array of structures, where each structure contains information about one of the functions or local functions called during execution. The following table lists these fields in the order that they appear in the structure.

Field	Description
`CompleteName`	Full path to `FunctionName`, including local functions
`FunctionName`	Function name; includes local functions
`FileName`	Full path to `FunctionName`, with file extension, excluding local functions
`Type`	MATLAB functions, MEX-functions, and many other types of functions including MATLAB local functions, nested functions, and anonymous functions
`NumCalls`	Number of times the function was called
`TotalTime`	Total time spent in the function and its child functions
`TotalRecursiveTime`	No longer used.
`Children`	`FunctionTable` indices to child functions
`Parents`	`FunctionTable` indices to parent functions
`ExecutedLines`	Array containing line-by-line details for the function being profiled. Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix. Column 2: Number of times the line was executed Column 3: Total time spent on that line. Note: The sum of Column 3 entries does not necessarily add up to the function's `TotalTime`.
`IsRecursive`	`BOOLEAN` value: Logical `1` (true) if recursive, otherwise logical `0` (false)
`PartialData`	`BOOLEAN` value: Logical `1` (true) if function was modified during profiling, for example by being edited or cleared. In that event, data was collected only up until the point when the function was modified.

Examples

Profile, View Results, and Save Profile Data as HTML

This example profiles the MATLAB `magic` command and then displays the results in the Profiler window. The example then retrieves the profile data on which the HTML display is based and uses the `profsave` command to save the profile data in HTML form.

```
profile on
plot(magic(35))
profile viewer
p = profile('info');
profsave(p,'profile_results')
```

Profile, Save Profile Data to a MAT-File, and View Results

Another way to save profile data is to store it in a MAT-file. This example stores the profile data in a MAT-file, clears the profile data from memory, and then loads the profile data from the MAT-file. This example also shows a way to bring the reloaded profile data into the Profiler graphical interface as live profile data, not as a static HTML page.

```
p = profile('info');
save myprofiledata p
clear p
load myprofiledata
profview(0,p)
```

Profile and Show Results Including History

This example illustrates an effective way to view the results of profiling when the `history` option is enabled. The history data describes the sequence of functions entered and exited during execution. The `profile` command returns history data in the `FunctionHistory` field of the structure it returns. The history data is a 2-by-`n` array. The first row contains Boolean values, where `0` means entrance into a function and `1` means exit from a function. The second row identifies the function being entered or exited by its index in the `FunctionTable` field. This example reads the history data and displays it in the MATLAB Command Window.

```
profile on -history
plot(magic(4));
p = profile('info');
```

```
for n = 1:size(p.FunctionHistory,2)
 if p.FunctionHistory(1,n)==0
        str = 'entering function: ';
 else
        str = 'exiting function: ';
 end
 disp([str p.FunctionTable(p.FunctionHistory(2,n)).FunctionName])
end
```

See Also

`matlab.codetools.requiredFilesAndProducts` | `mlint` | `profsave`

Introduced before R2006a

profsave

Save profile report in HTML format

Syntax

```
profsave
profsave(profinfo)
profsave(profinfo,dirname)
```

Description

`profsave` executes the `profile('info')` function and saves the results in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of the structure returned by `profile`. By default, `profsave` stores the HTML files in a subfolder of the current folder named `profile_results`.

`profsave(profinfo)` saves the profiling results, `profinfo`, in HTML format. `profinfo` is a structure of profiling information returned by the `profile('info')` function.

`profsave(profinfo,dirname)` saves the profiling results, `profinfo`, in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of `profinfo` and stores them in the folder specified by `dirname`.

Examples

Run `profile` and save the results.

```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

More About

- Profiling for Improving Performance

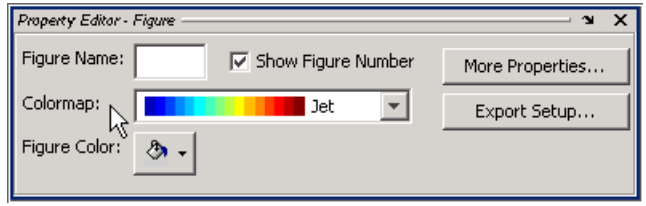
See Also

`profile`

Introduced before R2006a

propedit

Open Property Editor



Syntax

```
propedit
propedit(handle_list)
```

Description

`propedit` starts the Property Editor, a graphical user interface to the properties of graphics objects. If no current figure exists, `propedit` will create one.

`propedit(handle_list)` edits the properties for the object (or objects) in `handle_list`.

Starting the Property Editor enables plot editing mode for the figure.

See Also

`inspect` | `plotedit` | `propertyeditor`

Introduced before R2006a

propedit (COM)

Open built-in property page for control

Syntax

```
propedit(h)
```

Description

`propedit(h)` requests the control to display its built-in property page. Note that some controls do not have a built-in property page. For those controls, this command fails.

COM functions are available on Microsoft Windows systems only.

See Also

`get (COM)` | `inspect`

Introduced before R2006a

properties

Class property names

Syntax

```
properties('classname')
properties(obj)
p = properties(...)
```

Description

`properties('classname')` displays the names of the public properties for the MATLAB class named by `classname`. The `properties` function also displays inherited properties.

`properties(obj)` `obj` can be either a scalar object or an array of objects. When `obj` is scalar, `properties` also returns dynamic properties. See “Dynamic Properties — Adding Properties to an Instance” for information on using dynamic properties.

`p = properties(...)` returns the property names in a cell array of strings.

Definitions

A property is public when its `GetAccess` attribute value is `public` and its `Hidden` attribute value is `false` (default values for these attributes). See “Property Attributes” for a complete list of attributes.

`properties` is also a MATLAB class-definition keyword. See `classdef` for more information on class definition keywords.

Examples

Retrieve the names of the public properties of class `memmapfile` and store the result in a cell array of strings:

```
p = properties('memmapfile');
p
ans =

    'writable'
    'offset'
    'format'
    'repeat'
    'filename'
```

Construct an instance of the `MException` class and get its properties names:

```
me = MException('Msg:ID','MsgText');
properties(me)
Properties for class MException:

    identifier
    message
    cause
    stack
```

See Also

`fieldnames` | `events` | `methods`

propertyeditor

Show or hide **Property Editor**

Syntax

```
propertyeditor('on')
propertyeditor('off')
propertyeditor
propertyeditor(figure_handle,...)
```

Description

`propertyeditor('on')` displays the Property Editor tool on the current figure.

`propertyeditor('off')` hides the Property Editor on the current figure.

`propertyeditor` toggles the visibility of the Property Editor on the current figure. You can also use `propertyeditor('toggle')`instead for the same functionality.

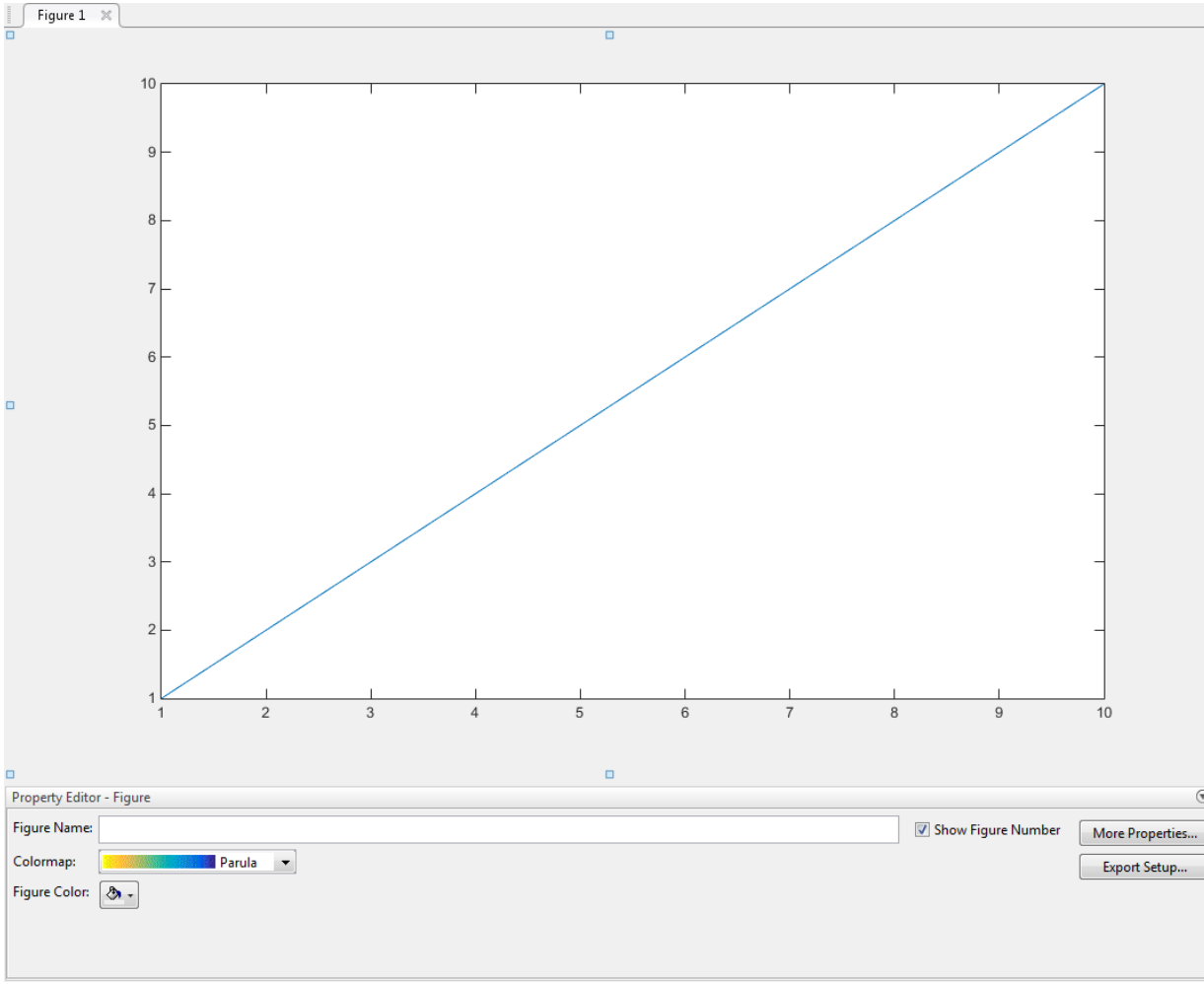
`propertyeditor(figure_handle,...)` displays or hides the Property Editor on the figure specified by `figure_handle`.

Examples

Open Property Editor

Create a simple plot and open the property editor.

```
plot(1:10)
propertyeditor
```



Alternatives

To collectively enable Plotting Tools, use the large Plotting Tool icon  on the figure toolbar. To collectively disable the Plotting Tools, use the smaller icon  . Open or close

the **Property Editor** tool from the figure's **View** menu. For details, see “Customize Objects in Graph”.

More About

Tips

If you call `propertyeditor` in a MATLAB program and subsequent lines depend on the Property Editor being fully initialized, follow it by `drawnow` to ensure complete initialization.

See Also

`plottools` | `plotbrowser` | `figurepalette` | `inspect`

Introduced before R2006a

matlab.mixin.util.PropertyGroup class

Package: matlab.mixin.util

Custom property list for object display

Description

Use the `PropertyGroup` class to create custom property display lists for class derived from `matlab.mixin.CustomDisplay`.

Construction

`P = matlab.mixin.util.PropertyGroup(propertyList)` constructs a property group with the supplied propertyList.

`P = matlab.mixin.util.PropertyGroup(propertyList,title)` displays title above the list of properties.

Input Arguments

`propertyList`

The propertyList is either a cell array of property names, or a scalar `struct` with property name-value pairs.

`title`

Text to display above properties.

Properties

`NumProperties`

The number of properties in the `PropertyList`.

Attributes:

Dependent	true
GetAccess	public
GetObservable	true
SetAccess	private
Transient	true

PropertyList

The list of properties to display, stored as a scalar struct or a cell array of strings.

Attributes:

GetAccess	public
GetObservable	true
SetAccess	public
SetObservable	true

Title

An optional Title for the PropertyGroup.

Attributes:

GetAccess	public
GetObservable	true
SetAccess	public
SetObservable	true

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

matlab.mixin.CustomDisplay

Related Examples

- “Custom Display Interface”
- “Customize Property Display”

More About

- Property Attributes

Introduced in R2013b

psi

Psi (polygamma) function

Syntax

```
Y = psi(X)
Y = psi(k,X)
```

Description

`Y = psi(X)` evaluates the ψ function for each element of array `X`. `X` must be real and nonnegative. The ψ function, also known as the digamma function, is the logarithmic derivative of the gamma function

$$\begin{aligned}\psi(x) &= \text{digamma}(x) \\ &= \frac{d(\log(\Gamma(x)))}{dx} \\ &= \frac{d(\Gamma(x))/dx}{\Gamma(x)}\end{aligned}$$

`Y = psi(k,X)` evaluates the `k`th derivative of ψ at the elements of `X`. `psi(0,X)` is the digamma function, `psi(1,X)` is the trigamma function, `psi(2,X)` is the tetragamma function, etc.

Examples

Example 1

Use the `psi` function to calculate Euler's constant, γ.

```
format long
-psi(1)
ans =
   0.57721566490153
```

```
-psi(0,1)
ans =
   0.57721566490153
```

Example 2

The trigamma function of 2, `psi(1,2)`, is the same as $(\pi^2/6) - 1$.

```
format long
psi(1,2)
ans =
   0.64493406684823

pi^2/6 - 1
ans =
   0.64493406684823
```

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Sections 6.3 and 6.4.

See Also

`gamma` | `gammainc` | `gammaln`

Introduced before R2006a

publish

Generate view of MATLAB file in specified format

Syntax

```
publish(file)
publish(file,format)

publish(file,Name,Value)
publish(file,options)

my_doc = publish(file, ___ )
```

Description

`publish(file)` generates a view of a MATLAB file in HTML format for sharing your code. For example, `publish('myfile.m')` executes the code in `myfile.m` using the base workspace and saves the formatted code and results in `/html/myfile.html`. The `html` subfolder is relative to the `file` folder.

When MATLAB publishes a file, it can delete existing files from the output folder that start with the same name as `file`.

`publish(file,format)` generates a view of a MATLAB file in the specified file format. The resulting files save to the `html` subfolder for all file formats.

`publish(file,Name,Value)` generates a view of the MATLAB file, `file`, with options specified by one or more name-value pair arguments.

`publish(file,options)` uses the options structure to customize the output, which is useful when you want to preconfigure and save your options for repeated use. The options structure fields and values correspond to names and values of name-value pair arguments, respectively.

`my_doc = publish(file, ___ )` generates a view of the MATLAB file `file`, and returns a string indicating the path of the resulting output file. It can include any of the input arguments in previous syntaxes.

Examples

Generate HTML View of MATLAB Script

Generate an HTML view of the code, MATLAB results, and comments in a MATLAB script.

Copy the example file, `fourier_demo2.m`, to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Generate an HTML view of the MATLAB file.

```
 publish('fourier_demo2.m');
```

The `publish` command executes the code for each cell in `fourier_demo2.m`, and saves the file to `/html/fourier_demo2.html`.

View the HTML file.

```
web('html/fourier_demo2.html')
```

Generate View of MATLAB Script in Microsoft Word Format

Generate a Microsoft Word view of the code, MATLAB results, and comments in a MATLAB script.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Publish the file in Microsoft Word format.

```
publish('fourier_demo2.m','doc');
```

View the published output.

```
winopen('html/fourier_demo2.doc')
```

Publish MATLAB Script Using Name-Value Pairs to Customize Output

Generate an HTML view of the code, MATLAB results, and comments in a MATLAB script. Use name-value pair arguments to include window decorations in the published output.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Publish the example MATLAB file to HTML format.

```
publish('fourier_demo2.m', 'figureSnapMethod', 'entireFigureWindow')
```

The `'figureSnapMethod','entireFigureWindow'` name-value pair argument specifies to include the window decorations in the figures, and to match the figure background color to the screen color.

View the published output.

```
web('html/fourier_demo2.html')
```

Customize `publish` Output Using Options Structure

Generate a Microsoft Word view of the code, MATLAB results, and comments in a MATLAB script. Use a structure to customize the published output.

Specifying options as a structure is useful when you want to preconfigure and save your options for repeated use.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Define options to customize the published output as a structure, `options_doc_nocode`.

```
options_doc_nocode.format = 'doc';
options_doc_nocode.showCode = false;
```

Publish the file, specifying the options structure.

```
publish('fourier_demo2.m',options_doc_nocode);
```

Save File Path of Published Script to Variable

Generate an HTML view of a MATLAB script, and save the path of the published HTML file to a variable.

This example assumes that the current folder is `C:\my_MATLAB_files`.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Publish the MATLAB file, and save the path of the resulting published HTML file to an output variable.

```
mydoc = publish('fourier_demo2.m')
```

```
mydoc =

C:\my_MATLAB_files\html\fourier_demo2.html
```

Input Arguments

`file` — MATLAB file

string

Full or partial path of the MATLAB file for which you want to generate a presentation view, specified as a string.

Example: `'myfile.m'`

`format` — Output format of published file

`'html'` (default) | `'doc'` | `'latex'` | `'ppt'` | `'xml'` | `'pdf'`

Output format of published MATLAB file, specified as one of the following string values.

Output Format	String Value
Hypertext Markup Language	`'html'` (default)
Microsoft Word	`'doc'`
LaTeX	`'latex'`
Microsoft PowerPoint	`'ppt'`
Extensible Markup Language	`'xml'`
Portable Document Format	`'pdf'`

Example: `publish('myfile.m','ppt');`

`options` — Options for published output

MATLAB structure

Options for published output, specified as a structure. Use the options structure instead of name-value pair arguments when you want to reuse the same configuration for publishing multiple MATLAB code files.

The options structure field and values correspond to names and values of the name-value pair arguments, respectively.

For example, to specify the PDF output format and the output folder `C:\myPublishedOutput`, use:

```
options = struct('format','pdf','outputDir','C:\myPublishedOutput')
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'format','latex','showCode',false` specifies LaTeX output file format and excludes the code from the output.

Output Options

`'format'` — Published output file format

`'html'` (default) | `'doc'` | `'latex'` | `'ppt'` | `'xml'` | `'pdf'`

Published output file format, specified as the comma-separated pair consisting of `'format'` and one of the following string values.

Output Format	String Value
Hypertext Markup Language	`'html'` (default)
Microsoft Word	`'doc'`
LaTeX	`'latex'`

Output Format	String Value
Microsoft PowerPoint	`'ppt'`
Extensible Markup Language	`'xml'`
Portable Document Format	`'pdf'`

`'outputDir'` — Output folder

`''` (default) | full path

Output folder to which the published document is saved, specified as the comma-separated pair consisting of `'outputDir'` and the full path. You must specify the full path as a string, for example `'C:\myPublishedOutput'`.

The default value, `''`, specifies the `html` subfolder of the current folder.

`'stylesheet'` — Extensible Stylesheet language (XSL) file

`''` (default) | full path to XSL file name

Extensible Stylesheet Language (XSL) file to use when publishing MATLAB code to HTML, XML, or LaTeX format, specified as the comma-separated pair consisting of `'stylesheet'` and the full path to the XSL file. The full path must be a string, for example, `'C:\myStylesheet\stylesheet.xsl'`

The default value, `''`, specifies the MATLAB default style sheet.

Figure Options

`'createThumbnail'` — Thumbnail image creation

`true` (default) | `false`

Thumbnail image creation for the published document, specified as the comma-separated pair consisting of `'createThumbnail'` and either `true` or `false`. You can use this thumbnail to represent your file in HTML pages.

`'figureSnapMethod'` — Published figure window appearance

`'entireGUIWindow'` (default) | `'print'` | `'getframe'` | `'entireFigureWindow'`

Appearance of published figure windows, including window decorations and background color, specified as the comma-separated pair consisting of `'figureSnapMethod'` and one of the following string values.

String Value	Window Decorations		Background Color	
	GUIs	Figures	GUIs	Figures
'entireGUIWindow' (default)	Included	Excluded	Matches screen	White
'print'	Excluded	Excluded	White	White
'getframe'	Excluded	Excluded	Matches screen	Matches screen
'entireFigureWindow'	Included	Included	Matches screen	Matches screen

'imageFormat' — Published image file format

'png' | 'epsc2' | 'jpg' | ...

Published image file format, specified as the comma-separated pair consisting of 'imageFormat' and one of the following string values.

'format' Value	Valid 'imageFormat' Value	Default 'imageFormat' Value
'doc'	Any image format that your installed version of Microsoft Office can import, including 'png' , 'jpg', 'bmp', and 'tiff'. If the 'figureSnapMethod' is 'print', then you can also specify 'eps', 'epsc', 'eps2', 'ill', 'meta', and 'pdf'.	'png'
'html'	Any format publishes successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.	'png'
'latex'	Any format publishes successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.	'epsc2', except when 'figureSnapMethod' is any one of the following: • 'getframe' • 'entireFigureWindow'

		• `'entireGUIWindow'` and the snapped window is a GUI window In these cases the default is `'png'`
`'pdf'`	`'bmp'` or `'jpg'`.	`'bmp'`
`'ppt'`	Any format that your installed version of Microsoft Office can import, including `'png'`, `'jpg'`, `'bmp'`, and `'tiff'`.	`'png'`
`'xml'`	Any format publishes successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.	`'png'`

`'maxHeight'` — Maximum height of published images

`[]` (default) | positive integer value

Maximum height of published images that the code generates, specified as the comma-separated pair consisting of `'maxHeight'` and one of the following values:

- `[]` (default)—Unrestricted height. This value is always used when the `'format'` value is `'pdf'`.
- Positive integer value that specifies the image height in pixels.

`'maxWidth'` — Maximum width of published images

`''` (default) | positive integer value

Maximum width of an image that the code generates, specified as the comma-separated pair consisting of `'maxWidth'` and one of the following values:

- `[]` (default)—Unrestricted width. This value is always used when the `'format'` value is `'pdf'`.
- Positive integer value that specifies the image width in pixels.

`'useNewFigure'` — Create new figure

`true` (default) | `false`

A logical value that determines if MATLAB creates a new Figure window for figures that the code generates, specified as the comma-separated pair consisting of `'useNewFigure'` and one of the following:

- `true` (default)—If the code generates a figure, then MATLAB creates a Figure window with a white background, and at the default size before publishing.
- `false`—MATLAB does not create a figure window.

  This value enables you to use a figure with different properties for publishing. For example, open a Figure window, change the size and background color, and then publish your code. Figures in your published document use the characteristics of the figure you opened before publishing.

Code Options

`'evalCode'` — Option to run code

`true` (default) | `false`

Option to evaluate code and include the MATLAB output in the published view, specified as a logical value.

`'catchError'` — Error handling during publishing

`true` (default) | `false`

Error handling during publishing, specified as the comma-separated pair consisting of `'catchError'` and one of the following logical values:

- `true` (default)—MATLAB continues publishing and includes the error in the published file.
- `false`—MATLAB displays the error at the command line and does not produce a published file.

`'codeToEvaluate'` — Additional code to evaluate during publishing

string

Additional code to evaluate during publishing, specified as the comma-separated pair consisting of `'codeToEvaluate'` and the string with the corresponding code. Use this option to specify code that does not appear in the MATLAB file, for example to set the value of an input argument for a function being published.

If this option is unspecified, MATLAB evaluates only the code in the MATLAB file you are publishing.

`'maxOutputLines'` — Maximum number of lines

`Inf` (default) | nonnegative integer value

Maximum number of lines in MATLAB output per cell evaluated during publishing, specified as the comma-separated pair consisting of `'maxOutputLines'` and one of the following values:

- `Inf` (default)—MATLAB includes all output lines in the published output.
- Nonnegative integer—MATLAB truncates the number of lines in the output at the number of lines you specify.

`'showCode'` — Option to include code in published file

`true` (default) | `false`

Option to include code in published file, specified as the comma-separated pair consisting of `'showCode'` and a logical value.

More About

Window Decorations

Window decorations include window title bar, toolbar, menu bar, and window border.

Syntax Highlighting

Syntax highlighting colors various elements in code to help you identify these elements while reading or editing code. By default, keywords are blue, strings are purple, and unterminated strings are maroon.

MATLAB does not preserve syntax highlighting when you set `'format'` to `'latex'` or `'ppt'`.

Tips

- If `'format` is `'html'`, MATLAB includes the code being published at the end of the published HTML file as comments, even when you set the `'showCode'` option to `false`. Because MATLAB includes the code as comments, this code does not display

in a Web browser. This enables you to use the `grabcode` function to extract the MATLAB code from an HTML file, even when the file does not display the code.

- "Publishing MATLAB Code"
- "Publishing Markup"

See Also

`grabcode` | `notebook`

Introduced before R2006a

PutCharArray

Store character array in Automation server

Syntax

IDL Method Signature

```
PutCharArray([in] BSTR varname, [in] BSTR workspace,
    [in] BSTR string)
```

Microsoft Visual Basic Client

```
PutCharArray(varname As String, workspace As String,
    string As String)
```

MATLAB Client

```
PutCharArray(h,'varname','workspace','string')
```

Description

`PutCharArray(h,'varname','workspace','string')` stores the character array in `string` in the specified `workspace` of the server attached to handle `h`, assigning to it the variable `varname`. The values for *`workspace`* are `base` or `global`. The function name is case-sensitive.

Examples

Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab As Object
```

```
Try
    Matlab = GetObject(, "matlab.application")
Catch e As Exception
    Matlab = CreateObject("matlab.application")
End Try
MsgBox("MATLAB window created; now open it...")
```

Open the MATLAB window, then click **Ok**.

```
Matlab.PutCharArray("str", "base", _
       "He jests at scars that never felt a wound.")
MsgBox("In MATLAB, type" & vbCrLf _
  & "str")
```

In the MATLAB window type `str`; MATLAB displays:

```
str =
He jests at scars that never felt a wound.
```

Click **Ok**.

```
MsgBox("closing MATLAB window...")
```

Click **Ok** to close and terminate MATLAB.

```
Matlab.Quit()
```

More About

Tips

The character array specified in the `string` argument can have any dimensions. However, `PutCharArray` changes the dimensions to a 1–by-n column-wise representation, where n is the number of characters in the array. Executing the following commands in MATLAB illustrates this behavior:

```
h = actxserver('matlab.application');
chArr = ['abc'; 'def'; 'ghk']
chArr =
abc
def
ghk
```

```
PutCharArray(h,'Foo','base',chArr)
tstArr = GetCharArray(h,'Foo','base')
tstArr =
adgbehcfk
```

See Also

GetCharArray | PutWorkspaceData | GetWorkspaceData | Execute

Introduced before R2006a

PutFullMatrix

Matrix in Automation server workspace

Syntax

IDL Method Signature

```
PutFullMatrix([in] BSTR varname, [in] BSTR workspace,
[in] SAFEARRAY(double) xreal, [in] SAFEARRAY(double) ximag)
```

Microsoft Visual Basic Client

```
PutFullMatrix([in] varname As String, [in] workspace As String,
[in] xreal As Double, [in] ximag As Double)
```

MATLAB Client

```
PutFullMatrix(h,'varname','workspace',xreal,ximag)
```

Description

`PutFullMatrix(h,'varname','workspace',xreal,ximag)` stores a matrix in the specified *`workspace`* of the server attached to handle `h` and assigns it to variable `varname`. Use `xreal` and `ximag` for the real and imaginary parts of the matrix. The matrix cannot be a scalar, an empty array, or have more than two dimensions. The values for *`workspace`* are `base` or `global`.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of `safearray`, which VBScript does not support.

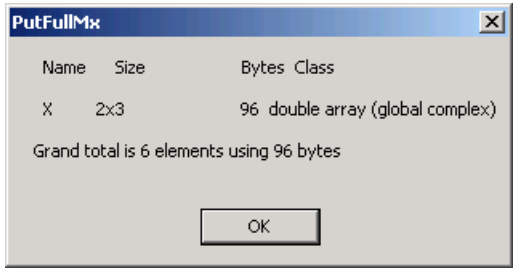
Examples

This example uses a Visual Basic .NET client to write a matrix to the base workspace of the server:

```
Dim MatLab As Object
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim ZReal(4, 4) As Double
Dim ZImag(4, 4) As Double
Dim i, j As Integer

For i = 0 To 4
   For j = 0 To 4
   XReal(i, j) = Rnd() * 6
   XImag(i, j) = 0
   Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("M","base",XReal,XImag)
MatLab.GetFullMatrix("M","base",ZReal,ZImag)
```

Use a Visual Basic .NET client to write a matrix to the global workspace of the server:

```
Dim MatLab As Object
Dim XReal(1,2) As Double
Dim XImag(1,2) As Double
Dim result As String
Dim i,j As Integer

For i = 0 To 1
 For j = 0 To 2
  XReal(i,j) = (j * 2 + 1) + i
  XImag(i,j) = 1
 Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("X","global",XReal,XImag)
result = Matlab.Execute("whos global")
MsgBox(result)
```



See Also

`GetFullMatrix` | `PutWorkspaceData` | `Execute`

Introduced before R2006a

PutWorkspaceData

Data in Automation server workspace

Syntax

IDL Method Signature

```
PutWorkspaceData([in] BSTR varname, [in] BSTR workspace,
[in] VARIANT data)
```

Microsoft Visual Basic Client

```
PutWorkspaceData(varname As String, workspace As String,
data As Object)
```

MATLAB Client

```
PutWorkspaceData(h,'varname','workspace',data)
```

Description

`PutWorkspaceData(h,'varname','workspace',data)` stores `data` in the *`workspace`* of the server attached to handle `h` and assigns it to `varname`. The values for *`workspace`* are `base` or `global`.

Use `PutWorkspaceData` to pass numeric and character array data respectively to the server. Do *not* use `PutWorkspaceData` on sparse arrays, structures, or function handles. Use the `Execute` method for these data types.

The `GetWorkspaceData` and `PutWorkspaceData` functions pass numeric data as a `variant` data type. These functions are especially useful for VBScript clients as VBScript does not support the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`.

Examples

Create an array in a Visual Basic .NET client and put it in the base workspace of the MATLAB Automation server:

1. Create the Visual Basic application. Use the `MsgBox` command to control flow between MATLAB and the application:

```
Dim Matlab As Object
Dim data(6) As Double
Dim i As Integer
MatLab = CreateObject("matlab.application")
For i = 0 To 6
   data(i)  = i * 15
Next i
MatLab.PutWorkspaceData("A","base",data)
MsgBox("In MATLAB, type" & vbCrLf & "A")
```

2. Open the MATLAB window and type `A`. MATLAB displays:

```
A =
     0    15    30    45    60    75    90
```

3. Click **Ok** to close and terminate MATLAB.

See Also

`GetWorkspaceData` | `PutFullMatrix` | `PutCharArray` | `Execute`

Introduced before R2006a

pwd

Identify current folder

Syntax

```
pwd
currentFolder = pwd
```

Description

`pwd` displays the MATLAB current folder.

`currentFolder = pwd` returns the current folder as a string to `currentFolder`.

Alternatives

- Use the Current Folder toolbar.

See Also

`cd` | `dir`

Introduced before R2006a

pyargs

Create keyword argument for Python function

Syntax

```
kwa = pyargs(argKey,argValue)
```

Description

`kwa = pyargs(argKey,argValue)` creates a `pyargs` keyword argument to pass to a Python function.

Examples

Use Keyword Argument to Modify the Display of Calendar Month

Create a calendar.

```
cal = py.calendar.TextCalendar;
```

Display calendar for December, 2014.

```
formatmonth(cal,int32(2014),int32(12))
```

```
ans =

  Python str with no properties.

       December 2014
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Read the function signature from the Python documentation for the `calendar.TextCalendar.formatmonth` function.

```
py.help('calendar.TextCalendar.formatmonth')
```

```
Help on method formatmonth in calendar.TextCalendar:

calendar.TextCalendar.formatmonth = formatmonth(self, theyear, themonth, w=0, l=0) unbo
    Return a month's calendar string (multi-line).
```

Arguments `w` and `l` are optional, with default values of `0`.

Read the function signature in MATLAB.

```
methods(cal,'-full')
```

```
lhs formatmonth(self, theyear, themonth, w, l, pyargs)
```

Search the output for the `formatmonth` function signature. You can use the `pyargs` argument instead of the optional `w` and `l` arguments.

Change the line spacing, `l`, using a `pyargs` argument.

```
formatmonth(cal,int32(2014),int32(12),pyargs('l',int32(2)))
```

```
ans =

  Python str with no properties.

       December 2014

Mo Tu We Th Fr Sa Su

 1  2  3  4  5  6  7

 8  9 10 11 12 13 14

15 16 17 18 19 20 21

22 23 24 25 26 27 28

29 30 31
```

Input Arguments

`argKey,argValue` — Python function arguments

keyword and value arguments

Python function keyword arguments specified as one or more comma-separated pairs of argKey,argValue arguments. `argKey` is the Python function key name and is a string. `argValue` is the argument value, represented by any type. Use the function argument list to identify `argKey` and `argValue`. You can specify several key and value pair arguments in any order as `argKey1,argValue1,...,argKeyN,argValueN`.

Example: `'length',int32(2)`

More About

- python.org calendar.TextCalendar method

Introduced in R2014b

matlab.exception.PyException class

Package: matlab.exception

Capture error information for Python exception

Description

Process information from a `matlab.exception.PyException` object to handle Python errors thrown from Python methods called from MATLAB. This class is derived from `MException`.

Construction

`e = matlab.exception.PyException(msgID,errMsg,excObj)` constructs instance e of `matlab.exception.PyException` class.

Input Arguments

`msgID` — Message identifier
string

Message identifier, specified as a string.

`errMsg` — Error message
string

Error message, specified as a string.

`excObj` — Exception object
`pythonclass`

Exception object, specified as the `pythonclass` that caused the exception.

Output Arguments

`e` — Exception object
exception object

Instance of `matlab.exception.PyException` class

Properties

`ExceptionObject` — Object

exception object

Python exception object that caused the error.

Tips

- Typically, you do not construct a `matlab.exception.PyException` object explicitly. MATLAB automatically constructs a `PyException` object whenever Python throws an exception. The `PyException` object wraps the original Python exception.

Examples

Catch Python Exception

Generate a Python exception and display information.

```
try
  py.list('x','y',1)
catch e
  e.message
  if(isa(e,''))
    e.ExceptionObject
  end
end
```

```
ans =

Python Error: list() takes at most 1 argument (3 given)
```

When MATLAB displays a message containing the text `Python Error`, refer to your Python documentation for more information.

More About

- " Capture Information About Exceptions"

- “Throw an Exception”

External Web Sites

- www.python.org/doc

Introduced in R2014b

pyversion

Change default version of Python interpreter

Syntax

```
pyversion
[version, executable, isloaded] = pyversion

___ = pyversion version
___ = pyversion executable
```

Description

`pyversion` displays details about the current Python version.

`[version, executable, isloaded] = pyversion` returns Python version information.

`___ = pyversion version` changes the default Python version on Microsoft Windows platforms. The setting is persistent across MATLAB sessions.

You cannot change the version after MATLAB loads Python. To change the version if Python is loaded already, restart MATLAB, and then call `pyversion`.

`___ = pyversion executable` specifies full path to Python executable. Use on any platform or for repackaged CPython implementation downloads.

Examples

Display Python Version

```
pyversion
```

```
       version: '2.7'
    executable: 'C:\Python27\python.exe'
       library: 'C:\windows\system32\python27.dll'
```

```
           home: 'C:\Python27'
      isloaded: 0
```

Use Python Version 2.7

```
[v, e, isloaded] = pyversion;
if isloaded
    disp('To change the Python version, restart MATLAB, then call pyversion.')
else
    pyversion 2.7
end
```

Input Arguments

`version` — Python version number

string

Python version number, specified as a string. Version must contain the major and minor version numbers separated by a period. Windows platform only.

`pyversion` looks for the version in the Windows registry. If you download from `www.python.org/downloads`, the installation automatically adds the version to the registry. If you download the Python application from a different source, either add it to the registry, or use the `pyversion(executable)` syntax to change the version.

Example: `3.3`

`executable` — Name of existing Python executable file

string

Name of an existing Python executable file, specified as a string. This argument must contain the name of the Python executable file and it can contain the full path.

Example: `C:\Python33\python.exe`

Output Arguments

`version` — Python version number

string

Python version number, specified as a string.

`executable` — Name of Python executable file

string

Name of Python executable file, specified as a string.

`isloaded` — Indicates if specified version is loaded

logical

Indicates if specified version is loaded, specified as logical. MATLAB loads Python when you type a `py.` command.

If MATLAB cannot load Python, isloaded is `0` and MATLAB displays “Undefined variable "py" or function "py.command"” when you type a `py.` command.

Limitations

- Do not use `pyversion` to set the version in a MATLAB function containing Python commands. MATLAB loads the Python interpreter when it processes a Python command in the function before it executes the `pyversion` function.

More About

- https://www.python.org/downloads

Introduced in R2014b

qmr

Quasi-minimal residual method

Syntax

```
x = qmr(A,b)
qmr(A,b,tol)
qmr(A,b,tol,maxit)
qmr(A,b,tol,maxit,M)
qmr(A,b,tol,maxit,M1,M2)
qmr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = qmr(A,b,...)
[x,flag,relres] = qmr(A,b,...)
[x,flag,relres,iter] = qmr(A,b,...)
[x,flag,relres,iter,resvec] = qmr(A,b,...)
```

Description

`x = qmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `qmr` converges, a message to that effect is displayed. If `qmr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`qmr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `qmr` uses the default, `1e-6`.

`qmr(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `qmr` uses the default, `min(n,20)`.

`qmr(A,b,tol,maxit,M)` and `qmr(A,b,tol,maxit,M1,M2)` use preconditioners `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `qmr` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `qmr` uses the default, an all zero vector.

`[x,flag] = qmr(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`qmr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`qmr` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	The method stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `qmr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = qmr(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = qmr(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = qmr(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Using qmr with a Matrix Input

This example shows how to use `qmr` with a matrix input. The code:

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8; maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = qmr(A,b,tol,maxit,M1,M2);
```

displays the message:

```
qmr converged at iteration 9 to a solution...
with relative residual
5.6e-009
```

Using qmr with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_qmr` that

- Calls `qmr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_qmr` are available to `afun`.

The following shows the code for `run_qmr`:

```
function x1 = run_qmr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = qmr(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
```

```
        elseif strcmp(transp_flag,'notransp') % y = A*x
           y = 4 * x;
           y(2:n) = y(2:n) - 2 * x(1:n-1);
           y(1:n-1) = y(1:n-1) - x(2:n);
        end
    end
end
```

When you enter

```
x1=run_qmr;
```

MATLAB software displays the message

```
qmr converged at iteration 9 to a solution with relative residual
5.6e-009
```

Using qmr with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

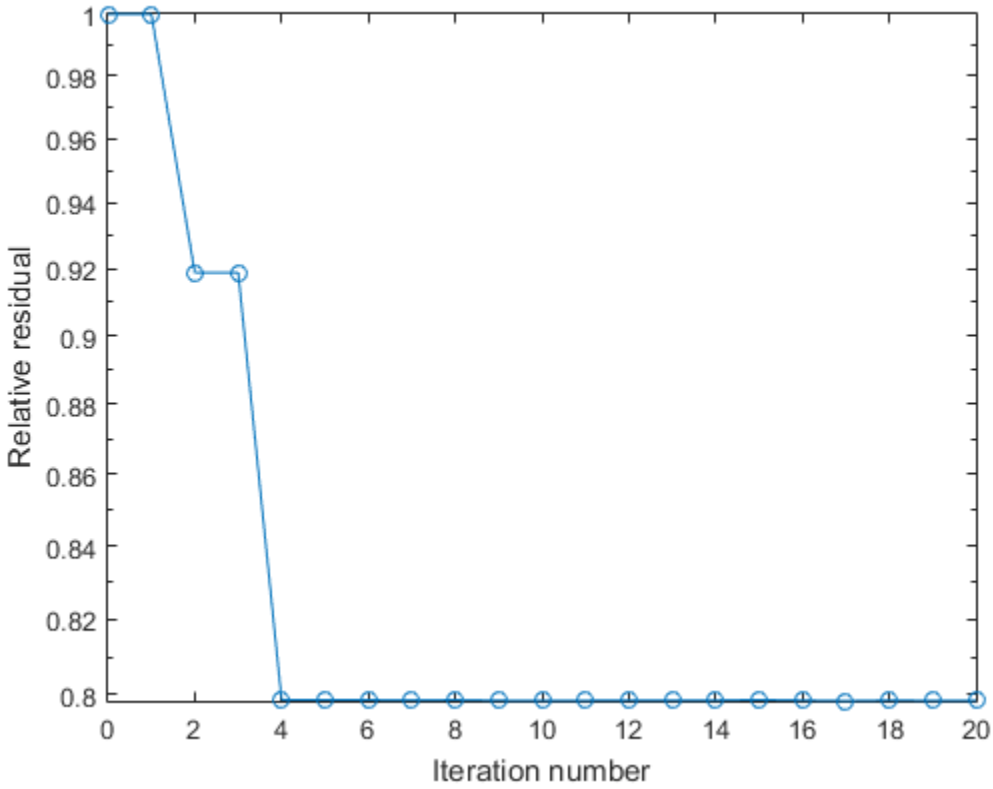
Use `qmr` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = qmr(A,b,tol,maxit);
```

`fl0` is 1 because `qmr` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The seventeenth iterate is the best approximate solution and is the one returned as indicated by `it0 = 17`. MATLAB stores the residual history in `rv0`.

Plot the behavior of `qmr`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
```

```
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

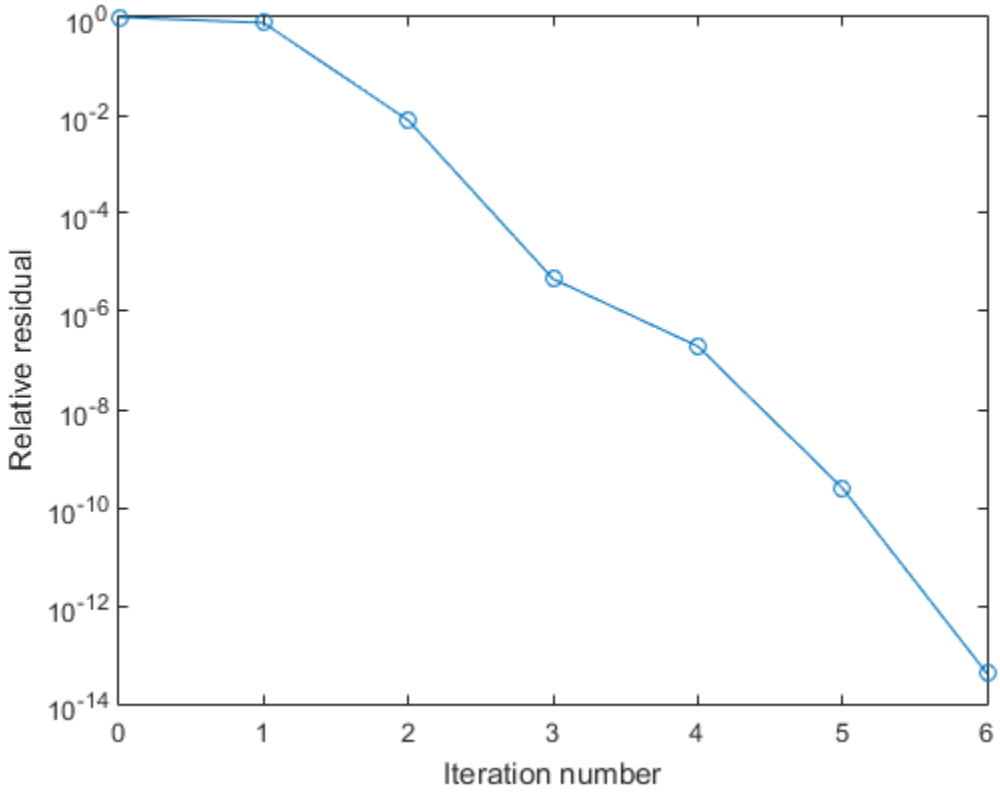
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = qmr(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `qmr` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

You can follow the progress of `qmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Freund, Roland W. and Nöel M. Nachtigal, "QMR: A quasi-minimal residual method for non-Hermitian linear systems," *SIAM Journal: Numer. Math.* 60, 1991, pp. 315–339.

See Also

`bicg` | `bicgstab` | `cgs` | `function_handle` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `symmlq`

Introduced before R2006a

qr

Orthogonal-triangular decomposition

Syntax

```
[Q,R] = qr(A)
[Q,R] = qr(A,0)
[Q,R,E] = qr(A)
[Q,R,E] = qr(A,'matrix')
[Q,R,e] = qr(A,'vector')
[Q,R,e] = qr(A,0)
X = qr(A)
X = qr(A,0)
R = qr(A)
R = qr(A,0)
[C,R] = qr(A,B)
[C,R,E] = qr(A,B)
[C,R,E] = qr(A,B,'matrix')
[C,R,e] = qr(A,B,'vector')
[C,R] = qr(A,B,0)
[C,R,e] = qr(A,B,0)
```

Description

`[Q,R] = qr(A)`, where `A` is `m`-by-`n`, produces an `m`-by-`n` upper triangular matrix `R` and an `m`-by-`m` unitary matrix `Q` so that `A = Q*R`.

`[Q,R] = qr(A,0)` produces the economy-size decomposition. If `m > n`, only the first `n` columns of `Q` and the first `n` rows of `R` are computed. If `m<=n`, this is the same as `[Q,R] = qr(A)`.

If A is full:

`[Q,R,E] = qr(A)` or `[Q,R,E] = qr(A,'matrix')` produces unitary `Q`, upper triangular `R` and a permutation matrix `E` so that `A*E = Q*R`. The column permutation `E` is chosen so that `abs(diag(R))` is decreasing.

`[Q,R,e] = qr(A,'vector')` returns the permutation information as a vector instead of a matrix. That is, `e` is a row vector such that `A(:,e) = Q*R`.

`[Q,R,e] = qr(A,0)` produces an economy-size decomposition in which `e` is a permutation vector, so that `A(:,e) = Q*R`.

`X = qr(A)` and `X = qr(A,0)` return a matrix `X` such that `triu(X)` is the upper triangular factor `R`.

If A is sparse:

`R = qr(A)` computes a `Q`-less `QR` decomposition and returns the upper triangular factor `R`. Note that `R = chol(A'*A)`. Since `Q` is often nearly full, this is preferred to `[Q,R] = QR(A)`.

`R = qr(A,0)` produces economy-size `R`. If `m>n`, `R` has only `n` rows. If `m<=n`, this is the same as `R = qr(A)`.

`[Q,R,E] = qr(A)` or `[Q,R,E] = qr(A,'matrix')` produces unitary `Q`, upper triangular `R` and a permutation matrix `E` so that `A*E = Q*R`. The column permutation `E` is chosen to reduce fill-in in R.

`[Q,R,e] = qr(A,'vector')` returns the permutation information as a vector instead of a matrix. That is, `e` is a row vector such that `A(:,e) = Q*R`.

`[Q,R,e] = qr(A,0)` produces an economy-size decomposition in which `e` is a permutation vector, so that `A(:,e) = Q*R`.

`[C,R] = qr(A,B)`, where `B` has as many rows as `A`, returns `C = Q'*B`. The least-squares solution to `A*X = B` is `X = R\C`.

`[C,R,E] = qr(A,B)` or `[C,R,E] = qr(A,B,'matrix')`, also returns a fill-reducing ordering. The least-squares solution to `A*X = B` is `X = E*(R\C)`.

`[C,R,e] = qr(A,B,'vector')` returns the permutation information as a vector instead of a matrix. That is, the least-squares solution to `A*X = B` is `X(e,:) = R\C`.

`[C,R] = qr(A,B,0)` produces economy-size results. If `m>n`, `C` and `R` have only `n` rows. If `m<=n`, this is the same as `[C,R] = qr(A,B)`.

`[C,R,e] = qr(A,B,0)` additionally produces a fill-reducing permutation vector `e`. In this case, the least-squares solution to `A*X = B` is `X(e,:) = R\C`.

Examples

Find the least squares approximate solution to `A*x = b` with the `Q`-less `QR` decomposition and one step of iterative refinement:

```
if issparse(A), R = qr(A);
else R = triu(qr(A)); end
x = R\(R'\(A'*b));
r = b - A*x;
err = R\(R'\(A'*r));
x = x + err;
```

See Also

`lu` | `chol` | `null` | `orth` | `qrdelete` | `qrinsert` | `qrupdate`

Introduced before R2006a

qrdelete

Remove column or row from QR factorization

Syntax

```
[Q1,R1] = qrdelete(Q,R,j)
[Q1,R1] = qrdelete(Q,R,j,'col')
[Q1,R1] = qrdelete(Q,R,j,'row')
```

Description

`[Q1,R1] = qrdelete(Q,R,j)` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the column `A(:,j)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

`[Q1,R1] = qrdelete(Q,R,j,'col')` is the same as `qrdelete(Q,R,j)`.

`[Q1,R1] = qrdelete(Q,R,j,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the row `A(j,:)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
[Q1,R1] = qrdelete(Q,R,j,'row');

Q1 =
    0.5274   -0.5197   -0.6697   -0.0578
    0.7135    0.6911    0.0158    0.1142
    0.3102   -0.1982    0.4675   -0.8037
    0.3413   -0.4616    0.5768    0.5811

R1 =
   32.2335   26.0908   19.9482   21.4063   23.3297
         0  -19.7045  -10.9891    0.4318   -1.4873
```

```
         0         0   22.7444    5.8357   -3.1977
         0         0         0  -14.5784    3.7796
```

returns a valid QR factorization, although possibly different from

```
A2 = A;
A2(j,:) = [];
[Q2,R2] = qr(A2)

Q2 =
   -0.5274    0.5197    0.6697   -0.0578
   -0.7135   -0.6911   -0.0158    0.1142
   -0.3102    0.1982   -0.4675   -0.8037
   -0.3413    0.4616   -0.5768    0.5811

R2 =
  -32.2335  -26.0908  -19.9482  -21.4063  -23.3297
         0   19.7045   10.9891   -0.4318    1.4873
         0         0  -22.7444   -5.8357    3.1977
         0         0         0  -14.5784    3.7796
```

More About

Algorithms

The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

See Also

`planerot` | `qr` | `qrinsert`

Introduced before R2006a

qrinsert

Insert column or row into QR factorization

Syntax

```
[Q1,R1] = qrinsert(Q,R,j,x)
[Q1,R1] = qrinsert(Q,R,j,x,'col')
[Q1,R1] = qrinsert(Q,R,j,x,'row')
```

Description

`[Q1,R1] = qrinsert(Q,R,j,x)` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with the column `x` inserted before `A(:,j)`. If `A` has `n` columns and `j = n+1`, then `x` is inserted after the last column of `A`.

`[Q1,R1] = qrinsert(Q,R,j,x,'col')` is the same as `qrinsert(Q,R,j,x)`.

`[Q1,R1] = qrinsert(Q,R,j,x,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with an extra row, `x`, inserted before `A(j,:)`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
x = 1:5;
[Q1,R1] = qrinsert(Q,R,j,x,'row')

Q1 =
    0.5231    0.5039   -0.6750    0.1205    0.0411    0.0225
    0.7078   -0.6966    0.0190   -0.0788    0.0833   -0.0150
    0.0308    0.0592    0.0656    0.1169    0.1527   -0.9769
    0.1231    0.1363    0.3542    0.6222    0.6398    0.2104
    0.3077    0.1902    0.4100    0.4161   -0.7264   -0.0150
    0.3385    0.4500    0.4961   -0.6366    0.1761    0.0225
```

```
R1 =
   32.4962   26.6801   21.4795   23.8182   26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0   24.4514   11.8132    3.9931
         0         0         0   20.2382   10.3392
         0         0         0         0   16.1948
         0         0         0         0         0
```

returns a valid QR factorization, although possibly different from

```
A2 = [A(1:j-1,:); x; A(j:end,:)];
[Q2,R2] = qr(A2)

Q2 =
   -0.5231    0.5039    0.6750   -0.1205    0.0411    0.0225
   -0.7078   -0.6966   -0.0190    0.0788    0.0833   -0.0150
   -0.0308    0.0592   -0.0656   -0.1169    0.1527   -0.9769
   -0.1231    0.1363   -0.3542   -0.6222    0.6398    0.2104
   -0.3077    0.1902   -0.4100   -0.4161   -0.7264   -0.0150
   -0.3385    0.4500   -0.4961    0.6366    0.1761    0.0225

R2 =
  -32.4962  -26.6801  -21.4795  -23.8182  -26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0  -24.4514  -11.8132   -3.9931
         0         0         0  -20.2382  -10.3392
         0         0         0         0   16.1948
         0         0         0         0         0
```

More About

Algorithms

The `qrinsert` function inserts the values of `x` into the `j`th column (row) of `R`. It then uses a series of Givens rotations to zero out the nonzero elements of `R` on and below the diagonal in the `j`th column (row).

See Also

`planerot` | `qr` | `qrdelete`

Introduced before R2006a

qrupdate

Rank 1 update to QR factorization

Syntax

```
[Q1,R1] = qrupdate(Q,R,u,v)
```

Description

`[Q1,R1] = qrupdate(Q,R,u,v)` when `[Q,R] = qr(A)` is the original QR factorization of `A`, returns the QR factorization of `A + u*v'`, where `u` and `v` are column vectors of appropriate lengths.

Examples

The matrix

```
mu = sqrt(eps)

mu =

   1.4901e-08

A = [ones(1,4); mu*eye(4)];
```

is a well-known example in least squares that indicates the dangers of forming `A'*A`. Instead, we work with the QR factorization – orthonormal Q and upper triangular R.

```
 [Q,R] = qr(A);
```

As we expect, `R` is upper triangular.

```
R =

   -1.0000   -1.0000   -1.0000   -1.0000
         0    0.0000    0.0000    0.0000
         0         0    0.0000    0.0000
```

```
         0         0         0    0.0000
         0         0         0         0
```

In this case, the upper triangular entries of `R`, excluding the first row, are on the order of `sqrt(eps)`.

Consider the update vectors

```
 u = [-1 0 0 0 0]'; v = ones(4,1);
```

Instead of computing the rather trivial QR factorization of this rank one update to `A` from scratch with

```
[QT,RT] = qr(A + u*v')

QT =

     0     0     0     0     1
    -1     0     0     0     0
     0    -1     0     0     0
     0     0    -1     0     0
     0     0     0    -1     0

RT =

  1.0e-007 *

   -0.1490         0         0         0
         0   -0.1490         0         0
         0         0   -0.1490         0
         0         0         0   -0.1490
         0         0         0         0
```

we may use `qrupdate`.

```
[Q1,R1] = qrupdate(Q,R,u,v)

Q1 =

   -0.0000   -0.0000   -0.0000   -0.0000    1.0000
    1.0000   -0.0000   -0.0000   -0.0000    0.0000
    0.0000    1.0000   -0.0000   -0.0000    0.0000
    0.0000    0.0000    1.0000   -0.0000    0.0000
   -0.0000   -0.0000   -0.0000    1.0000    0.0000
```

```
R1 =

   1.0e-007 *
    0.1490    0.0000    0.0000    0.0000
         0    0.1490    0.0000    0.0000
         0         0    0.1490    0.0000
         0         0         0    0.1490
         0         0         0         0
```

Note that both factorizations are correct, even though they are different.

More About

Tips

`qrupdate` works only for full matrices.

Algorithms

`qrupdate` uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. `qrupdate` is useful since, if we take `N = max(m,n)`, then computing the new QR factorization from scratch is roughly an $O(N^3)$ algorithm, while simply updating the existing factors in this way is an $O(N^2)$ algorithm.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

`cholupdate` | `qr`

Introduced before R2006a

quad

Numerically evaluate integral, adaptive Simpson quadrature

Compatibility

`quad` will be removed in a future release. Use `integral` instead.

Syntax

```
q = quad(fun,a,b)
q = quad(fun,a,b,tol)
q = quad(fun,a,b,tol,trace)
[q,fcnt] = quad(...)
```

Description

Quadrature is a numerical method used to find the area under the graph of a function, that is, to compute a definite integral.

$$q = \int_a^b f(x)dx$$

`q = quad(fun,a,b)` tries to approximate the integral of function `fun` from `a` to `b` to within an error of `1e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. Limits `a` and `b` must be finite. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`, the integrand evaluated at each element of `x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = quad(fun,a,b,tol)` uses an absolute error tolerance `tol` instead of the default which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results. In MATLAB version 5.3 and earlier, the `quad` function used a less reliable algorithm and a default relative tolerance of `1.0e-3`.

`q = quad(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q]` during the recursion.

`[q,fcnt] = quad(...)` returns the number of function evaluations.

The function `quadl` may be more efficient with high accuracies and smooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

To compute the integral

$$\int_0^2 \frac{1}{x^3 - 2x - 5} dx,$$

write a function `myfun` that computes the integrand:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Then pass `@myfun`, a function handle to `myfun`, to `quad`, along with the limits of integration, `0` to `2`:

```
Q = quad(@myfun,0,2)

Q =

   -0.4605
```

Alternatively, you can pass the integrand to `quad` as an anonymous function handle `F`:

```
F = @(x)1./(x.^3-2*x-5);
Q = quad(F,0,2);
```

Diagnostics

`quad` may issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

More About

Algorithms

`quad` implements a low order method using an adaptive recursive Simpson's rule.

- “Anonymous Functions”

References

[1] Gander, W. and W. Gautschi, “Adaptive Quadrature – Revisited,” BIT, Vol. 40, 2000, pp. 84-101. This document is also available at http://www.inf.ethz.ch/personal/gander.

See Also

`quad2d` | `dblquad` | `quadgk` | `quadl` | `quadv` | `trapz` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

Introduced before R2006a

quad2d

Numerically evaluate double integral, tiled method

Syntax

```
q = quad2d(fun,a,b,c,d)
[q,errbnd] = quad2d(...)
q = quad2d(fun,a,b,c,d,param1,val1,param2,val2,...)
```

Description

`q = quad2d(fun,a,b,c,d)` approximates the integral of `fun(x,y)` over the planar region $a \le x \le b$ and $c(x) \le y \le d(x)$. `fun` is a function handle, `c` and `d` may each be a scalar or a function handle.

All input functions must be vectorized. The function `Z=fun(X,Y)` must accept 2-D matrices `X` and `Y` of the same size and return a matrix `Z` of corresponding values. The functions `ymin=c(X)` and `ymax=d(X)` must accept matrices and return matrices of the same size with corresponding values.

`[q,errbnd] = quad2d(...)`. `errbnd` is an approximate upper bound on the absolute error, `|Q - I|`, where `I` denotes the exact value of the integral.

`q = quad2d(fun,a,b,c,d,param1,val1,param2,val2,...)` performs the integration as above with specified values of optional parameters:

`AbsTol`	absolute error tolerance
`RelTol`	relative error tolerance

`quad2d` attempts to satisfy `ERRBND <= max(AbsTol,RelTol*|Q|)`. This is absolute error control when `|Q|` is sufficiently small and relative error control when `|Q|` is larger. A default tolerance value is used when a tolerance is not specified. The default value of `AbsTol` is 1e-5. The default value of `RelTol` is `100*eps(class(Q))`. This is also the minimum value of `RelTol`. Smaller `RelTol` values are automatically increased to the default value.

`MaxFunEvals`	Maximum allowed number of evaluations of `fun` reached.

The `MaxFunEvals` parameter limits the number of vectorized calls to `fun`. The default is 2000.

`FailurePlot`	Generate a plot if `MaxFunEvals` is reached.

Setting `FailurePlot` to `true` generates a graphical representation of the regions needing further refinement when `MaxFunEvals` is reached. No plot is generated if the integration succeeds before reaching `MaxFunEvals`. These (generally) 4-sided regions are mapped to rectangles internally. Clusters of small regions indicate the areas of difficulty. The default is `false`.

`Singular`	Problem may have boundary singularities

With `Singular` set to `true`, `quad2d` will employ transformations to weaken boundary singularities for better performance. The default is `true`. Setting `Singular` to `false` will turn these transformations off, which may provide a performance benefit on some smooth problems.

Examples

Example 1

Integrate $y\sin(x)+x\cos(y)$ over $\pi \le x \le 2\pi$, $0 \le y \le \pi$. The true value of the integral is $-\pi^2$.

```
Q = quad2d(@(x,y) y.*sin(x)+x.*cos(y),pi,2*pi,0,pi)
```

Example 2

Integrate $[(x+y)^{1/2}(1+x+y)^2]^{-1}$ over the triangle $0 \le x \le 1$ and $0 \le y \le 1-x$. The integrand is infinite at (0,0). The true value of the integral is $\pi/4 - 1/2$.

```
fun = @(x,y) 1./(sqrt(x + y) .* (1 + x + y).^2 )
```

In Cartesian coordinates:

```
ymax = @(x) 1 - x;
Q = quad2d(fun,0,1,0,ymax)
```

In polar coordinates:

```
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
rmax = @(theta) 1./(sin(theta) + cos(theta));
Q = quad2d(polarfun,0,pi/2,0,rmax)
```

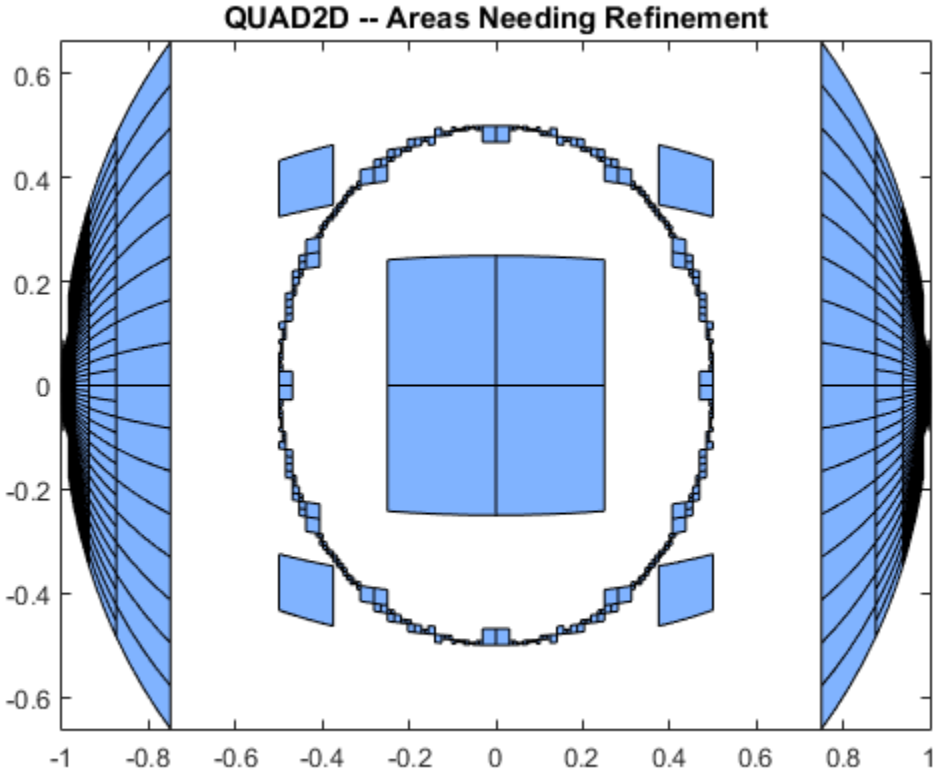
Limitations

`quad2d` begins by mapping the region of integration to a rectangle. Consequently, it may have trouble integrating over a region that does not have four sides or has a side that cannot be mapped smoothly to a straight line. If the integration is unsuccessful, some helpful tactics are leaving `Singular` set to its default value of `true`, changing between Cartesian and polar coordinates, or breaking the region of integration into pieces and adding the results of integration over the pieces.

For instance:

```
fun = @(x,y)abs(x.^2 + y.^2 - 0.25);
c = @(x)-sqrt(1 - x.^2);
d = @(x)sqrt(1 - x.^2);
quad2d(fun,-1,1,c,d,'AbsTol',1e-8,...
    'FailurePlot',true,'Singular',false);
```

```
Warning: Reached the maximum number of function evaluations (2000). The result
fails the global error test.
```

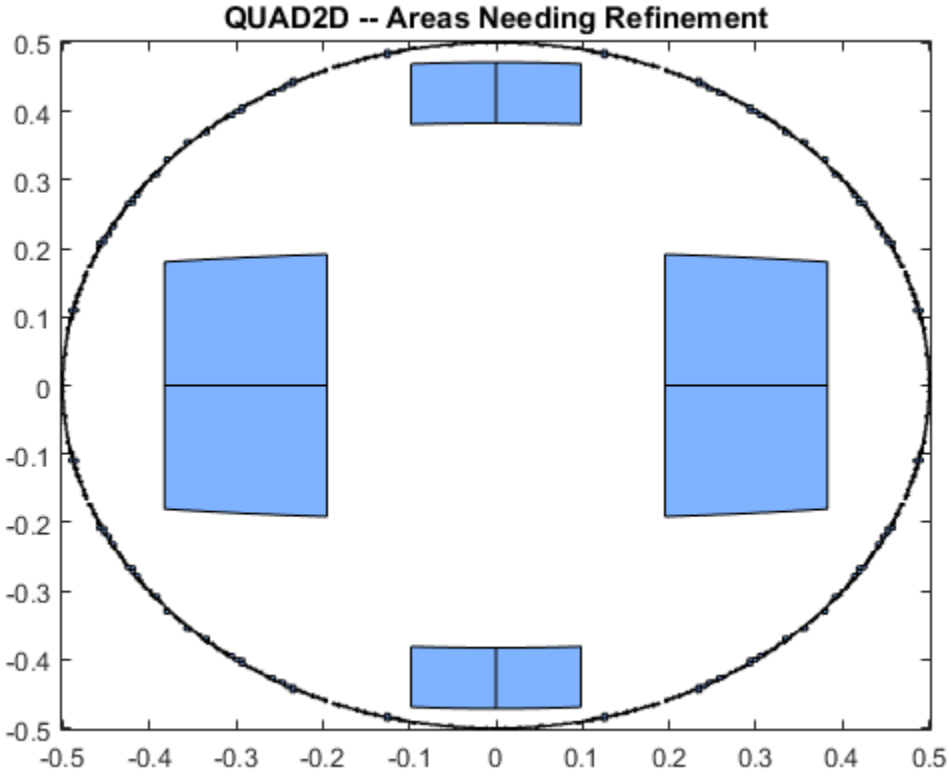


The failure plot shows two areas of difficulty, near the points `(-1,0)` and `(1,0)` and near the circle $x^2 + y^2 = 0.25$.

Changing the value of `Singular` to `true` will cope with the geometric singularities at `(-1,0)` and `(1,0)`. The larger shaded areas may need refinement but are probably not areas of difficulty.

```
Q = quad2d(fun,-1,1,c,d,'AbsTol',1e-8, ...
     'FailurePlot',true,'Singular',true);
```

```
Warning: Reached the maximum number of function evaluations (2000). The result
passes the global error test.
```



From here you can take advantage of symmetry:

```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-8,...
     'Singular',true,'FailurePlot',true)
```

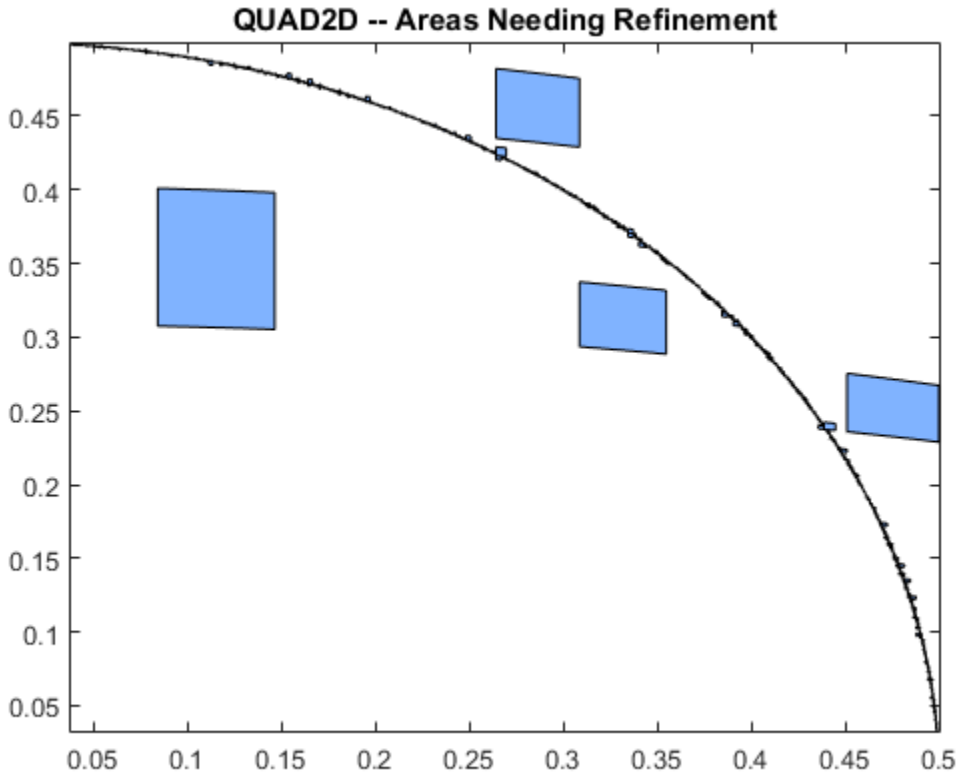
```
Q =

    0.9817
```

However, the code is still working very hard near the singularity. It may not be able to provide higher accuracy:

```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-10,...
```

```
    'Singular',true,'FailurePlot',true);
```

```
Warning: Reached the maximum number of function evaluations (2000). The result
passes the global error test.
```



At higher accuracy, a change in coordinates may work better.

```
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
Q = 4*quad2d(polarfun,0,pi/2,0,1,'AbsTol',1e-10);
```

It is best to put the singularity on the boundary by splitting the region of integration into two parts:

```
Q1 = 4*quad2d(polarfun,0,pi/2,0,0.5,'AbsTol',5e-11);
```

```
Q2 = 4*quad2d(polarfun,0,pi/2,0.5,1,'AbsTol',5e-11);
Q = Q1 + Q2;
```

More About

- “Anonymous Functions”

References

[1] L.F. Shampine, "Matlab Program for Quadrature in 2D." *Applied Mathematics and Computation*. Vol. 202, Issue 1, 2008, pp. 266–274.

See Also

`dblquad` | `quad` | `quadl` | `quadv` | `quadgk` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

quadgk

Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Syntax

```
q = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)
```

Description

`q = quadgk(fun,a,b)` attempts to approximate the integral of a scalar-valued function `fun` from `a` to `b` using high-order global adaptive quadrature and default error tolerances. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`, where `y` is the integrand evaluated at each element of `x`. `fun` must be a function handle. Limits `a` and `b` can be `-Inf` or `Inf`. If both are finite, they can be complex. If at least one is complex, the integral is approximated over a straight line path from `a` to `b` in the complex plane.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[q,errbnd] = quadgk(fun,a,b)` returns an approximate upper bound on the absolute error, `|Q - I|`, where `I` denotes the exact value of the integral.

`[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)` performs the integration with specified values of optional parameters. The available parameters are

Parameter	Description	
`'AbsTol'`	Absolute error tolerance. The default value of `'AbsTol'` is `1.e-10` (double), `1.e-5` (single).	`quadgk` attempts to satisfy `errbnd <= max(AbsTol,RelTol*|Q|)`. This is absolute error control when `|Q|` is sufficiently small and relative error control when `|Q|` is larger. For pure absolute error control use `'AbsTol'`
`'RelTol'`	Relative error tolerance.	

Parameter	Description	
	The default value of `'RelTol'` is `1.e-6` (double), `1.e-4` (single).	`> 0` and `'RelTol'= 0`. For pure relative error control use `'AbsTol' = 0`. Except when using pure absolute error control, the minimum relative tolerance is `'RelTol' >= 100*eps(class(Q))`.
`'Waypoints'`	Vector of integration waypoints.	If `fun(x)` has discontinuities in the interval of integration, the locations should be supplied as a `Waypoints` vector. When `a`, `b`, and the waypoints are all real, only the waypoints between `a` and `b` are used, and they are used in sorted order. Note that waypoints are not intended for singularities in `fun(x)`. Singular points should be handled by making them endpoints of separate integrations and adding the results. If `a`, `b`, or any entry of the waypoints vector is complex, the integration is performed over a sequence of straight line paths in the complex plane, from `a` to the first waypoint, from the first waypoint to the second, and so forth, and finally from the last waypoint to `b`.

Parameter	Description	
`'MaxIntervalCount'`	Maximum number of intervals allowed. The default value is `650`.	The `'MaxIntervalCount'` parameter limits the number of intervals that `quadgk` uses at any one time after the first iteration. A warning is issued if `quadgk` returns early because of this limit. Routinely increasing this value is not recommended, but it may be appropriate when `errbnd` is small enough that the desired accuracy has nearly been achieved.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

Integrand with a singularity at an integration end point

Write a function `myfun` that computes the integrand:

```
function y = myfun(x)
y = exp(x).*log(x);
```

Then pass `@myfun`, a function handle to `myfun`, to `quadgk`, along with the limits of integration, `0` to `1`:

```
q = quadgk(@myfun,0,1)

q =

   -1.3179
```

Alternatively, you can pass the integrand to `quadgk` as an anonymous function handle `F`:

```
f = (@(x)exp(x).*log(x));
q = quadgk(f,0,1);
```

Oscillatory integrand on a semi-infinite interval

Integrate over a semi-infinite interval with specified tolerances, and return the approximate error bound:

```
f = @(x)x.^5.*exp(-x).*sin(x);
[q,errbnd] = quadgk(f,0,inf,'RelTol',1e-8,'AbsTol',1e-12)

q =

  -15.0000

errbnd =

  9.4386e-009
```

Contour integration around a pole

Use `Waypoints` to integrate around a pole using a piecewise linear contour:

```
f = @(z)1./(2*z - 1);
```

```
q = quadgk(f,-1-i,-1-i,'Waypoints',[1-i,1+i,-1+i])

q =

   0.0000 + 3.1416i
```

Diagnostics

`quadgk` may issue one of the following warnings:

`'Minimum step size reached'` indicates that interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Reached the limit on the maximum number of intervals in use'` indicates that the integration was terminated before meeting the tolerance requirements and that continuing the integration would require more than `MaxIntervalCount` subintervals. The integral may not exist, or it may be difficult to approximate numerically. Increasing `MaxIntervalCount` usually does not help unless the tolerance requirements were nearly met when the integration was previously terminated.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

More About

Algorithms

`quadgk` implements adaptive quadrature based on a Gauss-Kronrod pair (15^{th} and 7^{th} order formulas).

- "Anonymous Functions"

References

[1] L.F. Shampine "*Vectorized Adaptive Quadrature in MATLAB*," *Journal of Computational and Applied Mathematics*, 211, 2008, pp.131–140.

See Also

quad2d | dblquad | quad | quadl | quadv | triplequad | function_handle | integral | integral2 | integral3

quadl

Numerically evaluate integral, adaptive Lobatto quadrature

Compatibility

`quadl` will be removed in a future release. Use `integral` instead.

Syntax

```
q = quadl(fun,a,b)
q = quadl(fun,a,b,tol)
quadl(fun,a,b,tol,trace)
[q,fcnt] = quadl(...)
```

Description

`q = quadl(fun,a,b)` approximates the integral of function `fun` from `a` to `b`, to within an error of 10^{-6} using recursive adaptive Lobatto quadrature. `fun` is a function handle. It accepts a vector `x` and returns a vector `y`, the function `fun` evaluated at each element of `x`. Limits `a` and `b` must be finite.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = quadl(fun,a,b,tol)` uses an absolute error tolerance of `tol` instead of the default, which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results.

`quadl(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a q]` during the recursion.

`[q,fcnt] = quadl(...)` returns the number of function evaluations.

Use array operators `.*`, `./` and `.^` in the definition of `fun` so that it can be evaluated with a vector argument.

The function quad might be more efficient with low accuracies or nonsmooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The quad function might be most efficient for low accuracies with nonsmooth integrands.
- The quadl function might be more efficient than quad at higher accuracies with smooth integrands.
- The quadgk function might be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, [a,Inf), then for the integral of fun(x) to exist, fun(x) must decay as x approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if fun(x) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint c like log|x-c| or $|x-c|^p$ for p >= -1/2. If the function is singular at points inside (a,b), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

Examples

Pass the function handle, @myfun, to quadl:

```
Q = quadl(@myfun,0,2);
```

where the function myfun.m is:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Pass anonymous function handle F to quadl:

```
F = @(x) 1./(x.^3-2*x-5);
Q = quadl(F,0,2);
```

Diagnostics

`quadl` might issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

More About

Algorithms

`quadl` implements a high order method using an adaptive Gauss/Lobatto quadrature rule.

- "Anonymous Functions"

References

[1] Gander, W. and W. Gautschi, "Adaptive Quadrature – Revisited," BIT, Vol.40, 2000, pp. 84-101. This document is also available at http://www.inf.ethz.ch/personal/gander.

See Also

`quad2d` | `dblquad` | `quad` | `quadgk` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

Introduced before R2006a

quadv

Vectorized quadrature

Compatibility

`quadv` will be removed in a future release. Use `integral` with the `'ArrayValued'` option instead.

Syntax

```
Q = quadv(fun,a,b)
Q = quadv(fun,a,b,tol)
Q = quadv(fun,a,b,tol,trace)
[Q,fcnt] = quadv(...)
```

Description

`Q = quadv(fun,a,b)` approximates the integral of the complex array-valued function `fun` from `a` to `b` to within an error of `1.e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. The function `Y = fun(x)` should accept a scalar argument `x` and return an array result `Y`, whose components are the integrands evaluated at `x`. Limits `a` and `b` must be finite.

"Parameterizing Functions" explains how to provide addition parameters to the function `fun`, if necessary.

`Q = quadv(fun,a,b,tol)` uses the absolute error tolerance `tol` for all the integrals instead of the default, which is `1.e-6`.

Note The same tolerance is used for all components, so the results obtained with `quadv` are usually not the same as those obtained with `quad` on the individual components.

`Q = quadv(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q(1)]` during the recursion.

`[Q,fcnt] = quadv(...)` returns the number of function evaluations.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function might be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function might be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function might be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

For the parameterized array-valued function `myarrayfun`, defined by

```
function Y = myarrayfun(x,n)
Y = 1./((1:n)+x);
```

the following command integrates `myarrayfun`, for the parameter value n = 10 between a = 0 and b = 1:

```
Qv = quadv(@(x)myarrayfun(x,10),0,1);
```

The resulting array `Qv` has 10 elements estimating `Q(k) = log((k+1)./(k))`, for `k = 1:10`.

The entries in `Qv` are slightly different than if you compute the integrals using `quad` in a loop:

```
for k = 1:10
 Qs(k) = quadv(@(x)myscalarfun(x,k),0,1);
end
```

where `myscalarfun` is:

```
function y = myscalarfun(x,k)
y = 1./(k+x);
```

See Also

`quad` | `quad2d` | `quadgk` | `quadl` | `dblquad` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

Introduced before R2006a

matlab.unittest.qualifications

Summary of classes in MATLAB Qualifications Interface

Description

Qualifications are methods for testing values and responding to failures. Qualification failures might or might not correspond to a test failure, and they might or might not continue execution in the test when one is encountered. To determine which qualification to use, see “Types of Qualifications”.

matlab.unittest.qualifications.Assertable	Qualification to validate preconditions of a test
matlab.unittest.qualifications.Assumable	Qualification to filter test content
matlab.unittest.qualifications.FatalAssertable	Qualification to abort test execution
matlab.unittest.qualifications.Verifiable	Qualification to produce soft-failure conditions

The package contains the following event data classes:

matlab.unittest.qualifications.ExceptionEventData	Event data for ExceptionThrown event listeners
matlab.unittest.qualifications.QualificationEventData	Event data for qualification event listeners

The package contains the following exception handling classes:

matlab.unittest.qualifications.AssertionFailedException	Exception used for assertion failures

matlab.unittest.qualifications.AssumptionFailedException
Exception used for assumption failures

matlab.unittest.qualifications.FatalAssertionFailedException
Exception used for fatal assertion failures

matlab.unittest.qualifications.Assertable class

Package: matlab.unittest.qualifications

Qualification to validate preconditions of a test

Description

The `Assertable` class provides a qualification to validate preconditions of a test. Apart from actions performed for failures, the `Assertable` class works the same as other `matlab.unittest` qualifications.

Upon an assertion failure, the `Assertable` class throws an `AssertionFailedException` to inform the testing framework of the failure. This is most useful when a failure at the assertion point renders the rest of the current test method invalid, yet does not prevent proper execution of other test methods. Often, you use assertions to ensure that preconditions of the current test are not violated or that fixtures are set up correctly. Make sure the test content is “Exception Safe” on page 1-6212. If you cannot make the fixture teardown exception safe or if you cannot recover it after failure, use fatal assertions instead.

Use assertions to allow remaining test methods to receive coverage when preconditions are violated in a test and all fixture states are restorable. Assertions also reduce the noise level of failures by not performing later verifications that fail due the precondition failures. In the event of a failure, however, the test framework marks the full content of the test method that failed as incomplete. Therefore, if the failure does not affect the preconditions of the test or cause problems with fixture setup or teardown, use verifications, which give the added information that the full test content was run.

Methods

assertClass	Assert exact class of specified value
assertEmpty	Assert value is empty
assertEqual	Assert value is equal to specified value

assertError	Assert function throws specified exception
assertFail	Produce unconditional assertion failure
assertFalse	Assert value is false
assertGreaterThan	Assert value is greater than specified value
assertGreaterThanOrEqual	Assert value is greater than or equal to specified value
assertInstanceOf	Assert value is object of specified type
assertLength	Assert value has specified length
assertLessThan	Assert value is less than specified value
assertLessThanOrEqual	Assert value is less than or equal to specified value
assertMatches	Assert string matches specified regular expression
assertNotEmpty	Assert value is not empty
assertNotEqual	Assert value is not equal to specified value
assertNotSameHandle	Assert value is not handle to specified instance
assertNumElements	Assert value has specified element count
assertReturnsTrue	Assert function returns true when evaluated

assertSameHandle	Assert two values are handles to same instance
assertSize	Assert value has specified size
assertSubstring	Assert string contains specified string
assertThat	Assert that value meets specified constraint
assertTrue	Assert value is true
assertWarning	Assert function issues specified warning
assertWarningFree	Assert function issues no warnings

Events

`AssertionFailed`	Triggered upon failing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssertionPassed`	Triggered upon passing assertion. A `QualificationEventData` object is passed to listener callback functions.

Definitions

Exception Safe

Test content is *exception safe* when all fixture teardown is performed with `addTeardown` or through the appropriate object destructors when a failure occurs. This ensures that the failure does not affect later testing due to stale fixtures.

This code is not exception safe. After an assertion failure, the test framework does not close the figure.

```
% Not exception safe
f = figure;
testCase.assertEqual(actual, expected);
close(f);
```

This code is exception safe because the test framework closes the figure in all cases.

```
% Exception safe
f = figure;
testCase.addTeardown(@close, f);
testCase.assertEqual(actual, expected);
```

However, tearing down a fixture using `addTeardown` does not guarantee code is exception safe. This code shows a failure in `assertEqual`.

```
% Not exception safe
f = figure;
testCase.assertEqual(actual, expected);
testCase.addTeardown(@close, f);
```

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Test for Preconditions Using Assertions

Use assertable qualifications to test for preconditions. This example will create a test case to write a polynomial to a MAT-file.

Create `DocPolynomSaveLoadTest` Test Case. Refer to the following `DocPolynomSaveLoadTest` test case in the subsequent steps in this example. The steps highlight specific code in the `testSaveLoad` function; the code statements are not intended to be executed outside the context of the class definition file.

DocPolynomSaveLoadTest Class Definition File

```
classdef DocPolynomSaveLoadTest < matlab.unittest.TestCase

    methods (TestClassSetup)
```

```
        function addDocPolynomClassToPath(testCase)
            origPath = path;
            testCase.addTeardown(@path, origPath);
            addpath(fullfile(matlabroot, ...
                'help', 'techdoc', 'matlab_oop', 'examples'));
        end
    end

    methods (Test)
        function testSaveLoad(testCase)

            import matlab.unittest.diagnostics.Diagnostic;

            %% Phase 1: Setup
            % Create a temporary working folder
            tempFolder = tempname;
            [success, message] = mkdir(tempFolder);
            testCase.assertTrue(success, ...
                Diagnostic.join('Could not create temporary folder.',...
                message));
            testCase.addTeardown(@() testCase.cleanUpTemporaryFolder(...
                tempFolder));

            % Change to the temporay folder and register the
            % teardown, which restores the original folder
            origFolder = pwd;
            testCase.addTeardown(@cd, origFolder);
            cd(tempFolder);

            %% Phase 2: Exercise
            % Save the instance to a mat file.
            p = DocPolynom([1, 0, 1]);
            save('DocPolynomFile', 'p');

            % Validate Precondition. Save resulted in valid .mat file
            testCase.assertEqual(exist('DocPolynomFile.mat','file'),...
                2, Diagnostic.join(...
                'mat file was not saved correctly.',@() dir(pwd)));

            loaded = load('DocPolynomFile');


            %% Phase 3: Verify
            testCase.verifyEqual(loaded.p, p,...
```

```
                'Loaded polynom did not equal original polynom.');


            %% Phase 4: Teardown
            % Done inline via calls to addTeardown at the points
            % at which the state was changed.


        end
    end

    methods(Access=private)
        function cleanUpTemporaryFolder(testCase,tempFolder)
            % Clean up the temporary folder and fatally assert
            % that it was correctly cleaned up.

            import matlab.unittest.diagnostics.Diagnostic;

            [success, message] = rmdir(tempFolder, 's');
            testCase.fatalAssertTrue(success, ...
                Diagnostic.join('Could not remove temporary folder.',...
                message));
        end
    end

end
```

To execute the MATLAB commands in "Run DocPolynomSaveLoadTest Test Case", add the `DocPolynomSaveLoadTest.m` file to a folder on your MATLAB path.

The `testSaveLoad` function consists of the following phases:

- Phase 1: Setup — Create and verify precondition code.
- Phase 2: Exercise — Create a `DocPolynom` object and save it to a MAT-file.
- Phase 3: Verify — Test that object was successfully saved.
- Phase 4: Teardown — Execute teardown code.

Define phase 1 precondition. For this test, use a temporary folder for creating a `DocPolynom` object. The precondition for continuing with this test is that the following commands execute successfully.

```
tempFolder = tempname;
[success, message] = mkdir(tempFolder);
```

Test the results of the `mkdir` function. Use the `assertTrue` method to test the `mkdir` success argument for errors. If an assertion occurs, the remainder of the `testSaveLoad` test method is invalid, and the test is marked `Incomplete`.

```
testCase.assertTrue(success, ...
    Diagnostic.join('Could not create the temporary folder.', ...
        message))
```

If the `mkdir` function fails, MATLAB displays the diagnostic message, `Could not create the temporary folder`, as well as the contents of the `mkdir` message argument.

Add teardown fixture code. Creating a temporary folder is setup code, which requires a corresponding call to the `rmdir` function to restore MATLAB to the original state. Use the `addTeardown` method to ensure the teardown code executes even when an exception is thrown in the middle of the test method. This makes the test Exception Safe.

```
testCase.addTeardown(@() testCase.cleanUpTemporaryFolder(tempFolder))
```

Place teardown code in the helper function. Although the `addTeardown` statement occurs in the same code block as the `mkdir` setup statement, the `cleanUpTemporaryFolder` code is executed in phase 4 of the test method.

In the `DocPolynomSaveLoadTest` test case, the helper function, `cleanUpTemporaryFolder`, executes the `rmdir` function.

Define the precondition for creating valid MAT-File. A precondition for verifying that the `DocPolynom` object was correctly saved and loaded is that the MAT-file, `DocPolynomFile.mat`, was successfully created. The following code in the `Phase 2: Exercise` block tests this condition. If an assertion occurs, the remainder of the `testSaveLoad` test method is invalid, and the test is marked `Failed` and `Incomplete`.

```
 testCase.assertEqual(exist('DocPolynomFile.mat','file'), 2, ...
     Diagnostic.join('The mat file was not saved correctly.', ...
     @() dir(pwd)))
```

If the file was not created, MATLAB displays the diagnostic message, `The mat file was not saved correctly`, as well as the contents of the temporary folder.

Run `DocPolynomSaveLoadTest` Test Case.

```
tc = DocPolynomSaveLoadTest;
run(tc);
```

```
Running DocPolynomSaveLoadTest
.
Done DocPolynomSaveLoadTest
__________
```

See Also

`Assumable` | `FatalAssertable` | `matlab.unittest.qualifications` | `QualificationEventData` | `TestCase` | `Verifiable`

More About

- "Types of Qualifications"

Introduced in R2013a

assertClass

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert exact class of specified value

Syntax

```
assertClass(assertable,actual,className)
assertClass(assertable,actual,metaClass)
assertClass( ___ ,diagnostic)
```

Description

`assertClass(assertable,actual,className)` asserts that actual is a MATLAB value whose class is the class specified by className.

`assertClass(assertable,actual,metaClass)` asserts that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass. The instance must be an exact class match. See `assertInstanceOf` to assert inclusion in a class hierarchy.

`assertClass( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  assertable.assertThat(actual, IsOfClass(className));
  assertable.assertThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via `assertThat`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyClass`, and replace calls to `verifyClass` with `assertClass`.

See Also

`assertInstanceOf` | `assertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assertEmpty

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is empty

Syntax

```
assertEmpty(assertable,actual)
assertEmpty(assertable,actual,diagnostic)
```

Description

`assertEmpty(assertable,actual)` asserts that actual is an empty MATLAB value.

`assertEmpty(assertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assertable.assertThat(actual, IsEmpty());
  ```

  This method is a convenience method. There exists more functionality when using the `IsEmpty` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyEmpty`, and replace calls to `verifyEmpty` with `assertEmpty`.

See Also

`assertNotEmpty` | `assertThat` | `isempty`

More About

- "Types of Qualifications"

Introduced in R2013a

assertEqual

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is equal to specified value

Syntax

```
assertEqual(assertable,actual,expected)
assertEqual( ___ ,Name,Value)
assertEqual( ___ ,diagnostic)
```

Description

`assertEqual(assertable,actual,expected)` asserts that actual is strictly equal to expected.

`assertEqual( ___ ,Name,Value)` asserts equality with additional options specified by one or more Name,Value pair arguments.

`assertEqual( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
assertable.assertThat(actual, IsEqualTo(expected));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
assertable.assertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.RelativeTolerance;
assertable.assertThat(actual, IsEqualTo(expected, ...
```

```
    'Within', RelativeTolerance(reltol)));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
assertable.assertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via `assertThat`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'AbsTol'`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`'RelTol'`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

See examples for `verifyEqual`, and replace calls to `verifyEqual` with `assertEqual`.

See Also

`assertNotEqual` | `assertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assertError

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert function throws specified exception

Syntax

```
assertError(assertable,actual,identifier)
assertError(assertable,actual,metaClass)
assertError( ___ ,diagnostic)
```

Description

`assertError(assertable,actual,identifier)` asserts that actual is a function handle that throws an exception with an error identifier that is equal to identifier.

`assertError(assertable,actual,metaClass)` asserts that actual is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in metaClass. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class.

`assertError( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  assertable.assertThat(actual, Throws(identifier));
  assertable.assertThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `assertThat`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`identifier`

Error identifier, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyError`, and replace calls to `verifyError` with `assertError`.

See Also

`MException` | `assertThat` | `assertWarning` | `error`

More About

- "Types of Qualifications"

Introduced in R2013a

assertFail

Produce unconditional assertion failure

Syntax

```
assertFail(assertable)
assertFail(assertable,diagnostic)
```

Description

`assertFail(assertable)` produces an unconditional assertion failure when encountered.

`assertFail(assertable,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Tips

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Examples

See examples for `verifyFail`, and replace calls to `verifyFail` with `assertFail`.

More About

- "Types of Qualifications"

Introduced in R2013a

assertFalse

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is false

Syntax

```
assertFalse(assertable,actual)
assertFalse(assertable,actual,diagnostic)
```

Description

`assertFalse(assertable,actual)` asserts that actual is a scalar logical with the value of false.

`assertFalse(assertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  assertable.assertThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `assertThat`.
- Unlike `assertTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `assertTrue`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

For examples, see `verifyFalse`, and replace calls to `verifyFalse` with `assertFalse`.

See Also

`assertThat` | `assertTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

assertGreaterThan

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is greater than specified value

Syntax

```
assertGreaterThan(assertable,actual,floor)
assertGreaterThan(assertable,actual,floor,diagnostic)
```

Description

`assertGreaterThan(assertable,actual,floor)` asserts that all elements of actual are greater than all the elements of floor.

`assertGreaterThan(assertable,actual,floor,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThan;
  assertable.assertThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThan`, and replace calls to `verifyGreaterThan` with `assertGreaterThan`.

See Also

`matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic` | `assertGreaterThanOrEqual` | `assertLessThan` | `assertLessThanOrEqual` | `assertThat` | `gt`

More About

- "Types of Qualifications"

Introduced in R2013a

assertGreaterThanOrEqual

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is greater than or equal to specified value

Syntax

```
assertGreaterThanOrEqual(assertable,actual,floor)
assertGreaterThanOrEqual(assertable,actual,floor,diagnostic)
```

Description

`assertGreaterThanOrEqual(assertable,actual,floor)` asserts that all elements of actual are greater than or equal to all the elements of floor.

`assertGreaterThanOrEqual(assertable,actual,floor,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  assertable.assertThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThanOrEqual`, and replace calls to `verifyGreaterThanOrEqual` with `assertGreaterThanOrEqual`.

See Also

`matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `assertGreaterThan` | `assertLessThan` | `assertLessThanOrEqual` | `assertThat` | `ge`

More About

- "Types of Qualifications"

Introduced in R2013a

assertInstanceOf

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is object of specified type

Syntax

```
assertInstanceOf(assertable,actual,className)
assertInstanceOf(assertable,actual,metaClass)
assertInstanceOf( ___ ,diagnostic)
```

Description

`assertInstanceOf(assertable,actual,className)` asserts that actual is a MATLAB value whose class is the class specified by className.

`assertInstanceOf(assertable,actual,metaClass)` asserts that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass.

`assertInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  assertable.assertThat(actual, IsInstanceOf(className));
  assertable.assertThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test

methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyInstanceOf`, and replace calls to `verifyInstanceOf` with `assertInstanceOf`.

See Also

`assertClass` | `assertThat` | `isa`

More About

- "Types of Qualifications"

Introduced in R2013a

assertLength

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value has specified length

Syntax

```
assertLength(assertable,actual,expectedLength)
assertLength(assertable,actual,expectedLength,diagnostic)
```

Description

`assertLength(assertable,actual,expectedLength)` that actual is a MATLAB array whose length is expectedLength.

`assertLength(assertable,actual,expectedLength,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  assertable.assertThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLength`, and replace calls to `verifyLength` with `assertLength`.

See Also

`assertNumElements` | `assertSize` | `assertThat` | `length`

More About

- "Types of Qualifications"

Introduced in R2013a

assertLessThan

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is less than specified value

Syntax

```
assertLessThan(assertable,actual,ceiling)
assertLessThan(assertable,actual,ceiling,diagnostic)
```

Description

`assertLessThan(assertable,actual,ceiling)` asserts that all elements of actual are less than all the elements of ceiling.

`assertLessThan(assertable,actual,ceiling,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assertable.assertThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as ceiling unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThan`, and replace calls to LessThan with `assertLessThan`.

See Also

`matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic` | `assertGreaterThan` | `assertGreaterThanOrEqual` | `assertLessThanOrEqual` | `assertThat` | `lt`

More About

- "Types of Qualifications"

Introduced in R2013a

assertLessThanOrEqual

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is less than or equal to specified value

Syntax

```
assertLessThanOrEqual(assertable,actual,ceiling)
assertLessThanOrEqual(assertable,actual,ceiling,diagnostic)
```

Description

`assertLessThanOrEqual(assertable,actual,ceiling)` asserts that all elements of actual are less than or equal to all the elements of ceiling.

`assertLessThanOrEqual(assertable,actual,ceiling,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  assertable.assertThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as ceiling unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThanOrEqual`, and replace calls to `verifyLessThanOrEqual` with `assertLessThanOrEqual`.

See Also

`matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `assertGreaterThan` | `assertGreaterThanOrEqual` | `assertLessThan` | `assertThat` | `le`

More About

- "Types of Qualifications"

Introduced in R2013a

assertMatches

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert string matches specified regular expression

Syntax

```
assertMatches(assertable,actual,expression)
assertMatches(assertable,actual,expression,diagnostic)
```

Description

`assertMatches(assertable,actual,expression)` asserts that actual is a string that matches the regular expression defined by expression.

`assertMatches(assertable,actual,expression,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  assertable.assertThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyMatches`, and replace calls to `verifyMatches` with `assertMatches`.

See Also

`assertSubstring` | `assertThat` | `regexp`

More About

- "Types of Qualifications"

Introduced in R2013a

assertNotEmpty

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is not empty

Syntax

```
assertNotEmpty(assertable,actual)
assertNotEmpty(assertable,actual,diagnostic)
```

Description

`assertNotEmpty(assertable,actual)` asserts that actual is a non-empty MATLAB value.

`assertNotEmpty(assertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assertable.assertThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEmpty`, and replace calls to `verifyNotEmpty` with `assertNotEmpty`.

See Also

`assertEmpty` | `assertThat` | `isempty`

More About

- "Types of Qualifications"

Introduced in R2013a

assertNotEqual

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is not equal to specified value

Syntax

```
assertNotEqual(assertable,actual,notExpected)
assertNotEqual(assertable,actual,notExpected,diagnostic)
```

Description

`assertNotEqual(assertable,actual,notExpected)` asserts that actual is not equal to notExpected.

`assertNotEqual(assertable,actual,notExpected,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assertable.assertThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEqual`, and replace calls to `verifyNotEqual` with `assertNotEqual`.

See Also

`assertEqual` | `assertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assertNotSameHandle

Class: matlab.unittest.TestCase
Package: matlab.unittest

Assert value is not handle to specified instance

Syntax

```
assertNotSameHandle(assertable,actual,notExpectedHandle)
assertNotSameHandle(assertable,actual,notExpectedHandle,diagnostic)
```

Description

`assertNotSameHandle(assertable,actual,notExpectedHandle)` asserts that actual is a different size and/or does not contain the same instances as the notExpectedHandle handle array.

`assertNotSameHandle(assertable,actual,notExpectedHandle,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assertable.assertThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotSameHandle`, and replace calls to `verifyNotSameHandle` with `assertNotSameHandle`.

See Also

`assertSameHandle` | `assertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assertNumElements

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value has specified element count

Syntax

```
assertNumElements(assertable,actual,expectedElementCount)
assertNumElements(assertable,actual,expectedElementCount,diagnostic)
```

Description

`assertNumElements(assertable,actual,expectedElementCount)` asserts that actual is a MATLAB array with expectedElementCount number of elements.

`assertNumElements(assertable,actual,expectedElementCount,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  assertable.assertThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNumElements`, and replace calls to `verifyNumElements` with `assertNumElements`.

See Also

`assertLength` | `assertSize` | `assertThat` | `numel`

More About

- "Types of Qualifications"

Introduced in R2013a

assertReturnsTrue

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert function returns true when evaluated

Syntax

```
assertReturnsTrue(assertable,actual)
assertReturnsTrue(assertable,actual,diagnostic)
```

Description

`assertReturnsTrue(assertable,actual)` asserts that actual is a function handle that returns a scalar logical whose value is true.

`assertReturnsTrue(assertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `assertTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `assertTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  assertable.assertThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test

methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyReturnsTrue`, and replace calls to `verifyReturnsTrue` with `assertReturnsTrue`.

See Also

`assertThat` | `assertTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

assertSameHandle

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert two values are handles to same instance

Syntax

```
assertSameHandle(assertable,actual,expectedHandle)
assertSameHandle(assertable,actual,expectedHandle,diagnostic)
```

Description

`assertSameHandle(assertable,actual,expectedHandle)` asserts that actual is the same size and contains the same instances as the expectedHandle handle array.

`assertSameHandle(assertable,actual,expectedHandle,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assertable.assertThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedHandle`

The expected handle array.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySameHandle`, and replace calls to `verifySameHandle` with `assertSameHandle`.

See Also

`handle` | `assertNotSameHandle` | `assertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assertSize

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value has specified size

Syntax

```
assertSize(assertable,actual,expectedSize)
assertSize(assertable,actual,expectedSize,diagnostic)
```

Description

`assertSize(assertable,actual,expectedSize)` asserts that actual is a MATLAB array whose size is expectedSize.

`assertSize(assertable,actual,expectedSize,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  assertable.assertThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySize`, and replace calls to `verifySize` with `assertSize`.

See Also

`assertLength` | `assertNumElements` | `assertThat` | `size`

More About

- "Types of Qualifications"

Introduced in R2013a

assertSubstring

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert string contains specified string

Syntax

```
assertSubstring(assertable,actual,substring)
assertSubstring(assertable,actual,substring,diagnostic)
```

Description

`assertSubstring(assertable,actual,substring)` asserts that actual is a string that contains substring.

`assertSubstring(assertable,actual,substring,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  assertable.assertThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`substring`

The value to match, specified as a string.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySubstring`, and replace calls to `verifySubstring` with `assertSubstring`.

See Also

`assertMatches` | `assertThat` | `strfind`

More About

- "Types of Qualifications"

Introduced in R2013a

assertThat

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert that value meets specified constraint

Syntax

```
assertThat(assertable,actual,constraint)
assertThat(assertable,actual,constraint,diagnostic)
```

Description

`assertThat(assertable,actual,constraint)` asserts that actual is a value that satisfies the constraint provided.

If the constraint is not satisfied, an assertion failure is produced utilizing only the framework diagnostic generated by the constraint.

`assertThat(assertable,actual,constraint,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

When using this signature, both the diagnostic information contained within diagnostic is used in addition to the diagnostic information provided by the constraint.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyThat`, and replace calls to `verifyThat` with `assertThat`.

Tips

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

More About

- "Types of Qualifications"

Introduced in R2013a

assertTrue

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert value is true

Syntax

```
assertTrue(assertable,actual)
assertTrue(assertable,actual,diagnostic)
```

Description

`assertTrue(assertable,actual)` asserts that actual is a scalar logical with the value of true.

`assertTrue(assertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and non-zero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  assertable.assertThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `assertThat`.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered to by other constraints such as `IsEqualTo`. A similar approach that is generally

less performant but can provide slightly better diagnostic information is the use of `assertReturnsTrue`, which at least shows the display of the function evaluated to generate the failing result.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyTrue`, and replace calls to `verifyTrue` with `assertTrue`.

See Also

`assertFalse` | `assertReturnsTrue` | `assertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assertWarning

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert function issues specified warning

Syntax

```
assertWarning(assertable,actual,warningID)
assertWarning(assertable,actual,warningID,diagnostic)
[output1,...,outputN] = assertWarning( ___ )
```

Description

`assertWarning(assertable,actual,warningID)` asserts that actual issues a warning with the identifier warningID.

`assertWarning(assertable,actual,warningID,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`[output1,...,outputN] = assertWarning( ___ )` also returns the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  assertable.assertThat(actual, IssuesWarnings({warningID}));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`warningID`

Warning ID, specified as a string.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

See examples for `verifyWarning`, and replace calls to `verifyWarning` with `assertWarning`.

See Also

`assertError` | `assertThat` | `assertWarningFree` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

assertWarningFree

Class: matlab.unittest.qualifications.Assertable
Package: matlab.unittest.qualifications

Assert function issues no warnings

Syntax

```
assertWarningFree(assertable,actual)
assertWarningFree(assertable,actual,diagnostic)
[output1,...,outputN] = assertWarningFree( ___ )
```

Description

`assertWarningFree(assertable,actual)` asserts that actual is a function handle that issues no warnings.

`assertWarningFree(assertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`[output1,...,outputN] = assertWarningFree( ___ )` also returns the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  assertable.assertThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

See examples for `verifyWarningFree`, and replace calls to `verifyWarningFree` with `assertWarningFree`.

See Also

`assertThat` | `assertWarning` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

matlab.unittest.qualifications.AssertionFailedException class

Package: matlab.unittest.qualifications

Exception used for assertion failures

Description

The `AssertionFailedException` class provides an exception used for assertion failures. This class is used exclusively by the `Assertable` qualification type.

See Also

`MException` | `Assertable`

matlab.unittest.qualifications.Assumable class

Package: matlab.unittest.qualifications

Qualification to filter test content

Description

The `Assumable` class provides a qualification to filter test content. Apart from actions performed in the event of failures, the `Assumable` class works the same as other `matlab.unittest` qualifications.

Upon an assumption failure, the `Assumable` class informs the testing framework of the failure by throwing an `AssumptionFailedException`. The test framework then marks the test content as filtered and continues testing. Often, assumptions are used to ensure that the test is run only when certain preconditions are met. However, running the test without satisfying the preconditions does not produce a test failure. Ensure that the test content is "Exception Safe" on page 1-6297. If the failure condition is meant to produce a test failure, use assertions or verifications instead of assumptions.

The attributes specified in the `TestCase` method definition determine which tests are filtered. The following behavior occurs when the test framework encounters an assumption failure inside of a `TestCase` method:

- If you define the `TestCase` method using the `Test` attribute, the framework marks the entire method as filtered and runs subsequent test methods.
- If you define the `TestCase` method using the `TestMethodSetup` or `TestMethodTeardown` attributes, the test framework marks the method to run for that instance as filtered.
- If you define the `TestCase` method using the `TestClassSetup` or `TestClassTeardown` attributes, the test framework filters the entire `TestCase` class.

Filtering test content using assumptions does not produce test failures. Therefore, dead test code can result. Avoid this by monitoring filtered tests.

Methods

assumeClass	Assume exact class of specified value
assumeEmpty	Assume value is empty
assumeEqual	Assume value is equal to specified value
assumeError	Assume function throws specified exception
assumeFail	Produce unconditional assumption failure
assumeFalse	Assume value is false
assumeGreaterThan	Assume value is greater than specified value
assumeGreaterThanOrEqual	Assume value is greater than or equal to specified value
assumeInstanceOf	Assume value is object of specified type
assumeLength	Assume value has specified length
assumeLessThan	Assume value is less than specified value
assumeLessThanOrEqual	Assume value is less than or equal to specified value
assumeMatches	Assume string matches specified regular expression
assumeNotEmpty	Assume value is not empty

assumeNotEqual	Assume value is not equal to specified value
assumeNotSameHandle	Assume value is not handle to specified instance
assumeNumElements	Assume value has specified element count
assumeReturnsTrue	Assume function returns true when evaluated
assumeSameHandle	Assume two values are handles to same instance
assumeSize	Assume value has specified size
assumeSubstring	Assume string contains specified string
assumeThat	Assume value meets specified constraint
assumeTrue	Assume value is true
assumeWarning	Assume function issues specified warning
assumeWarningFree	Assume function issues no warnings

Events

`AssumptionFailed`	Triggered upon failing assumption. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionPassed`	Triggered upon passing assumption. A `QualificationEventData` object is passed to listener callback functions.

Definitions

Exception Safe

Test content is *exception safe* when all fixture teardown is performed with `addTeardown` or through the appropriate object destructors when a failure occurs. This ensures that the failure does not affect later testing due to stale fixtures.

This code is not exception safe. After an assertion failure, the test framework does not close the figure.

```
% Not exception safe
f = figure;
testCase.assumeEqual(actual, expected)
close(f)
```

This code is exception safe because the test framework closes the figure in all cases.

```
% Exception safe
f = figure;
testCase.addTeardown(@close, f)
testCase.assumeEqual(actual, expected)
```

However, tearing down a fixture using `addTeardown` does not guarantee code is exception safe. This code shows a failure in `assumeEqual`.

```
% Not exception safe
f = figure;
testCase.assumeEqual(actual, expected)
testCase.addTeardown(@close, f)
```

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Write TestClassSetup Method Using Assumptions

Assumptions assure that a test runs only when certain preconditions are satisfied and when such an event should not produce a test failure. When an assumption failure occurs, the test is marked as filtered.

Create `IsSupportedTest` test case. Refer to the following `IsSupportedTest` test case in the subsequent steps in this example, which highlight specific functions in the file.

IsSupportedTest Class Definition File

```
classdef IsSupportedTest < matlab.unittest.TestCase
    methods(TestClassSetup)
        function TestPlatform(testcase)
            testcase.assumeFalse(ispc,...
                'Do not run any of these tests on Windows.')
        end
    end
    methods(Test)
        function test1(testcase)
            % write test code here
        end
    end
end
```

To execute the MATLAB commands in this example, add the `IsSupportedTest.m` file to a folder on your MATLAB path.

Write Test to Verify Platform. All tests in this test case must run on UNIX platforms only. The `TestPlatform` function uses the `assumeFalse` method to test if MATLAB is running on a Windows platform. If it is, the test fails.

```
function TestPlatform(testcase)
    testcase.assumeFalse(ispc,...
        'Do not run any of these tests on Windows.')
end
```

Make `TestPlatform` a `TestClassSetup` Test. To make the `TestPlatform` test a precondition, add it inside the `methods (TestClassSetup)` block.

Run the test case. Create a test case object and run the tests on a Windows platform.

```
tc = IsSupportedTest;
res = tc.run;

Running IsSupportedTest

================================================================================
All tests in IsSupportedTest were filtered.
    Test Diagnostic: Do not run any of these tests on Windows.
    Details
================================================================================

Done IsSupportedTest
__________

Failure Summary:

     Name                   Failed  Incomplete  Reason(s)
    ============================================================================
     IsSupportedTest/test1              X       Filtered by assumption.
```

The test(s) were filtered, and did not run (marked `Incomplete`).

For more information, click the Details link.

```
================================================================================
An assumption was not met while setting up or tearing down IsSupportedTest.
As a result, all IsSupportedTest tests were filtered.

    ----------------
    Test Diagnostic:
    ----------------
    Do not run any of these tests on Windows.

    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeFalse failed.
    --> The value must evaluate to "false".

    Actual Value:
             1

    ------------------
    Stack Information:
    ------------------
```

```
    In C:\work\IsSupportedTest.m (IsSupportedTest.TestPlatform) at 4
==============================================================================
```

The link to `IsSupportedTest.TestPlatform` under `Stack Information` takes you to the failed `assumeFalse` method.

See Also

`Assertable` | `FatalAssertable` | `matlab.unittest.qualifications` | `QualificationEventData` | `TestCase` | `Verifiable`

More About

- "Types of Qualifications"
- "Dynamically Filtered Tests"

Introduced in R2013a

assumeClass

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume exact class of specified value

Syntax

```
assumeClass(assumable,actual,className)
assumeClass(assumable,actual,metaClass)
assumeClass( ___ ,diagnostic)
```

Description

`assumeClass(assumable,actual,className)` assumes that actual is a MATLAB value whose class is the class specified by className.

`assumeClass(assumable,actual,metaClass)` assumes that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass. The instance must be an exact class match. See `assumeInstanceOf` to assume inclusion in a class hierarchy.

`assumeClass( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  assumable.assumeThat(actual, IsOfClass(className));
  assumable.assumeThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures

result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyClass`, and replace calls to `verifyClass` with `assumeClass`.

See Also

`assumeInstanceOf` | `assumeThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeEmpty

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is empty

Syntax

```
assumeEmpty(assumable,actual)
assumeEmpty(assumable,actual,diagnostic)
```

Description

`assumeEmpty(assumable,actual)` assumes that actual is an empty MATLAB value.

`assumeEmpty(assumable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assumable.assumeThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not

require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyEmpty`, and replace calls to `verifyEmpty` with `assumeEmpty`.

See Also

`assumeNotEmpty` | `assumeThat` | `isempty`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeEqual

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is equal to specified value

Syntax

```
assumeEqual(assumable,actual,expected)
assumeEqual( ___ ,Name,Value)
assumeEqual( ___ ,diagnostic)
```

Description

`assumeEqual(assumable,actual,expected)` assumes that actual is strictly equal to expected.

`assumeEqual( ___ ,Name,Value)` assumes equality with additional options specified by one or more Name,Value pair arguments.

`assumeEqual( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
assumable.assumeThat(actual, IsEqualTo(expected));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
assumable.assumeThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));
```

```
import matlab.unittest.constraints.IsEqualTo;
```

```
import matlab.unittest.constraints.RelativeTolerance;
assumable.assumeThat(actual, IsEqualTo(expected, ...
    'Within', RelativeTolerance(reltol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
assumable.assumeThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via `assumeThat`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'AbsTol'`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`'RelTol'`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

See examples for `verifyEqual`, and replace calls to `verifyEqual` with `assumeEqual`.

See Also

`assumeNotEqual` | `assumeThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeError

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume function throws specified exception

Syntax

```
assumeError(assumable,actual,identifier)
assumeError(assumable,actual,metaClass)
assumeError( ___ ,diagnostic)
```

Description

`assumeError(assumable,actual,identifier)` assumes that actual is a function handle that throws an exception with an error identifier that is equal to identifier.

`assumeError(assumable,actual,metaClass)` assumes that actual is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in metaClass. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class.

`assumeError( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  assumable.assumeThat(actual, Throws(identifier));
  assumable.assumeThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `assumeThat`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`identifier`

Error identifier, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyError`, and replace calls to `verifyError` with `assumeError`.

See Also

`MException` | `assumeThat` | `assumeWarning` | `error`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeFail

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Produce unconditional assumption failure

Syntax

```
assumeFail(assumable)
assumeFail(assumable,diagnostic)
```

Description

`assumeFail(assumable)` produces an unconditional assumption failure when encountered.

`assumeFail(assumable,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Tips

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Examples

See examples for `verifyFail`, and replace calls to `verifyFail` with `assumeFail`.

More About

- "Types of Qualifications"

Introduced in R2013a

assumeFalse

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is false

Syntax

```
assumeFalse(assumable,actual)
assumeFalse(assumable,actual,diagnostic)
```

Description

`assumeFalse(assumable,actual)` assumes that actual is a scalar logical with the value of false.

`assumeFalse(assumable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  assumable.assumeThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `assumeThat`.
- Unlike `assumeTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `assumeTrue`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures

result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

For examples, see `verifyFalse`, and replace calls to `verifyFalse` with `assumeFalse`.

See Also

`assumeThat` | `assumeTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeGreaterThan

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is greater than specified value

Syntax

```
assumeGreaterThan(assumable,actual,floor)
assumeGreaterThan(assumable,actual,floor,diagnostic)
```

Description

`assumeGreaterThan(assumable,actual,floor)` assumes that all elements of actual are greater than all the elements of floor.

`assumeGreaterThan(assumable,actual,floor,diagnostic)` also displays the diagnostic information in diagnostic upon a failure

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThan;
  assumable.assumeThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThan`, and replace calls to `verifyGreaterThan` with `assumeGreaterThan`.

See Also

`matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic` | `assumeGreaterThanOrEqual` | `assumeLessThan` | `assumeLessThanOrEqual` | `assumeThat` | `gt`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeGreaterThanOrEqual

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is greater than or equal to specified value

Syntax

```
assumeGreaterThanOrEqual(assumable,actual,floor)
assumeGreaterThanOrEqual(assumable,actual,floor,diagnostic)
```

Description

`assumeGreaterThanOrEqual(assumable,actual,floor)` assumes that all elements of actual are greater than or equal to all the elements of floor.

`assumeGreaterThanOrEqual(assumable,actual,floor,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  assumable.assumeThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThanOrEqual`, and replace calls to `verifyGreaterThanOrEqual` with `assumeGreaterThanOrEqual`.

See Also

`matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `assumeGreaterThan` | `assumeLessThan` | `assumeLessThanOrEqual` | `assumeThat` | `ge`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeInstanceOf

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is object of specified type

Syntax

```
assumeInstanceOf(assumable,actual,className)
assumeInstanceOf(assumable,actual,metaClass)
assumeInstanceOf( ___ ,diagnostic)
```

Description

`assumeInstanceOf(assumable,actual,className)` assumes that actual is a MATLAB value whose class is the class specified by className.

`assumeInstanceOf(assumable,actual,metaClass)` assumes that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass.

`assumeInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsInstanceOf;
  assumable.assumeThat(actual, IsInstanceOf(className));
  assumable.assumeThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures

result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyInstanceOf`, and replace calls to `verifyInstanceOf` with `assumeInstanceOf`.

See Also

`assumeClass` | `assumeThat` | `isa`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeLength

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value has specified length

Syntax

```
assumeLength(assumable,actual,expectedLength)
assumeLength(assumable,actual,expectedLength,diagnostic)
```

Description

`assumeLength(assumable,actual,expectedLength)` assumes that actual is a MATLAB array whose length is expectedLength.

`assumeLength(assumable,actual,expectedLength,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  assumable.assumeThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLength`, and replace calls to `verifyLength` with `assumeLength`.

See Also

`assumeNumElements` | `assumeSize` | `assumeThat` | `length`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeLessThan

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is less than specified value

Syntax

```
assumeLessThan(assumable,actual,ceiling)
assumeLessThan(assumable,actual,ceiling,diagnostic)
```

Description

`assumeLessThan(assumable,actual,ceiling)` assumes that all elements of actual are less than all the elements of ceiling.

`assumeLessThan(assumable,actual,ceiling,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assumable.assumeThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as ceiling unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThan`, and replace calls to LessThan with `assumeLessThan`.

See Also

`matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic` | `assumeGreaterThan` | `assumeGreaterThanOrEqual` | `assumeLessThanOrEqual` | `assumeThat` | `lt`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeLessThanOrEqual

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is less than or equal to specified value

Syntax

```
assumeLessThanOrEqual(assumable,actual,ceiling)
assumeLessThanOrEqual(assumable,actual,ceiling,diagnostic)
```

Description

`assumeLessThanOrEqual(assumable,actual,ceiling)` assumes that all elements of actual are less than or equal to all the elements of ceiling.

`assumeLessThanOrEqual(assumable,actual,ceiling,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  assumable.assumeThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as ceiling unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThanOrEqual`, and replace calls to `verifyLessThanOrEqual` with `assumeLessThanOrEqual`.

See Also

`matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `assumeGreaterThan` | `assumeGreaterThanOrEqual` | `assumeLessThan` | `assumeThat` | `le`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeMatches

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume string matches specified regular expression

Syntax

```
assumeMatches(assumable,actual,expression)
assumeMatches(assumable,actual,expression,diagnostic)
```

Description

`assumeMatches(assumable,actual,expression)` assumes that actual is a string that matches the regular expression defined by expression.

`assumeMatches(assumable,actual,expression,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  assumable.assumeThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The string to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyMatches`, and replace calls to `verifyMatches` with `assumeMatches`.

See Also

`assumeSubstring` | `assumeThat` | `regexp`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeNotEmpty

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is not empty

Syntax

```
assumeNotEmpty(assumable,actual)
assumeNotEmpty(assumable,actual,diagnostic)
```

Description

`assumeNotEmpty(assumable,actual)` assumes that actual is a non-empty MATLAB value.

`assumeNotEmpty(assumable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assumable.assumeThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEmpty`, and replace calls to `verifyNotEmpty` with `assumeNotEmpty`.

See Also

`assumeEmpty` | `assumeThat` | `isempty`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeNotEqual

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is not equal to specified value

Syntax

```
assumeNotEqual(assumable,actual,notExpected)
assumeNotEqual(assumable,actual,notExpected,diagnostic)
```

Description

`assumeNotEqual(assumable,actual,notExpected)` assumes that actual is not equal to notExpected.

`assumeNotEqual(assumable,actual,notExpected,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assumable.assumeThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEqual`, and replace calls to `verifyNotEqual` with `assumeNotEqual`.

See Also

`assumeEqual` | `assumeThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeNotSameHandle

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is not handle to specified instance

Syntax

```
assumeNotSameHandle(assumable,actual,notExpectedHandle)
assumeNotSameHandle(assumable,actual,notExpectedHandle,diagnostic)
```

Description

`assumeNotSameHandle(assumable,actual,notExpectedHandle)` assumes that actual is a different size and/or does not contain the same instances as the notExpectedHandle handle array.

`assumeNotSameHandle(assumable,actual,notExpectedHandle,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assumable.assumeThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotSameHandle`, and replace calls to `verifyNotSameHandle` with `assumeNotSameHandle`.

See Also

`assumeSameHandle` | `assumeThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeNumElements

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value has specified element count

Syntax

```
assumeNumElements(assumable,actual,expectedElementCount)
assumeNumElements(assumable,actual,expectedElementCount,diagnostic)
```

Description

`assumeNumElements(assumable,actual,expectedElementCount)` assumes that actual is a MATLAB array with expectedElementCount number of elements.

`assumeNumElements(assumable,actual,expectedElementCount,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  assumable.assumeThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNumElements`, and replace calls to `verifyNumElements` with `assumeNumElements`.

See Also

`assumeLength` | `assumeSize` | `assumeThat` | `numel`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeReturnsTrue

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume function returns true when evaluated

Syntax

```
assumeReturnsTrue(assumable,actual)
assumeReturnsTrue(assumable,actual,diagnostic)
```

Description

`assumeReturnsTrue(assumable,actual)` assumes that actual is a function handle that returns a scalar logical whose value is true.

`assumeReturnsTrue(assumable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `assumeTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `assumeTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  assumable.assumeThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures

result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyReturnsTrue`, and replace calls to `verifyReturnsTrue` with `assumeReturnsTrue`.

See Also

`assumeThat` | `assumeTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeSameHandle

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume two values are handles to same instance

Syntax

```
assumeSameHandle(assumable,actual,expectedHandle)
assumeSameHandle(assumable,actual,expectedHandle,diagnostic)
```

Description

`assumeSameHandle(assumable,actual,expectedHandle)` assumes that actual is the same size and contains the same instances as the expectedHandle handle array.

`assumeSameHandle(assumable,actual,expectedHandle,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assumable.assumeThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications

are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedHandle`

The expected handle array.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySameHandle`, and replace calls to `verifySameHandle` with `assumeSameHandle`.

See Also

`handle` | `assumeNotSameHandle` | `assumeThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeSize

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value has specified size

Syntax

```
assumeSize(assumable,actual,expectedSize)
assumeSize(assumable,actual,expectedSize,diagnostic)
```

Description

`assumeSize(assumable,actual,expectedSize)` assumes that actual is a MATLAB array whose size is expectedSize.

`assumeSize(assumable,actual,expectedSize,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  assumable.assumeThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySize`, and replace calls to `verifySize` with `assumeSize`.

See Also

`assumeLength` | `assumeNumElements` | `assumeThat` | `size`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeSubstring

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume string contains specified string

Syntax

```
assumeSubstring(assumable,actual,substring)
assumeSubstring(assumable,actual,substring,diagnostic)
```

Description

`assumeSubstring(assumable,actual,substring)` assumes that actual is a string that contains substring.

`assumeSubstring(assumable,actual,substring,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  assumable.assumeThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs

to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`substring`

The value to match, specified as a string.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySubstring`, and replace calls to `verifySubstring` with `assumeSubstring`.

See Also

`assumeMatches` | `assumeThat` | `strfind`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeThat

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value meets specified constraint

Syntax

```
assumeThat(assumable,actual,constraint)
assumeThat(assumable,actual,constraint,diagnostic)
```

Description

`assumeThat(assumable,actual,constraint)` assumes that actual is a value that satisfies the constraint provided.

If the constraint is not satisfied, an assumption failure is produced utilizing only the framework diagnostic generated by the constraint.

`assumeThat(assumable,actual,constraint,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

When using this signature, both the diagnostic information contained within diagnostic is used in addition to the diagnostic information provided by the constraint.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Tips

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Examples

See examples for `verifyThat`, and replace calls to `verifyThat` with `assumeThat`.

More About

- "Types of Qualifications"

Introduced in R2013a

assumeTrue

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume value is true

Syntax

```
assumeTrue(assumable,actual)
assumeTrue(assumable,actual,diagnostic)
```

Description

`assumeTrue(assumable,actual)` assumes that actual is a scalar logical with the value of true.

`assumeTrue(assumable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  assumable.assumeThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `assumeThat`.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered

to by other constraints such as `IsEqualTo`. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of `assumeReturnsTrue`, which at least shows the display of the function evaluated to generate the failing result.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyTrue`, and replace calls to `verifyTrue` with `assumeTrue`.

See Also

`assumeFalse` | `assumeReturnsTrue` | `assumeThat`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeWarning

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume function issues specified warning

Syntax

```
assumeWarning(assumable,actual,warningID)
assumeWarning(assumable,actual,warningID,diagnostic)
[output1,...,outputN] = assumeWarning( ___ )
```

Description

`assumeWarning(assumable,actual,warningID)` assumes that actual issues a warning with the identifier warningID.

`assumeWarning(assumable,actual,warningID,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`[output1,...,outputN] = assumeWarning( ___ )` also return the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  assumable.assumeThat(actual, IssuesWarnings({warningID}));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`warningID`

Warning ID, specified as a string.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

See examples for `verifyWarning`, and replace calls to `verifyWarning` with `assumeWarning`.

See Also

`assumeError` | `assumeThat` | `assumeWarningFree` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

assumeWarningFree

Class: matlab.unittest.qualifications.Assumable
Package: matlab.unittest.qualifications

Assume function issues no warnings

Syntax

```
assumeWarningFree(assumable,actual)
assumeWarningFree(assumable,actual,diagnostic)
[output1,...,outputN] = assumeWarningFree( ___ )
```

Description

`assumeWarningFree(assumable,actual)` assumes that actual is a function handle that issues no warnings.

`assumeWarningFree(assumable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`[output1,...,outputN] = assumeWarningFree( ___ )` also returns the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  assumable.assumeThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

See examples for `verifyWarningFree`, and replace calls to `verifyWarningFree` with `assumeWarningFree`.

See Also

`assumeThat` | `assumeWarning` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

matlab.unittest.qualifications.AssumptionFailedException class

Package: matlab.unittest.qualifications

Exception used for assumption failures

Description

The `AssumptionFailedException` class provides an exception used for assumption failures. This class is used exclusively by the `Assumable` qualification type.

See Also

`MException` | `Assumable`

matlab.unittest.qualifications.ExceptionEventData class

Package: matlab.unittest.qualifications

Event data for ExceptionThrown event listeners

Description

The `ExceptionEventData` class holds event data for `ExceptionThrown` event listeners. `ExceptionThrown` event listeners are callback functions that you register with the testing framework to listen for the `TestRunner` to encounter an error during execution of test content. Typically, authors of custom plugins use this class. Only the test framework constructs this class directly.

Properties

`Exception`

Unexpected exception caught by `TestRunner` during its execution of test content

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.TestRunnerPlugin` | `MException`

Introduced in R2014a

matlab.unittest.qualifications.FatalAssertable class

Package: matlab.unittest.qualifications

Qualification to abort test execution

Description

The `FatalAssertable` class provides a qualification to abort test execution. Apart from actions performed for failures, the `FatalAssertable` class works the same as `matlab.unittest` qualifications.

Upon a fatal assertion failure, the `FatalAssertable` class informs the testing framework of the failure by throwing a `FatalAssertionFailedException`. The test running framework then displays diagnostic information for the failure and aborts the entire test session. This is useful when the software under test contains so many errors that it does not make sense to continue the test session. Also, you can use fatal assertions in fixture teardown to guarantee the fixture state is restored correctly. If it is not restored, the full testing session will aborts and indicates to restart MATLAB before you resume testing. This allows later tests to run in a consistent MATLAB state. If you can recover the fixture teardown and make it "Exception Safe" on page 1-6381 for failures, use assertions instead.

Fatal assertions prevent false test failures due to the failure of a fundamental test. They also prevent false test failures when a prior test failed to restore test fixtures. If the test framework cannot properly tear down fixtures, restart MATLAB to ensure testing can resume in a clean state.

Methods

fatalAssertClass	Fatally assert exact class of specified value
fatalAssertEmpty	Fatally assert value is empty
fatalAssertEqual	Fatally assert value is equal to specified value

fatalAssertError	Fatally assert function throws specified exception
fatalAssertFail	Produce unconditional fatal assertion failure
fatalAssertFalse	Fatally assert value is false
fatalAssertGreaterThanOrEqual	Fatally assert value is greater than or equal to specified value
fatalAssertInstanceOf	Fatally assert value is object of specified type
fatalAssertLength	Fatally assert value has specified length
fatalAssertLessThan	Fatally assert value is less than specified value
fatalAssertLessThanOrEqual	Fatally assert value is less than or equal to specified value
fatalAssertMatches	Fatally assert string matches specified regular expression
fatalAssertNotEmpty	Fatally assert value is not empty
fatalAssertNotEqual	Fatally assert value is not equal to specified value
fatalAssertNotSameHandle	Fatally assert value is not handle to specified instance

fatalAssertNumElements	Fatally assert value has specified element count
fatalAssertReturnsTrue	Fatally assert function returns true when evaluated
fatalAssertSameHandle	Fatally assert two values are handles to same instance
fatalAssertSize	Fatally assert value has specified size
fatalAssertSubstring	Fatally assert string contains specified string
fatalAssertThat	Fatally assert value meets specified constraint
fatalAssertTrue	Fatally assert value is true
fatalAssertWarning	Fatally assert function issues specified warning
fatalAssertWarningFree	Fatally assert function issues no warnings

Events

`FatalAssertionFailed`	Triggered upon failing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionPassed`	Triggered upon passing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.

Definitions

Exception Safe

Test content is *exception safe* when all fixture teardown is performed with `addTeardown` or through the appropriate object destructors when a failure occurs. This ensures that the failure does not affect later testing due to stale fixtures.

This code is not exception safe. After an assertion failure, the test framework does not close the figure.

```
% Not exception safe
f = figure;
testCase.fatalAssertEqual(actual, expected)
close(f)
```

This code is exception safe because the test framework closes the figure in all cases.

```
% Exception safe
f = figure;
testCase.addTeardown(@close, f)
testCase.fatalAssertEqual(actual, expected)
```

However, tearing down a fixture using `addTeardown` does not guarantee code is exception safe. This code shows a failure in `fatalAssertEqual`.

```
% Not exception safe
f = figure;
testCase.fatalAssertEqual(actual, expected);
testCase.addTeardown(@close, f)
```

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Write Helper Function Using Fatal Assertions

A fatal assertion renders the remainder of the current test method invalid because the state is unrecoverable. A *helper function* is a function in the `TestCase` class but not located within any of the `methods` block statement. Execution of these functions is not controlled by the `matlab.unittest` framework.

Add the `DocPolynomSaveLoadTest.m` file to a folder on your MATLAB path. Refer to the helper function, `cleanUpTemporaryFolder`, in the `DocPolynomSaveLoadTest` test case.

DocPolynomSaveLoadTest Class Definition File

```
classdef DocPolynomSaveLoadTest < matlab.unittest.TestCase

    methods (TestClassSetup)
        function addDocPolynomClassToPath(testCase)
            origPath = path;
            testCase.addTeardown(@path, origPath);
            addpath(fullfile(matlabroot, ...
                'help', 'techdoc', 'matlab_oop', 'examples'));
        end
    end

    methods (Test)
        function testSaveLoad(testCase)

            import matlab.unittest.diagnostics.Diagnostic;

            %% Phase 1: Setup
            % Create a temporary working folder
            tempFolder = tempname;
            [success, message] = mkdir(tempFolder);
            testCase.assertTrue(success, ...
                Diagnostic.join('Could not create temporary folder.',...
                message));
            testCase.addTeardown(@() testCase.cleanUpTemporaryFolder(...
                tempFolder));

            % Change to the temporay folder and register the
            % teardown, which restores the original folder
            origFolder = pwd;
```

```
            testCase.addTeardown(@cd, origFolder);
            cd(tempFolder);

            %% Phase 2: Exercise
            % Save the instance to a mat file.
            p = DocPolynom([1, 0, 1]);
            save('DocPolynomFile', 'p');

            % Validate Precondition. Save resulted in valid .mat file
            testCase.assertEqual(exist('DocPolynomFile.mat','file'),...
                2, Diagnostic.join(...
                'mat file was not saved correctly.',@() dir(pwd)));

            loaded = load('DocPolynomFile');


            %% Phase 3: Verify
            testCase.verifyEqual(loaded.p, p,...
                'Loaded polynom did not equal original polynom.');


            %% Phase 4: Teardown
            % Done inline via calls to addTeardown at the points
            % at which the state was changed.


        end
    end

    methods(Access=private)
        function cleanUpTemporaryFolder(testCase,tempFolder)
            % Clean up the temporary folder and fatally assert
            % that it was correctly cleaned up.

            import matlab.unittest.diagnostics.Diagnostic;

            [success, message] = rmdir(tempFolder, 's');
            testCase.fatalAssertTrue(success, ...
                Diagnostic.join('Could not remove temporary folder.',...
                message));
        end
    end

end
```

Make the `cleanUpTemporaryFolder` function a helper function by placing it inside a separate `methods` block.

```
methods(Access=private)
    function cleanUpTemporaryFolder(testCase, tempFolder)
        % code
    end
end
```

Use the `fatalAssertTrue` method to test the `rmdir` success argument for errors. If a fatal assertion occurs, the test run is aborted.

```
function cleanUpTemporaryFolder(testCase, tempFolder)

    import matlab.unittest.diagnostics.Diagnostic

    [success, message] = rmdir(tempFolder, 's');
    testCase.fatalAssertTrue(success, ...
        Diagnostic.join('Could not remove the temporary folder.',...
        message))
end
```

If the `rmdir` function fails, then this test has failed to restore the state of MATLAB and the machine at initial startup. Aborting prevents subsequent tests to fail because MATLAB is left in an unexpected state by this test.

See Also

`Assertable` | `Assumable` | `matlab.unittest.qualifications` | `QualificationEventData` | `TestCase` | `Verifiable`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertClass

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert exact class of specified value

Syntax

```
fatalAssertClass(fatalAssertable,actual,className)
fatalAssertClass(fatalAssertable,actual,metaClass)
fatalAssertClass( ___ ,diagnostic)
```

Description

`fatalAssertClass(fatalAssertable,actual,className)` fatally asserts that actual is a MATLAB value whose class is the class specified by className.

`fatalAssertClass(fatalAssertable,actual,metaClass)` fatally asserts that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass.

`fatalAssertClass( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  fatalAssertable.fatalAssertThat(actual, IsOfClass(className));
  fatalAssertable.fatalAssertThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is

no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyClass`, and replace calls to `verifyClass` with `fatalAssertClass`.

See Also

`fatalAssertInstanceOf` | `fatalAssertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertEmpty

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is empty

Syntax

```
fatalAssertEmpty(fatalAssertable,actual)
fatalAssertEmpty(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertEmpty(fatalAssertable,actual)` fatally asserts that actual is an empty MATLAB value.

`fatalAssertEmpty(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  fatalAssertable.fatalAssertThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyEmpty`, and replace calls to `verifyEmpty` with `fatalAssertEmpty`.

See Also

`fatalAssertNotEmpty` | `fatalAssertThat` | `isempty`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertEqual

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is equal to specified value

Syntax

```
fatalAssertEqual(fatalAssertable,actual,expected)
fatalAssertEqual( ___ ,Name,Value)
fatalAssertEqual( ___ ,diagnostic)
```

Description

`fatalAssertEqual(fatalAssertable,actual,expected)` fatally asserts that actual is strictly equal to expected .

`fatalAssertEqual( ___ ,Name,Value)` fatally asserts equality with additional options specified by one or more Name,Value pair arguments.

`fatalAssertEqual( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.RelativeTolerance;
fatalAassertable.fatalAssertThat(actual, IsEqualTo(expected, ...
```

```
    'Within', RelativeTolerance(reltol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'AbsTol'`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`'RelTol'`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

See examples for `verifyEqual`, and replace calls to `verifyEqual` with `fatalAssertEqual`.

See Also

`fatalAssertNotEqual` | `fatalAssertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertError

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert function throws specified exception

Syntax

```
fatalAssertError(fatalAssertable,actual,identifier)
fatalAssertError(fatalAssertable,actual,metaClass)
fatalAssertError( ___ ,diagnostic)
```

Description

`fatalAssertError(fatalAssertable,actual,identifier)` fatally asserts that actual is a function handle that throws an exception with an error identifier that is equal to identifier.

`fatalAssertError(fatalAssertable,actual,metaClass)` fatally asserts that actual is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in metaClass. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class..

`fatalAssertError( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  fatalAssertable.fatalAssertThat(actual, Throws(identifier));
  fatalAssertable.fatalAssertThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`identifier`

Error identifier, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyError`, and replace calls to `verifyError` with `fatalAssertError`.

See Also

`MException` | `error` | `fatalAssertThat` | `fatalAssertWarning`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertFail

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Produce unconditional fatal assertion failure

Syntax

```
fatalAssertFail(fatalAssertable)
fatalAssertFail(fatalAssertable,diagnostic)
```

Description

`fatalAssertFail(fatalAssertable)` produces an unconditional fatal assertion failure when encountered.

`fatalAssertFail(fatalAssertable,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Tips

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Examples

See examples for `verifyFail`, and replace calls to `verifyFail` with `fatalAssertFail`.

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertFalse

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is false

Syntax

```
fatalAssertFalse(fatalAssertable,actual)
fatalAssertFalse(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertFalse(fatalAssertable,actual)` fatally asserts that actual is a scalar logical with the value of false.

`fatalAssertFalse(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  fatalAssertable.fatalAssertThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `fatalAssertThat`.
- Unlike `fatalAssertTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `fatalAssertTrue`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyFalse`, and replace calls to `verifyFalse` with `fatalAssertFalse`.

See Also

`fatalAssertThat` | `fatalAssertTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertGreaterThan

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is greater than specified value

Syntax

```
fatalAssertGreaterThan(fatalAssertable,actual,floor)
fatalAssertGreaterThan(fatalAssertable,actual,floor,diagnostic)
```

Description

`fatalAssertGreaterThan(fatalAssertable,actual,floor)` fatally asserts that all elements of actual are greater than all the elements of floor.

`fatalAssertGreaterThan(fatalAssertable,actual,floor,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  matlab.unittest.constraints.IsGreaterThan;
  fatalAssertable.fatalAssertThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Default:

Examples

See examples for `verifyGreaterThan`, and replace calls to `verifyGreaterThan` with `fatalAssertGreaterThan`.

See Also

`matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic` | `fatalAssertGreaterThanOrEqual` | `fatalAssertLessThan` | `fatalAssertLessThanOrEqual` | `fatalAssertThat` | `gt`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertGreaterThanOrEqual

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is greater than or equal to specified value

Syntax

```
fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor)
fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor,
diagnostic)
```

Description

`fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor)` fatally asserts that all elements of actual are greater than or equal to all the elements of floor.

`fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor, diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  fatalAssertable.fatalAssertThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThanOrEqual`, and replace calls to `verifyGreaterThanOrEqual` with `fatalAssertGreaterThanOrEqual`.

See Also

`matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `fatalAssertGreaterThan` | `fatalAssertLessThan` | `fatalAssertLessThanOrEqual` | `fatalAssertThat` | `ge`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertInstanceOf

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is object of specified type

Syntax

```
fatalAssertInstanceOf(fatalAssertable,actual,className)
fatalAssertInstanceOf(fatalAssertable,actual,metaClass)
fatalAssertInstanceOf( ___ ,diagnostic)
```

Description

`fatalAssertInstanceOf(fatalAssertable,actual,className)` fatally asserts that actual is a MATLAB value whose class is the class specified by className.

`fatalAssertInstanceOf(fatalAssertable,actual,metaClass)` fatally asserts that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass.

`fatalAssertInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsInstanceOf;
  fatalAssertable.fatalAssertThat(actual, IsInstanceOf(className));
  fatalAssertable.fatalAssertThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is

no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyInstanceOf`, and replace calls to `verifyInstanceOf` with `fatalAssertInstanceOf`.

See Also

`fatalAssertClass` | `fatalAssertThat` | `isa`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertLength

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value has specified length

Syntax

```
fatalAssertLength(fatalAssertable,actual,expectedLength)
fatalAssertLength(fatalAssertable,actual,expectedLength,diagnostic)
```

Description

`fatalAssertLength(fatalAssertable,actual,expectedLength)` fatally asserts that actual is a MATLAB array whose length is expectedLength.

`fatalAssertLength(fatalAssertable,actual,expectedLength,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  fatalAssertable.fatalAssertThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLength`, and replace calls to `verifyLength` with `fatalAssertLength`.

See Also

`fatalAssertNumElements` | `fatalAssertSize` | `fatalAssertThat` | `length`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertLessThan

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is less than specified value

Syntax

```
fatalAssertLessThan(fatalAssertable,actual,ceiling)
fatalAssertLessThan(fatalAssertable,actual,ceiling,diagnostic)
```

Description

`fatalAssertLessThan(fatalAssertable,actual,ceiling)` fatally asserts that all elements of actual are less than all the elements of ceiling.

`fatalAssertLessThan(fatalAssertable,actual,ceiling,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assertable.assertThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `assertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as ceiling unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThan`, and replace calls to `verifyLessThan` with `fatalAssertLessThan`.

See Also

`matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic` | `fatalAssertGreaterThan` | `fatalAssertGreaterThanOrEqual` | `fatalAssertLessThanOrEqual` | `fatalAssertThat` | `lt`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertLessThanOrEqual

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is less than or equal to specified value

Syntax

```
fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling)
fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling,
diagnostic)
```

Description

`fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling)` fatally asserts that all elements of actual are less than or equal to all the elements of ceiling.

`fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling, diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  fatalAssertable.fatalAssertThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as ceiling unless either one is scalar, at which point scalar expansion occurs.

Default:

`actual`

The value to test.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThanOrEqual`, and replace calls to `verifyLessThanOrEqual` with `fatalAssertLessThanOrEqual`.

See Also

`matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `fatalAssertGreaterThan` | `fatalAssertGreaterThanOrEqual` | `fatalAssertLessThan` | `fatalAssertThat` | `le`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertMatches

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert string matches specified regular expression

Syntax

```
fatalAssertMatches(fatalAssertable,actual,expression)
fatalAssertMatches(fatalAssertable,actual,expression,diagnostic)
```

Description

`fatalAssertMatches(fatalAssertable,actual,expression)` fatally asserts that actual is a string that matches the regular expression defined by expression.

`fatalAssertMatches(fatalAssertable,actual,expression,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  fatalAssertable.fatalAssertThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyMatches`, and replace calls to `verifyMatches` with `fatalAssertMatches`.

See Also

`fatalAssertSubstring` | `fatalAssertThat` | `regexp`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertNotEmpty

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is not empty

Syntax

```
fatalAssertNotEmpty(fatalAssertable,actual)
fatalAssertNotEmpty(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertNotEmpty(fatalAssertable,actual)` fatally asserts that actual is a non-empty MATLAB value.

`fatalAssertNotEmpty(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  fatalAssertable.fatalAssertThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEmpty`, and replace calls to `verifyNotEmpty` with `fatalAssertNotEmpty`.

See Also

`fatalAssertEmpty` | `fatalAssertThat` | `isempty`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertNotEqual

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is not equal to specified value

Syntax

```
fatalAssertNotEqual(fatalAssertable,actual,notExpected)
fatalAssertNotEqual(fatalAssertable,actual,notExpected,diagnostic)
```

Description

`fatalAssertNotEqual(fatalAssertable,actual,notExpected)` fatally asserts that actual is not equal to notExpected.

`fatalAssertNotEqual(fatalAssertable,actual,notExpected,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  fatalAssertable.fatalAssertThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

Examples

See examples for `verifyNotEqual`, and replace calls to `verifyNotEqual` with `fatalAssertNotEqual`.

See Also

`fatalAssertEqual` | `fatalAssertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertNotSameHandle

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is not handle to specified instance

Syntax

```
fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle)
fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle,
diagnostic)
```

Description

`fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle)` fatally asserts that actual is a different size and/or does not contain the same instances as the notExpectedHandle handle array.

`fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle, diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  fatalAssertable.fatalAssertThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotSameHandle`, and replace calls to `verifyNotSameHandle` with `fatalAssertNotSameHandle`.

See Also

`fatalAssertSameHandle` | `fatalAssertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertNumElements

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value has specified element count

Syntax

```
fatalAssertNumElements(fatalAssertable,actual,expectedElementCount)
fatalAssertNumElements(fatalAssertable,actual,expectedElementCount,
diagnostic)
```

Description

`fatalAssertNumElements(fatalAssertable,actual,expectedElementCount)` fatally asserts that actual is a MATLAB array with expectedElementCount number of elements.

`fatalAssertNumElements(fatalAssertable,actual,expectedElementCount, diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  fatalAssertable.fatalAssertThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNumElements`, and replace calls to `verifyNumElements` with `fatalAssertNumElements`.

See Also

`fatalAssertLength` | `fatalAssertSize` | `fatalAssertThat` | `numel`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertReturnsTrue

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert function returns true when evaluated

Syntax

```
fatalAssertReturnsTrue(fatalAssertable,actual)
fatalAssertReturnsTrue(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertReturnsTrue(fatalAssertable,actual)` fatally asserts that actual is a function handle that returns a scalar logical whose value is true.

`fatalAssertReturnsTrue(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `fatalAssertTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `fatalAssertTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  fatalAssertable.fatalAssertThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is

no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyReturnsTrue`, and replace calls to `verifyReturnsTrue` with `fatalAssertReturnsTrue`.

See Also

`fatalAssertThat` | `fatalAssertTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertSameHandle

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert two values are handles to same instance

Syntax

```
fatalAssertSameHandle(fatalAssertable,actual,expectedHandle)
fatalAssertSameHandle(fatalAssertable,actual,expectedHandle,
diagnostic)
```

Description

`fatalAssertSameHandle(fatalAssertable,actual,expectedHandle)` fatally asserts that actual is the same size and contains the same instances as the expectedHandle handle array.

`fatalAssertSameHandle(fatalAssertable,actual,expectedHandle, diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  fatalAssertable.fatalAssertThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedHandle`

The expected handle array.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySameHandle`, and replace calls to `verifySameHandle` with `fatalAssertSameHandle`.

See Also

`handle` | `fatalAssertNotSameHandle` | `fatalAssertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertSize

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value has specified size

Syntax

```
fatalAssertSize(fatalAssertable,actual,expectedSize)
fatalAssertSize(fatalAssertable,actual,expectedSize,diagnostic)
```

Description

`fatalAssertSize(fatalAssertable,actual,expectedSize)` fatally asserts that actual is a MATLAB array whose size is expectedSize.

`fatalAssertSize(fatalAssertable,actual,expectedSize,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  fatalAssertable.fatalAssertThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySize`, and replace calls to `verifySize` with `fatalAssertSize`.

See Also

`fatalAssertLength` | `fatalAssertNumElements` | `fatalAssertThat` | `size`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertSubstring

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert string contains specified string

Syntax

```
fatalAssertSubstring(fatalAssertable,actual,substring)
fatalAssertSubstring(fatalAssertable,actual,substring,diagnostic)
```

Description

`fatalAssertSubstring(fatalAssertable,actual,substring)` fatally asserts that actual is a string that contains substring.

`fatalAssertSubstring(fatalAssertable,actual,substring,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  fatalAssertable.fatalAssertThat(actual,...
  ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution

of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`substring`

The value to match, specified as a string.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySubstring`, and replace calls to `verifySubstring` with `fatalAssertSubstring`.

See Also

`fatalAssertMatches` | `fatalAssertThat` | `strfind`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertThat

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value meets specified constraint

Syntax

```
fatalAssertThat(fatalAssertable,actual,constraint)
fatalAssertThat(fatalAssertable,actual,constraint,diagnostic)
```

Description

`fatalAssertThat(fatalAssertable,actual,constraint)` fatally asserts that actual is a value that satisfies the constraint provided.

If the constraint is not satisfied, a fatal assertion failure is produced utilizing only the framework diagnostic generated by the constraint.

`fatalAssertThat(fatalAssertable,actual,constraint,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

When using this signature, both the diagnostic information contained within diagnostic is used in addition to the diagnostic information provided by the constraint.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Tips

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Examples

See examples for `verifyThat`, and replace calls to `verifyThat` with `fatalAssertThat`.

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertTrue

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert value is true

Syntax

```
fatalAssertTrue(fatalAssertable,actual)
fatalAssertTrue(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertTrue(fatalAssertable,actual)` fatally asserts that actual is a scalar logical with the value of true.

`fatalAssertTrue(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  fatalAssertable.fatalAssertThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `fatalAssertThat`.

  However, this method is optimized for performance and does not construct a new `IsTrue` constraint for each call. Sometimes such use can come at the expense of less

diagnostic information. Use the `fatalAssertReturnsTrue` method for a similar approach which may provide better diagnostic information.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyTrue`, and replace calls to `verifyTrue` with `fatalAssertTrue`.

See Also

`fatalAssertFalse` | `fatalAssertReturnsTrue` | `fatalAssertThat`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertWarning

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert function issues specified warning

Syntax

```
fatalAssertWarning(fatalAssertable,actual,warningID)
fatalAssertWarning(fatalAssertable,actual,warningID,diagnostic)
[output1,...,outputN] = fatalAssertWarning( ___ )
```

Description

`fatalAssertWarning(fatalAssertable,actual,warningID)` fatally asserts that actual issues a warning with the identifier warningID.

`fatalAssertWarning(fatalAssertable,actual,warningID,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`[output1,...,outputN] = fatalAssertWarning( ___ )` also returns the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  fatalAssertable.fatalAssertThat(actual, IssuesWarnings({warningID}));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture

teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`warningID`

Warning ID, specified as a string.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

See examples for `verifyWarning`, and replace calls to `verifyWarning` with `fatalAssertWarning`.

See Also

`fatalAssertError` | `fatalAssertThat` | `fatalAssertWarningFree` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

fatalAssertWarningFree

Class: matlab.unittest.qualifications.FatalAssertable
Package: matlab.unittest.qualifications

Fatally assert function issues no warnings

Syntax

```
fatalAssertWarningFree(fatalAssertable,actual)
fatalAssertWarningFree(fatalAssertable,actual,diagnostic)
[output1,...,outputN] = fatalAssertWarningFree( ___ )
```

Description

`fatalAssertWarningFree(fatalAssertable,actual)` fatally asserts that actual is a function handle that issues no warnings.

`fatalAssertWarningFree(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`[output1,...,outputN] = fatalAssertWarningFree( ___ )` also returns the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  fatalAssertable.fatalAssertThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture

teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

See examples for `verifyWarningFree`, and replace calls to `verifyWarningFree` with `fatalAssertWarningFree`.

See Also

`fatalAssertThat` | `fatalAssertWarning` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

matlab.unittest.qualifications.FatalAssertionFailedException class

Package: matlab.unittest.qualifications

Exception used for fatal assertion failures

Description

The `FatalAssertionFailedException` class provides an exception used for fatal assertion failures. This class is used exclusively by the `FatalAssertable` qualification type.

See Also

`MException` | `FatalAssertable`

matlab.unittest.qualifications.QualificationEventData class

Package: matlab.unittest.qualifications

Event data for qualification event listeners

Description

The `QualificationEventData` class holds event data for qualification event listeners. Qualification event listeners are callback functions that you register with the testing framework to listen for passing and/or failing qualifications. Qualifications can be assertions, fatal assertions, assumptions, or verifications performed on test content. The corresponding qualification classes define these events. Typically, authors of custom plugins use this class. Only the test framework constructs this class directly.

Properties

`ActualValue`

Value tested to satisfy the qualification logic of the `Constraint`

`Constraint`

Instance of `matlab.unittest.constraints.Constraint` used for the qualification

When you use a qualification method on a `TestCase` or `Fixture` object, the `Constraint` property contains the underlying constraint used for the qualification. For example, if you use the `verifyEqual` method, the underlying constraint is the `IsEqualTo` constraint. Therefore, if you invoke the constraint's `getDiagnosticFor` method, the diagnostic result can appear different than what the test framework displays.

`TestDiagnostic`

Diagnostic specified in the qualification, represented as a string, function handle, or instance of the `Diagnostic` class

`TestDiagnosticResult`

Result of diagnostic specified in the qualification, represented as a cell array of strings

`FrameworkDiagnosticResult`

Result of diagnostic from constraint used for the qualification, represented as a cell array of strings

`Stack`

Function call stack leading up to the qualification event, represented as a structure array

See Also

`matlab.unittest.qualifications.Assertable` | `matlab.unittest.qualifications.Assumable` | `matlab.unittest.qualifications.FatalAssertable` | `matlab.unittest.qualifications.Verifiable` | `matlab.unittest.fixtures.Fixture`

Introduced in R2014a

matlab.unittest.qualifications.Verifiable class

Package: matlab.unittest.qualifications

Qualification to produce soft-failure conditions

Description

The `Verifiable` class provides a qualification to produce soft-failure conditions. Apart from actions performed for failures, the `Verifiable` class works the same as other `matlab.unittest` qualifications.

Upon a verification failure, the `Verifiable` class informs the testing framework of the failure, including all diagnostic information associated with the failure, but continues to execute the currently running test without throwing an `MException`. This is most useful when a failure at the verification point is not fatal to the remaining test content. Often, you use verifications as the primary verification of a Four-Phase Test. Use other qualification types, such as assertions, fatal assertions, and assumptions, to test for violation of preconditions or incorrect test setup.

Since verifications do not throw `MExecptions`, all test content runs to completion even when verification failures occur. This helps you understand how close a piece of software is to meeting the test suite requirements. Qualification types that throw exceptions do not provide this insight, since once an exception is thrown an arbitrary amount of code remains that is not reached or exercised. Verifications also provide more testing coverage in failure conditions. However, when you overuse verifications, they can produce excess noise for a single failure condition. If a failure condition will cause later qualification points to also fail, use assertions or fatal assertions instead.

Methods

verifyClass	Verify exact class of specified value
verifyEmpty	Verify value is empty
verifyEqual	Verify value is equal to specified value

verifyError	Verify function throws specified exception
verifyFail	Produce unconditional verification failure
verifyFalse	Verify value is false
verifyGreaterThan	Verify value is greater than specified value
verifyGreaterThanOrEqual	Verify value is greater than or equal to specified value
verifyInstanceOf	Verify value is object of specified type
verifyLength	Verify value has specified length
verifyLessThan	Verify value is less than specified value
verifyLessThanOrEqual	Verify value is less than or equal to specified value
verifyMatches	Verify string matches specified regular expression
verifyNotEmpty	Verify value is not empty
verifyNotEqual	Verify value is not equal to specified value
verifyNotSameHandle	Verify value is not handle to specified instance
verifyNumElements	Verify value has specified element count
verifyReturnsTrue	Verify function returns true when evaluated

verifySameHandle	Verify two values are handles to same instance
verifySize	Verify value has specified size
verifySubstring	Verify string contains specified string
verifyThat	Verify value meets given constraint
verifyTrue	Verify value is true
verifyWarning	Verify function issues specified warning
verifyWarningFree	Verify function issues no warnings

Events

`VerificationFailed`	Triggered upon failing verification. A `QualificationEventData` object is passed to listener callback functions.
`VerificationPassed`	Triggered upon passing verification. A `QualificationEventData` object is passed to listener callback functions.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Write Test Methods Using Verifications

Verifications produce and record failures without throwing an exception, meaning the currently running test runs to completion. This example creates a test case to verify arithmetic operations on objects of the `DocPolynom` example class.

Create the `DocPolynomTest` Test Case. Refer to the following `DocPolynomTest` test case in the subsequent steps in this example, which highlight specific functions in the file.

DocPolynomTest Class Definition File

```
classdef DocPolynomTest < matlab.unittest.TestCase
    % Tests the DocPolynom class.

   properties
      msgEqn = 'Equation under test: ';
   end

   methods (TestClassSetup)
        function addDocPolynomClassToPath(testCase)
            testCase.addTeardown(@path,addpath(fullfile(matlabroot,...
                'help', 'techdoc', 'matlab_oop', 'examples')))
        end
    end

    methods (Test)
        function testConstructor(testCase)
            p = DocPolynom([1, 0, 1]);
            testCase.verifyClass(p, ?DocPolynom)
        end

        function testAddition(testCase)
            p1 = DocPolynom([1, 0, 1]);
            p2 = DocPolynom([5, 2]);

            actual = p1 + p2;
            expected = DocPolynom([1, 5, 3]);

            msg = [testCase.msgEqn,...
                '(x^2 + 1) + (5*x + 2) = x^2 + 5*x + 3'];
```

```
            testCase.verifyEqual(actual, expected, msg)
        end

        function testMultiplication(testCase)
            p1 = DocPolynom([1, 0, 3]);
            p2 = DocPolynom([5, 2]);

            actual = p1 * p2;
            expected = DocPolynom([5, 2, 15, 6]);

            msg = [testCase.msgEqn,...
                '(x^2 + 3) * (5*x + 2) = 5*x^3 + 2*x^2 + 15*x + 6'];
            testCase.verifyEqual(actual, expected, msg)
        end

    end
end
```

To execute the MATLAB commands in this example, add the `DocPolynomTest.m` file to a folder on your MATLAB path.

Write Test to Verify Constructor. Create a function, `testConstructor`, using the `verifyClass` method to test the `DocPolynom` class constructor.

```
function testConstructor(testCase)
    p = DocPolynom([1, 0, 1]);
    testCase.verifyClass(p, ?DocPolynom)
end
```

Write Tests to Verify Operations. In the `testAddition` function, use the `verifyEqual` method to test the equation `(x^2 + 1) + (5*x + 2) = x^2 + 5*x + 3`. The `verifyEqual` method includes this equation in the diagnostic argument.

```
function testAddition(testCase)
    p1 = DocPolynom([1, 0, 1]);
    p2 = DocPolynom([5, 2]);

    actual = p1 + p2;
    expected = DocPolynom([1, 5, 3]);

    msg = [testCase.msgEqn,...
        '(x^2 + 1) + (5*x + 2) = x^2 + 5*x + 3'];
    testCase.verifyEqual(actual, expected, msg)
end
```

The function, `testMultiplication`, tests multiplication operations.

Run the tests in the `DocPolynomTest` test case.

```
tc = DocPolynomTest;
ts = matlab.unittest.TestSuite.fromClass(?DocPolynomTest);
res = run(ts);
```

```
Running DocPolynomTest
...
Done DocPolynomTest
__________
```

All tests passed.

See Also

`Assertable` | `Assumable` | `FatalAssertable` | `matlab.unittest.qualifications` | `QualificationEventData` | `TestCase`

More About

- "Types of Qualifications"

External Web Sites

- Four-Phase Test

Introduced in R2013a

verifyClass

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify exact class of specified value

Syntax

```
verifyClass(verifiable,actual,className)
verifyClass(verifiable,actual,metaClass)
verifyClass( ___ ,diagnostic)
```

Description

`verifyClass(verifiable,actual,className)` verifies that actual is a MATLAB value whose class is the class specified by className.

`verifyClass(verifiable,actual,metaClass)` verifies that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass. The instance must be an exact class match. Use `verifyInstanceOf` to verify inclusion in a class hierarchy.

`verifyClass( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- The method is functionally equivalent to the following methods:

  ```
  import matlab.unittest.constraints.IsOfClass;
  verifiable.verifyThat(actual, IsOfClass(className));
  verifiable.verifyThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test a Class

These interactive tests verify the class of the number, 5.

Create a `TestCase` object and the value to test.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
actvalue = 5;
```

Verify class of `actvalue` is `double`.

```
verifyClass(testCase, actvalue, 'double');
```

```
Interactive verification passed.
```

Verify class of `actvalue` is `char`.

```
verifyClass(testCase, actvalue, 'char');
```

```
Interactive verification failed.
```

```
----------------------
Framework Diagnostic:
----------------------
verifyClass failed.
--> The value's class is incorrect.

    Actual Class:
        double
    Expected Class:
        char

Actual Value:
         5
```

Test fails.

Test a Function Handle

These interactive tests verify function handles, specified as a meta.class instance, `?function_handle`.

Create a `TestCase` object.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Create a function handle.

```
fh = @sin;
verifyClass(testCase, fh, ?function_handle);
```

```
Interactive verification passed.
```

Test the function name.

```
fh = 'sin';
verifyClass(testCase, fh, ?function_handle);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyClass failed.
--> The value's class is incorrect.
```

```
    Actual Class:
        char
    Expected Class:
        function_handle

Actual Value:
    sin
```

Test fails.

Test a Derived Class

Verify that a derived class is not the same class as its base class.

Create a class, `BaseExample`.

```
classdef BaseExample
end
```

Create a derived class, `DerivedExample`.

```
classdef DerivedExample < BaseExample
end
```

Verify the classes are not equal.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyClass(testCase, DerivedExample(), ?BaseExample);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyClass failed.
--> The value's class is incorrect.

    Actual Class:
        DerivedExample
    Expected Class:
        BaseExample

Actual Value:
      DerivedExample with no properties.
```

Test fails.

Test Class of Output Value

Use `verifyClass` to test the `add5` function returns a `double` value.

Function for unit testing:

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

`TestCase` class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double')
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m')
```

```
result = run(suite);
```

```
Running Add5Test
..
Done Add5Test
__________
```

See Also

`matlab.unittest.constraints.IsOfClass` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyInstanceOf` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyEmpty

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is empty

Syntax

```
verifyEmpty(verifiable,actual)
verifyEmpty( ___ ,diagnostic)
```

Description

`verifyEmpty(verifiable,actual)` verifies that actual is an empty MATLAB value.

`verifyEmpty( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  verifiable.verifyThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures

result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Empty Strings

Create a `TestCase` object.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifyEmpty(testCase, '');
```

```
Interactive verification passed.
```

Test for Empty Arrays

An array with any zero dimension is empty.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEmpty(testCase, ones(2, 5, 0, 3));
```

```
Interactive verification passed.
```

```
verifyEmpty(testCase, [2 3], 'Array is not empty.');
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Array is not empty.

---------------------
Framework Diagnostic:
---------------------
verifyEmpty failed.
--> The value must be empty.
--> The value has a size of [1  2].

Actual Value:
         2     3
```

Test failed.

Test for Empty Cell Arrays

Test empty cell array, `{}`.

```
matlab.unittest.TestCase.forInteractiveUse;
verifyEmpty(testCase, {}, 'Cell array is not empty.');

Interactive verification passed.
```

A cell array of empty arrays is not empty.

```
verifyEmpty(testCase, {[], [], []}, 'Cell array is not empty.');

Interactive verification failed.

----------------
Test Diagnostic:
----------------
Cell array is not empty.

---------------------
Framework Diagnostic:
---------------------
verifyEmpty failed.
--> The value must be empty.
--> The value has a size of [1  3].

Actual Value:
        []    []    []
```

Test failed.

Test for Empty Test Suite

Test for empty object, `emptyTestSuite`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
emptyTestSuite = matlab.unittest.TestSuite.empty;
verifyEmpty(testCase, emptyTestSuite);

Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsEmpty` | `matlab.unittest.diagnostics.Diagnostic` | `isempty` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyNotEmpty` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyEqual

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is equal to specified value

Syntax

```
verifyEqual(verifiable,actual,expected)
verifyEqual( ___ ,Name,Value)
verifyEqual( ___ ,diagnostic)
```

Description

`verifyEqual(verifiable,actual,expected)` verifies that actual is strictly equal to expected.

`verifyEqual( ___ ,Name,Value)` verifies equality with additional options specified by one or more Name,Value pair arguments.

`verifyEqual( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
verifiable.verifyThat(actual, IsEqualTo(expected));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
verifiable.verifyThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));
```

```
import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.RelativeTolerance;
verifiable.verifyThat(actual, IsEqualTo(expected, ...
```

```
    'Within', RelativeTolerance(reltol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
verifiable.verifyThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'AbsTol'`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`'RelTol'`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

Comparing Numeric Values

Numeric values are equivalent if they are of the same class with equivalent size, complexity, and sparsity.

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

A value is equal to itself.

```
verifyEqual(testCase, 5, 5);
```

```
Interactive verification passed.
```

Values must have equal sizes.

```
verifyEqual(testCase, [5 5], 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
--> Sizes do not match.

    Actual double size:
              1     2
    Expected double size:
```

```
             1     1

Actual Value:
         5     5
Expected Value:
         5
```

Test failed.

Test Classes

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEqual(testCase, int8(5), int16(5));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
--> Classes do not match.

    Actual Class:
        int8
    Expected Class:
        int16

Actual Value:
        5
Expected Value:
         5
```

Test failed.

Test Cell Arrays

Each element of a cell array must be equal in value, class, and size.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEqual(testCase, {'cell', struct, 5}, {'cell', struct, 5});
```

```
Interactive verification passed.
```

Test Numeric Tolerances

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifyEqual(testCase, 4.95, 5);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
--> The values are not equal using "isequaln".
--> Failure table:
            Index    Actual    Expected           Error               RelativeError
            _____    ______    ________    ___________________    ____________________

            1        4.95      5           -0.0499999999999998    -0.00999999999999996

Actual Value:
       4.950000000000000
Expected Value:
         5
```

Test failed.

```
verifyEqual(testCase, 1.5, 2, 'AbsTol', 1)

Interactive verification passed.

verifyEqual(testCase, 1.5, 2, 'RelTol', 0.1, ...
    'Difference between actual and expected exceeds relative tolerance')

Interactive verification failed.

----------------
Test Diagnostic:
----------------
Difference between actual and expected exceeds relative tolerance

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
--> The values are not equal using "isequaln".
--> The error was not within relative tolerance.
--> Failure table:
            Index    Actual    Expected    Error    RelativeError    RelativeTolerance
            _____    ______    ________    _____    _____________    _________________
```

```
           1        1.5        2           -0.5      -0.25            0.1

Actual Value:
       1.500000000000000
Expected Value:
         2
```

Test failed.

See Also

`matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.AbsoluteTolerance` | `matlab.unittest.constraints.RelativeTolerance` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyNotEqual` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyError

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify function throws specified exception

Syntax

```
verifyError(verifiable,actual,identifier)
verifyError(verifiable,actual,metaClass)
verifyError( ___ ,diagnostic)
```

Description

`verifyError(verifiable,actual,identifier)` verifies that actual is a function handle that throws an exception with an error identifier that is equal to identifier.

`verifyError(verifiable,actual,metaClass)` verifies that actual is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in metaClass. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class.

`verifyError( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  verifiable.verifyThat(actual, Throws(identifier));
  verifiable.verifyThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`identifier`

Error identifier, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Error IDs

```
testCase = matlab.unittest.TestCase.forInteractiveUse;

% Passing scenarios
%%%%%%%%%%%%%%%%%%%%%%
verifyError(testCase, @() error('SOME:error:id','Error!'), 'SOME:error:id');
verifyError(testCase, @testCase.assertFail, ...
    ?matlab.unittest.qualifications.AssertionFailedException);

% Failing scenarios
%%%%%%%%%%%%%%%%%%%%%%
verifyError(testCase, 5, 'some:id', '5 is not a function handle');
verifyError(testCase, @testCase.verifyFail, ...
    ?matlab.unittest.qualifications.AssertionFailedException, ...
    'Verifications dont throw exceptions.');
verifyError(testCase, @() error('SOME:id'), 'OTHER:id', 'Wrong id');
verifyError(testCase, @() error('whoops'), ...
    ?matlab.unittest.qualifications.AssertionFailedException, ...
```

```
    'Wrong type of exception thrown');
```

Test Error Condition

Create `testNonNumericInput` to test if function throws expected error message, `add5:InputMustBeNumeric`, for unexpected condition, input is `char`.

Function for unit testing:

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

`TestCase` class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double')
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m')
```

```
result = run(suite);
```

```
Running Add5Test
..
Done Add5Test
__________
```

See Also

`MException` | `matlab.unittest.constraints.Throws` | `matlab.unittest.diagnostics.Diagnostic` | `error` |

`matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyThat` | `verifyWarning`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyFail

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Produce unconditional verification failure

Syntax

```
verifyFail(verifiable)
verifyFail(verifiable,diagnostic)
```

Description

`verifyFail(verifiable)` produces an unconditional verification failure when encountered.

`verifyFail(verifiable,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Tips

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Examples

Test for Failure Condition

An example of where this method may be used is in a callback function that should not be executed in a given scenario. A test can confirm this does not occur by unconditionally performing a failure if the code path is reached.

Create a handle class, `MyHandle`, with a `SomethingHappened` event.

```
classdef MyHandle < handle
    events
        SomethingHappened
    end
```

```
end
```

Create a file, `ListenerTest`, on your MATLAB path that contains the following `TestCase` class.

```
classdef ListenerTest < matlab.unittest.TestCase
    methods(Test)
        function testDisabledListeners(testCase)
            h = MyHandle;

            % Add a listener to a test helper method
            listener = h.addlistener('SomethingHappened', ...
                @testCase.shouldNotGetCalled);

            % Passing scenario (code path is not reached)
            %%%%%%%%%%%%%%%%%%%%%
            % Disabled listener should not invoke callbacks
            listener.Enabled = false;
            h.notify('SomethingHappened');

            % Failing scenario (code path is reached)
            %%%%%%%%%%%%%%%%%%%%%
            % Enabled listener invoke callback and fail
            listener.Enabled = true;
            h.notify('SomethingHappened');
        end
    end

    methods
        function shouldNotGetCalled(testCase, ~, ~)
            % A test helper callback method that should not execute
            testCase.verifyFail('This listener callback should not have executed');
        end
    end

end
```

From the command prompt, run the test.

```
run(ListenerTest);
```

```
Running ListenerTest

================================================================================
Verification failed in ListenerTest/testDisabledListeners.
```

```
    ----------------
    Test Diagnostic:
    ----------------
    This listener callback should not have executed

    ------------------
    Stack Information:
    ------------------
    In C:\Desktop\ListenerTest.m (ListenerTest.shouldNotGetCalled) at 27
    In C:\\Desktop\ListenerTest.m (@(varargin)testCase.shouldNotGetCalled(varargin{:}))
    In C:\Desktop\ListenerTest.m (ListenerTest.testDisabledListeners) at 20
================================================================================
.
Done ListenerTest
__________

Failure Summary:

     Name                                Failed  Incomplete  Reason(s)
    ================================================================================
     ListenerTest/testDisabledListeners    X                 Failed by verification.
```

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints` | `matlab.unittest.qualifications`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyFalse

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is false

Syntax

```
verifyFalse(verifiable,actual)
verifyFalse( ___ ,diagnostic)
```

Description

`verifyFalse(verifiable,actual)` verifies that actual is a scalar logical with the value of false.

`verifyFalse( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  verifiable.verifyThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `verifyThat`.
- Unlike `verifyTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `verifyTrue`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test MATLAB Logical Functions

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test `true`.

```
verifyFalse(testCase, true);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must evaluate to "false".

Actual Value:
         1
```

Test failed.

Test `false`.

```
verifyFalse(testCase, false);
```

```
Interactive verification passed.
```

Test the Value 0

The number `0` is a `double` value, not a logical value.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifyFalse(testCase, 0);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must be logical. It is of type "double".

Actual Value:
         0
```

Test failed.

Test Array of Logical Values

To be false, the value must be scalar.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyFalse(testCase, [false false false]);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     0     0
```

Test failed.

Test an array of mixed logical values.

```
verifyFalse(testCase, [false true false], ...
    'A mixed array of logicals is not the one false value');

Interactive verification failed.

---------------
Test Diagnostic:
---------------
```

```
A mixed array of logicals is not the one false value

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     1     0
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IsFalse` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyThat` | `verifyTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyGreaterThan

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is greater than specified value

Syntax

```
verifyGreaterThan(verifiable,actual,floor)
verifyGreaterThan( ___ ,diagnostic)
```

Description

`verifyGreaterThan(verifiable,actual,floor)` verifies that all elements of actual are greater than all the elements of floor.

`verifyGreaterThan( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  matlab.unittest.constraints.IsGreaterThan;
  verifiable.verifyThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify `3` is greater than `2`.

```
verifyGreaterThan(testCase, 3, 2);
```

```
Interactive verification passed.
```

Test if `5` is greater than `9`.

```
verifyGreaterThan(testCase, 5, 9);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThan failed.
--> The value must be greater than the minimum value.

Actual Value:
         5
Minimum Value (Exclusive):
         9
```

Test failed.

Compare an Array to a Scalar

Test if each element is greater than the FLOOR value, `2`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThan(testCase, [5 6 7], 2);

Interactive verification passed.
```

Test if value `5` is greater than each element in the FLOOR array, `[1 2 3]`.

```
verifyGreaterThan(testCase, 5, [1 2 3]);

Interactive verification passed.
```

Test if each element in the matrix is greater than the FLOOR value, `4`.

```
verifyGreaterThan(testCase, [1 2 3; 4 5 6], 4);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThan failed.
--> Each element must be greater than the minimum value.

    Failing Indices:
              1     2     3     5

Actual Value:
         1     2     3
         4     5     6
Minimum Value (Exclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is greater than each corresponding element of the FLOOR array, `[4 -9 0]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThan(testCase, [5 -3 2], [4 -9 0]);

Interactive verification passed.
```

Compare an array to itself.

```
verifyGreaterThan(testCase, eye(2), eye(2));

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThan failed.
--> Each element must be greater than each corresponding element of the minimum value a

    Failing Indices:
              1     2     3     4

Actual Value:
         1     0
         0     1
Minimum Value (Exclusive):
         1     0
         0     1
```

Test failed.

See Also

`matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic` | `gt` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyGreaterThanOrEqual` | `verifyLessThan` | `verifyLessThanOrEqual` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyGreaterThanOrEqual

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is greater than or equal to specified value

Syntax

```
verifyGreaterThanOrEqual(verifiable,actual,floor)
verifyGreaterThanOrEqual( ___ ,diagnostic)
```

Description

`verifyGreaterThanOrEqual(verifiable,actual,floor)` that all elements of actual are greater than or equal to all the elements of floor.

`verifyGreaterThanOrEqual( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  verifiable.verifyThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as floor unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify `3` is greater than `2`.

```
verifyGreaterThanOrEqual(testCase, 3, 2);
```

```
Interactive verification passed.
```

Verify `3` is greater than or equal to `3`.

```
verifyGreaterThanOrEqual(testCase, 3, 3);
```

```
Interactive verification passed.
```

Test if `5` is greater than `9`.

```
verifyGreaterThanOrEqual(testCase, 5, 9);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThanOrEqual failed.
--> The value must be greater than or equal to the minimum value.

Actual Value:
         5
Minimum Value (Inclusive):
         9
```

Test failed.

Compare an Array to a Scalar

Test if each element is greater than or equal to the FLOOR value, `2`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThanOrEqual(testCase, [5 2 7], 2);
```

```
Interactive verification passed.
```

Test if each element in the matrix is greater than or equal to the FLOOR value, `4`.

```
verifyGreaterThanOrEqual(testCase, [1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThanOrEqual failed.
--> Each element must be greater than or equal to the minimum value.

    Failing Indices:
              1     3     5

Actual Value:
         1     2     3
         4     5     6
Minimum Value (Inclusive):
         4
```

Compare Arrays

Test if each element is greater than or equal to each corresponding element of the FLOOR array, `[4 -3 0]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThanOrEqual(testCase, [5 -3 2], [4 -3 0]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
verifyGreaterThanOrEqual(testCase, eye(2), eye(2));
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `ge` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyGreaterThan` | `verifyLessThan` | `verifyLessThanOrEqual` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyInstanceOf

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is object of specified type

Syntax

```
verifyInstanceOf(verifiable,actual,className)
verifyInstanceOf(verifiable,actual,metaClass)
verifyInstanceOf( ___ ,diagnostic)
```

Description

`verifyInstanceOf(verifiable,actual,className)` verifies that actual is a MATLAB value whose class is the class specified by className.

`verifyInstanceOf(verifiable,actual,metaClass)` verifies that actual is a MATLAB value whose class is the class specified by the `meta.class` instance metaClass.

`verifyInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsInstanceOf;
  verifiable.verifyThat(actual, IsInstanceOf(className));
  verifiable.verifyThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to

completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a string.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test a Class

These interactive tests verify the class of the number, 5.

Create a `TestCase` object and the value to test.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
actvalue = 5;
```

Verify `actvalue` is an instance of class `double`.

```
verifyInstanceOf(testCase, actvalue, 'double');
```

```
Interactive verification passed.
```

Verify if `actvalue` is an instance of `char`.

```
verifyInstanceOf(testCase, 5, 'char');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
```

```
---------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        char

Actual Value:
         5
```

Test failed.

Test a Function Handle

These tests verify function handles, specified as a meta.class instance, `?function_handle`.

Create a function handle.

```
fh = @sin;
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyInstanceOf(testCase, fh, ?function_handle);
```

```
Interactive verification passed.
```

Test the function name.

```
fh = 'sin';
verifyInstanceOf(testCase, fh, ?function_handle);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        char
    Expected Type:
        function_handle
```

```
Actual Value:
    sin
```

Test failed.

Test a Derived Class

Verify that a derived class is not the same class as its base class.

Create a class, `BaseExample`.

```
classdef BaseExample
end
```

Create a derived class, `DerivedExample`.

```
classdef DerivedExample < BaseExample
end
```

Verify `DerivedExample` is an instance of `BaseExample`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
testCase.verifyInstanceOf(DerivedExample(), ?BaseExample);
```

```
Interactive verification passed.
```

Verify `BaseExample` is not an instance of `DerivedExample`.

```
testCase.verifyInstanceOf(BaseExample(), ?DerivedExample);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        BaseExample
    Expected Type:
        DerivedExample

Actual Value:
      BaseExample with no properties.
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IsInstanceOf` | `isa` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyClass` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyLength

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value has specified length

Syntax

```
verifyLength(verifiable,actual,expectedLength)
verifyLength( ___ ,diagnostic)
```

Description

`verifyLength(verifiable,actual,expectedLength)` verifies that actual is a MATLAB array whose length is expectedLength.

`verifyLength( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  verifiable.verifyThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Array Lengths

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify length of array is the expected value, 5.

```
verifyLength(testCase, ones(2, 5, 3), 5, 'User diagnostic');
```

```
Interactive verification passed.
```

Length of array is not the expected value, 3.

```
verifyLength(testCase, [2 3], 3);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
                2
    Expected Length:
                3

Actual Array:
         2     3
```

Test failed.

The length of a 2x3 array is 3.

```
verifyLength(testCase, [1 2 3; 4 5 6], 3);
```

```
Interactive verification passed.
```

Verify the length of a 2x3 array is not the number of elements, 6.

```
verifyLength(testCase, [1 2 3; 4 5 6], 6);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
             3
    Expected Length:
             6

Actual Array:
         1     2     3
         4     5     6
```

Test failed.

```
verifyLength(testCase, eye(2), 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
             2
    Expected Length:
             4

Actual Array:
```

```
     1     0
     0     1
```

Test Cell Array Lengths

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLength(testCase, {'somestring', 'someotherstring'}, 2);
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.HasLength` | `length` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyNumElements` | `verifySize` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyLessThan

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is less than specified value

Syntax

```
verifyLessThan(verifiable,actual,ceiling)
verifyLessThan( ___ ,diagnostic)
```

Description

`verifyLessThan(verifiable,actual,ceiling)` verifies that all elements of actual are less than all the elements of ceiling.

`verifyLessThan( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  verifiable.verifyThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test. actual must be the same size as ceiling unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify `2` is less than `3`.

```
verifyLessThan(testCase, 2, 3);
```

```
Interactive verification passed.
```

Test if `9` is less than `5`.

```
verifyLessThan(testCase, 9, 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThan failed.
--> The value must be less than the maximum value.

Actual Value:
         9
Maximum Value (Exclusive):
         5
```

Test failed.

Compare an Array to a Scalar

Test if each element is less than the CEILING value, `9`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThan(testCase, [5 6 7], 9);

Interactive verification passed.
```

Test if each element in the matrix is less than the CEILING value, `4`.

```
verifyLessThan(testCase, [1 2 3; 4 5 6], 4);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThan failed.
--> Each element must be less than the maximum value.

    Failing Indices:
              2     4     6

Actual Value:
         1     2     3
         4     5     6
Maximum Value (Exclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is less than each corresponding element of the CEILING array, `[7 -1 8]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThan(testCase, [5 -3 2], [7 -1 8]);

Interactive verification passed.
```

Compare an array to itself.

```
verifyLessThan(testCase, eye(2), eye(2));

Interactive verification failed.

---------------------
```

```
Framework Diagnostic:
---------------------
verifyLessThan failed.
--> Each element must be less than each corresponding element of the maximum value arra

    Failing Indices:
              1     2     3     4

Actual Value:
         1     0
         0     1
Maximum Value (Exclusive):
         1     0
         0     1
```

Test failed.

See Also

`matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic` | `lt` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyGreaterThan` | `verifyGreaterThanOrEqual` | `verifyLessThanOrEqual` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyLessThanOrEqual

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is less than or equal to specified value

Syntax

```
verifyLessThanOrEqual(verifiable,actual,ceiling)
verifyLessThanOrEqual( ___ ,diagnostic)
```

Description

`verifyLessThanOrEqual(verifiable,actual,ceiling)` verifies that all elements of actual are less than or equal to all the elements of ceiling.

`verifyLessThanOrEqual( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  verifiable.verifyThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify `2` is less than `3`.

```
verifyLessThanOrEqual(testCase, 2, 3);
```

```
Interactive verification passed.
```

Verify `3` is less than or equal to `3`.

```
verifyLessThanOrEqual(testCase, 3, 3);
```

```
Interactive verification passed.
```

Test if `9` is less than `5`.

```
verifyLessThanOrEqual(testCase, 9, 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThanOrEqual failed.
--> The value must be less than or equal to the maximum value.

Actual Value:
         9
Maximum Value (Inclusive):
         5
```

Test failed.

Compare an Array to a Scalar

Test if each element is less than or equal to the ceiling value, `7`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThanOrEqual(testCase, [5 2 7], 7);
```

```
Interactive verification passed.
```

Test if each element in the matrix is less than or equal to the ceiling value, `4`.

```
verifyLessThanOrEqual(testCase, [1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThanOrEqual failed.
--> Each element must be less than or equal to the maximum value.

    Failing Indices:
              4     6

Actual Value:
         1     2     3
         4     5     6
Maximum Value (Inclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is less than or equal to each corresponding element of the ceiling array, `[5 -3 8]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThanOrEqual(testCase, [5 -3 2], [5 -3 8]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
verifyLessThanOrEqual(testCase, eye(2), eye(2));
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `le` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyGreaterThan` | `verifyGreaterThanOrEqual` | `verifyLessThan` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyMatches

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify string matches specified regular expression

Syntax

```
verifyMatches(verifiable,actual,expression)
verifyMatches( ___ ,diagnostic)
```

Description

`verifyMatches(verifiable,actual,expression)` that actual is a string that matches the regular expression defined by expression.

`verifyMatches( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  verifiable.verifyThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The string to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for String Matches

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify that strings matches a regular expression.

```
verifyMatches(testCase, 'Some String', 'Some [Ss]tring', ...
    'My result should have matched the expression');
```

```
Interactive verification passed.
```

```
verifyMatches(testCase, 'Another string', '(Some |An)other');
```

```
Interactive verification passed.
```

```
verifyMatches(testCase, 'Another 3 strings', '^Another \d+ strings?$');
```

```
Interactive verification passed.
```

```
verifyMatches(testCase, '3 more strings', '\d+ strings?');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyMatches failed.
--> The string did not match the regular expression.

Actual String:
    3 more strings
Regular Expression:
    \d+ strings?
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.Matches` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `regexp` | `verifySubstring` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyNotEmpty

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is not empty

Syntax

```
verifyNotEmpty(verifiable,actual)
verifyNotEmpty( ___ ,diagnostic)
```

Description

`verifyNotEmpty(verifiable,actual)` verifies that actual is a non-empty MATLAB value.

`verifyNotEmpty( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  verifiable.verifyThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Non-Empty Strings

Create a `TestCase` object.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifyNotEmpty(testCase, '');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0  0].

Actual Value:
         ''
```

Test failed.

Test for Non-Empty Arrays

An array with any zero dimension is empty.

Test array `[2 3]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEmpty(testCase, [2 3]);
```

```
Interactive verification passed.
```

Test array with a zero dimension.

```
verifyNotEmpty(testCase, ones(2, 5, 0, 3));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEmpty failed.
```

```
--> The value must not be empty.
--> The value has a size of [2  5  0  3].

Actual Value:
       Empty array: 2-by-5-by-0-by-3
```

Test failed.

Test for Non-Empty Cell Arrays

A cell array of empty arrays is not empty.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEmpty(testCase, {[], [], []}, '');
```

```
Interactive verification passed.
```

Test for Non-Empty Test Suite

Test an empty object, `emptyTestSuite`.

```
emptyTestSuite = matlab.unittest.TestSuite.empty;
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEmpty(testCase, emptyTestSuite);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0  0].

Actual Value:
      0x0 TestCase array with no properties.
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IsEmpty` | `isempty` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyEmpty` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyNotEqual

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is not equal to specified value

Syntax

```
verifyNotEqual(verifiable,actual,notExpected)
verifyNotEqual( ___ ,diagnostic)
```

Description

`verifyNotEqual(verifiable,actual,notExpected)` verifies that actual is not equal to notExpected.

`verifyNotEqual( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  verifiable.verifyThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Numeric Values

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Compare a value to itself.

```
verifyNotEqual(testCase, 5, 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEqual failed.
--> NumericComparator passed.

Actual Value:
         5
Prohibited Value:
         5
```

Test failed.

Compare different number values.

```
verifyNotEqual(testCase, 4.95, 5, '4.95 should be different from 5');
```

```
Interactive verification passed.
```

Values `4.95` and `5` are not equal.

Compare values of different sizes.

```
verifyNotEqual(testCase, [5 5], 5, '[5 5] is not equal to 5');

Interactive verification passed.
```

Values are not equal.

Compare Classes

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEqual(testCase, int8(5), int16(5), 'Classes dont match');

Interactive verification passed.
```

Compare Cell Arrays

Test a cell array by comparing each element.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEqual(testCase, {'cell', struct, 5}, {'cell', struct, 5});

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEqual failed.
--> CellComparator passed.

Actual Value:
        'cell'    [1x1 struct]    [5]
Prohibited Value:
        'cell'    [1x1 struct]    [5]
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyEqual` | `verifyThat`

More About

- “Types of Qualifications”

Introduced in R2013a

verifyNotSameHandle

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is not handle to specified instance

Syntax

```
verifyNotSameHandle(verifiable,actual,notExpectedHandle)
verifyNotSameHandle( ___ ,diagnostic)
```

Description

`verifyNotSameHandle(verifiable,actual,notExpectedHandle)` verifies that actual is a different size and/or does not contain the same instances as the notExpectedHandle handle array.

`verifyNotSameHandle( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  verifiable.verifyThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Handles from Same Class

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Create a handle class, `ExampleHandle`.

```
classdef ExampleHandle < handle
end
```

Create two handle variables.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Handles point to different objects.

```
verifyNotSameHandle(testCase, h1, h2);
```

```
Interactive verification passed.
```

Show matching handle combinations.

```
verifyNotSameHandle(testCase, [h1 h2 h1], [h1 h2 h1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.
```

```
Actual Value:
      1x3 ExampleHandle array with no properties.
Handle Object:
      1x3 ExampleHandle array with no properties.
```

Test failed.

The order of the handle arguments matters.

```
verifyNotSameHandle(testCase, [h1 h2], [h2 h1]);
```

```
Interactive verification passed.
```

Test a handle with itself.

```
verifyNotSameHandle(testCase, h1, h1);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      ExampleHandle with no properties.
Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Variables are not same size.

```
verifyNotSameHandle(testCase, h2, [h2 h2]);
```

```
Interactive verification passed.
```

Variables are the same size.

```
verifyNotSameHandle(testCase, [h1 h1], [h1 h1]);
```

```
Interactive verification failed.

---------------------
```

```
Framework Diagnostic:
---------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      1x2 ExampleHandle array with no properties.
Handle Object:
      1x2 ExampleHandle array with no properties.
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IsSameHandleAs` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifySameHandle` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyNumElements

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value has specified element count

Syntax

```
verifyNumElements(verifiable,actual,expectedElementCount)
verifyNumElements( ___ ,diagnostic)
```

Description

`verifyNumElements(verifiable,actual,expectedElementCount)` verifies that actual is a MATLAB array with expectedElementCount number of elements.

`verifyNumElements( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  verifiable.verifyThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Matrices

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
n = 7;
verifyNumElements(testCase, eye(n), n^2);
```

```
Interactive verification passed.
```

```
verifyNumElements(testCase, 3, 1);
```

```
Interactive verification passed.
```

```
verifyNumElements(testCase, [1 2 3; 4 5 6], 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNumElements failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             6
    Expected Number of Elements:
             5

Actual Value:
         1     2     3
         4     5     6
```

Test failed.

Test Cell Array

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNumElements(testCase, {'SomeString', 'SomeOtherString'}, 2);
```

```
Interactive verification passed.
```

Test Structure

```
s.Field1 = 1;
s.Field2 = 2;
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNumElements(testCase, s, 2);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNumElements failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             1
    Expected Number of Elements:
             2

Actual Value:
        Field1: 1
        Field2: 2
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.HasElementCount` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `numel` | `verifyLength` | `verifySize` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyReturnsTrue

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify function returns true when evaluated

Syntax

```
verifyReturnsTrue(verifiable,actual)
verifyReturnsTrue( ___ ,diagnostic)
```

Description

`verifyReturnsTrue(verifiable,actual)` verifies that actual is a function handle that returns a scalar logical whose value is true.

`verifyReturnsTrue( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `verifyTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `verifyTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  verifiable.verifyThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the

primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test if Condition is True

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyReturnsTrue(testCase, @true);
```

```
Interactive verification passed.
```

Verify that it is true that two numbers are equal.

```
verifyReturnsTrue(testCase, @() isequal(1,1));
```

```
Interactive verification passed.
```

Verify that it is true that two letters are not the same.

```
verifyReturnsTrue(testCase, @() ~strcmp('a','b'));
```

```
Interactive verification passed.
```

Cause verification to fail by trying to verify that “false” evaluates to “true”.

```
verifyReturnsTrue(testCase, @false);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have evaluated to "true".
--> Returned value:
             0

Actual Function Handle:
        @false
```

Test failed.

Cause verification to fail by having the test specified in the function handle return a vector of logical values not a scalar logical value.

```
verifyReturnsTrue(testCase, @() strcmp('a',{'a','a'}));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have returned a scalar. The return value had a size of
--> Returned value:
             1     1

Actual Function Handle:
        @()strcmp('a',{'a','a'})
```

Test failed.

Cause verification to fail by having the test specified in the function handle return a double not a logical.

```
verifyReturnsTrue(testCase, @() exist('exist'));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have returned a logical value. It was of type "double".
--> Returned value:
             5

Actual Function Handle:
        @()exist('exist')
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.ReturnsTrue` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyThat` | `verifyTrue`

More About

- "Types of Qualifications"

Introduced in R2013a

verifySameHandle

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify two values are handles to same instance

Syntax

```
verifySameHandle(verifiable,actual,expectedHandle)
verifySameHandle( ___ ,diagnostic)
```

Description

`verifySameHandle(verifiable,actual,expectedHandle)` verifies that actual is the same size and contains the same instances as the expectedHandle handle array.

`verifySameHandle( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  verifiable.verifyThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedHandle`

The expected handle array.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Handles from Same Class

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Create a handle class, `ExampleHandle`.

```
classdef ExampleHandle < handle
end
```

Create two handle variables.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Show matching handle combinations.

```
verifySameHandle(testCase, h1, h1);
```

```
Interactive verification passed.
```

```
verifySameHandle(testCase, [h1 h1], [h1 h1]);
```

```
Interactive verification passed.
```

```
verifySameHandle(testCase, [h1 h2 h1], [h1 h2 h1]);
```

```
Interactive verification passed.
```

Handles must point to same object.

```
verifySameHandle(testCase, h1, h2);
```

```
Interactive verification failed.

--------------------
```

```
Framework Diagnostic:
---------------------
verifySameHandle failed.
--> Values do not refer to the same handle.

Actual Value:
      ExampleHandle with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Size of handle objects must match.

```
verifySameHandle(testCase, [h1 h1], h1);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySameHandle failed.
--> Sizes do not match.
     Actual Value Size           : [1  2]
     Expected Handle Object Size : [1  1]

Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Order of arguments is important.

```
verifySameHandle(testCase, [h1 h2], [h2 h1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySameHandle failed.
--> Some elements in the handle array refer to the wrong handle.
```

```
Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      1x2 ExampleHandle array with no properties.
```

Test failed.

See Also

`handle` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IsSameHandleAs` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyNotSameHandle` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifySize

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value has specified size

Syntax

```
verifySize(verifiable,actual,expectedSize)
verifySize( ___ ,diagnostic)
```

Description

`verifySize(verifiable,actual,expectedSize)` verifies that actual is a MATLAB array whose size is expectedSize.

`verifySize( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  verifiable.verifyThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Arrays

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifySize(testCase, ones(2, 5, 3), [2 5 3]);
```

```
Interactive verification passed.
```

```
verifySize(testCase, [1 2 3; 4 5 6], [2 3]);
```

```
Interactive verification passed.
```

```
verifySize(testCase, [2 3], [3 2]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             1     2
    Expected Size:
             3     2

Actual Value:
         2     3
```

Test failed.

Number of elements is not the same as size.

```
verifySize(testCase, [1 2 3; 4 5 6], [6 1]);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             2     3
    Expected Size:
             6     1

Actual Value:
         1     2     3
         4     5     6
```

Test failed.

```
verifySize(testCase, eye(2), [4 1]);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             2     2
    Expected Size:
             4     1

Actual Value:
         1     0
         0     1
```

Test failed.

Test Cell Array

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifySize(testCase, {'SomeString', 'SomeOtherString'}, [1 2]);

Interactive verification passed.
```

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.HasSize` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `size` | `verifyLength` | `verifyNumElements` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifySubstring

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify string contains specified string

Syntax

```
verifySubstring(verifiable,actual,substring)
verifySubstring( ___ ,diagnostic)
```

Description

`verifySubstring(verifiable,actual,substring)` verifies that actual is a string that contains substring.

`verifySubstring( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  verifiable.verifyThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The string to test.

Default:

`substring`

The value to match, specified as a string.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Substrings

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test that a substring is contained in a string.

```
verifySubstring(testCase, 'SomeLongString', 'Long');
```

```
Interactive verification passed.
```

Show that case matters.

```
verifySubstring(testCase, 'SomeLongString', 'lonG');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The string must contain the substring.

Actual String:
    SomeLongString
Expected Substring:
    lonG
```

Test failed.

Cause the verification to fail by testing a substring that isn't contained in the actual string.

```
verifySubstring(testCase, 'SomeLongString', 'OtherString');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The string must contain the substring.

Actual String:
    SomeLongString
Expected Substring:
    OtherString
```

Test failed.

Show that the verification will fail if the substring is longer than the actual string.

```
verifySubstring(testCase, 'SomeLongString', 'SomeLongStringThatIsLonger');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The string must contain the substring.

Actual String:
    SomeLongString
Expected Substring:
    SomeLongStringThatIsLonger
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.ContainsSubstring` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `strfind` | `verifyMatches` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyThat

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value meets given constraint

Syntax

```
verifyThat(verifiable,actual,constraint)
verifyThat( ___ ,diagnostic)
```

Description

`verifyThat(verifiable,actual,constraint)` verifies that actual is a value that satisfies the constraint provided.

If the constraint is not satisfied, a verification failure is produced utilizing only the framework diagnostic generated by the constraint.

`verifyThat( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

When using this signature, both the diagnostic information contained within diagnostic is used in addition to the diagnostic information provided by the constraint.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Tips

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is

preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Examples

Test Conditions Using Constraints

```
testCase = matlab.unittest.TestCase.forInteractiveUse;

% Passing scenarios
%%%%%%%%%%%%%%%%%%%%%%
import matlab.unittest.constraints.IsTrue
verifyThat(testCase, true, IsTrue)

import matlab.unittest.constraints.IsEqualTo
verifyThat(testCase, 5, IsEqualTo(5), '5 should be equal to 5')

import matlab.unittest.constraints.IsGreaterThan
import matlab.unittest.constraints.HasNaN
verifyThat(testCase, [5 NaN], IsGreaterThan(10) | HasNaN, ...
    'The value was not greater than 10 or NaN')


% Failing scenarios
%%%%%%%%%%%%%%%%%%%%%%
import matlab.unittest.constraints.AnyCellOf
import matlab.unittest.constraints.ContainsSubstring
verifyThat(testCase, AnyCellOf({'cell','of','strings'}), ...
    ContainsSubstring('char'),'Test description')

import matlab.unittest.constraints.HasSize
verifyThat(testCase, zeros(10,4,2), HasSize([10,5,2]), ...
    @() disp('A function handle diagnostic.'))

import matlab.unittest.constraints.IsEmpty
verifyThat(testCase, 5, IsEmpty)
```

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.Constraint` | `matlab.unittest.constraints` | `matlab.unittest.qualifications`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyTrue

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify value is true

Syntax

```
verifyTrue(verifiable,actual)
verifyTrue( ___ ,diagnostic)
```

Description

`verifyTrue(verifiable,actual)` verifies that actual is a scalar logical with the value of true.

`verifyTrue( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  verifiable.verifyThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `verifyThat`.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered

to by other constraints such as `IsEqualTo`. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of `verifyReturnsTrue`, which at least shows the display of the function evaluated to generate the failing result.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test MATLAB Logical Functions

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test `true`.

```
verifyTrue(testCase, true);
```

```
Interactive verification passed.
```

Test `false`.

```
verifyTrue(testCase, false);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyTrue failed.
--> The value must evaluate to "true".

Actual Value:
         0
```

Test failed.

Test the Value 1

The number `1` is a `double` value, not a logical value.

A double value of `1` is not `true`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyTrue(testCase, 1);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyTrue failed.
--> The value must be logical. It is of type "double".

Actual Value:
         1
```

Test failed.

Test Array of Logical Values

To be true, the value must be scalar.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyTrue(testCase, [true true true]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyTrue failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         1     1     1
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IsTrue` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyFalse` | `verifyReturnsTrue` | `verifyThat`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyWarning

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify function issues specified warning

Syntax

```
verifyWarning(verifiable,actual,warningID)
verifyWarning( ___ ,diagnostic)
[output1,...,outputN] = verifyWarning( ___ )
```

Description

`verifyWarning(verifiable,actual,warningID)` verifies that actual issues a warning with the identifier warningID.

`verifyWarning( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`[output1,...,outputN] = verifyWarning( ___ )` also returns the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  verifiable.verifyThat(actual, IssuesWarnings({warningID}));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the

primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`warningID`

Warning ID, specified as a string.

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

Test warning Function

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify actual warning ID is the same as input warning ID.

```
verifyWarning(testCase, @() warning('SOME:warning:id','Warning!'), ...
    'SOME:warning:id');
```

```
Interactive verification passed.
```

```
verifyWarning(testCase, @() warning('SOME:other:id', 'Warning message'),...
    'SOME:warning:id', 'Did not issue specified warning');
```

```
Warning: Warning message
> In @()warning('SOME:other:id','Warning message')
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 43
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 58
  In IssuesWarnings>IssuesWarnings.invoke at 364
  In IssuesWarnings>IssuesWarnings.issuesExpectedWarnings at 411
  In IssuesWarnings>IssuesWarnings.satisfiedBy at 240
```

```
  In QualificationDelegate>QualificationDelegate.qualifyThat at 90
  In QualificationDelegate>QualificationDelegate.qualifyWarning at 196
  In Verifiable>Verifiable.verifyWarning at 701
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Did not issue specified warning

---------------------
Framework Diagnostic:
---------------------
verifyWarning failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue the correct warnings.

        Missing Warnings:
             SOME:warning:id

    Actual Warning Profile:
         SOME:other:id
    Expected Warning Profile:
         SOME:warning:id

Evaluated Function:
        @()warning('SOME:other:id','Warning message')
```

Test a Function Without Warnings

Test the `true` function, which does not issue warnings.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyWarning(testCase, @true, 'SOME:warning:id', ...
'@true did not issue any warning');

Interactive verification failed.

----------------
Test Diagnostic:
----------------
```

```
@true did not issue any warning

---------------------
Framework Diagnostic:
---------------------
verifyWarning failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue any warnings.

    Expected Warning Profile:
         SOME:warning:id

Evaluated Function:
        @true
```

Test failed.

Test Function With Output Arguments

Create a `helper` function that generates a warning and returns output.

```
function varargout = helper()
    warning('SOME:warning:id','Warning!');
    varargout = {123, 'abc'};
end
```

Call `helper`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
[actualOut1, actualOut2] = verifyWarning(testCase, @helper, ...
    'SOME:warning:id');

Interactive verification passed.
```

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IssuesWarnings` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyError` | `verifyThat` | `verifyWarningFree` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

verifyWarningFree

Class: matlab.unittest.qualifications.Verifiable
Package: matlab.unittest.qualifications

Verify function issues no warnings

Syntax

```
verifyWarningFree(verifiable,actual)
verifyWarningFree( ___ ,diagnostic)
output1,...,outputN = verifyWarningFree( ___ )
```

Description

`verifyWarningFree(verifiable,actual)` verifies that actual is a function handle that issues no warnings.

`verifyWarningFree( ___ ,diagnostic)` also displays the diagnostic information in diagnostic upon a failure.

`output1,...,outputN = verifyWarningFree( ___ )` also returns the output arguments output1,...,outputN that are produced when invoking actual.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  verifiable.verifyThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the

primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from actual, returned as any type. The argument type is specified by the actual argument list.

Examples

Test for Warnings from MATLAB Functions

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test the `why` function.

```
verifyWarningFree(testCase, @why);
```

```
The bald and not excessively bald and not excessively smart hamster obeyed a terrified
Interactive verification passed.
```

This is a randomly-generated message.

Test the `true` function.

```
verifyWarningFree(testCase, @true);
```

```
Interactive verification passed.
```

Test the `false` function.

```
actualOutputFromFalse = verifyWarningFree(testCase, @false);
```

```
Interactive verification passed.
```

Test a value that is not a function handle.

```
verifyWarningFree(testCase, 5,'diagnostic');

Interactive verification failed.

----------------
Test Diagnostic:
----------------
diagnostic

---------------------
Framework Diagnostic:
---------------------
verifyWarningFree failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
```

Test failed.

Test a function that generates warning.

```
verifyWarningFree(testCase, @() warning('some:id', 'Message'));

Warning: Message
> In @()warning('some:id','Message')
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 43
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 58
  In IssuesNoWarnings>IssuesNoWarnings.issuesNoWarnings at 131
  In IssuesNoWarnings>IssuesNoWarnings.satisfiedBy at 82
  In QualificationDelegate>QualificationDelegate.qualifyThat at 90
  In QualificationDelegate>QualificationDelegate.qualifyWarningFree at 204
  In Verifiable>Verifiable.verifyWarningFree at 757
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyWarningFree failed.
--> The function issued warnings.
```

```
    Warnings Issued:
         some:id

Evaluated Function:
        @()warning('some:id','Message')
```

Test failed.

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.constraints.IssuesNoWarnings` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `verifyThat` | `verifyWarning` | `warning`

More About

- "Types of Qualifications"

Introduced in R2013a

quarter

Quarter number

Syntax

```
q = quarter(t)
```

Description

`q = quarter(t)` returns the quarter numbers for the datetime values in `t`. The `q` output is a `double` array containing integer values from 1 to 4, and is the same size as `t`.

Examples

Find Quarter Number of Dates

```
t = datetime(2013,05,31):calmonths(3):datetime(2014,05,31)
```

```
t =

   31-May-2013   31-Aug-2013   30-Nov-2013   28-Feb-2014   31-May-2014
```

```
q = quarter(t)
```

```
q =

     2     3     4     1     2
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

`day` | `month` | `week` | `year`

Introduced in R2014b

questdlg

Create question dialog box

Syntax

```
button = questdlg('qstring')
button = questdlg('qstring','title')
button = questdlg('qstring','title',default)
button = questdlg('qstring','title','str1','str2',default)
button = questdlg('qstring','title','str1','str2','str3',default)
button = questdlg('qstring','title', ..., options)
```

Description

`button = questdlg('qstring')` displays a modal dialog box presenting the question `'qstring'`. The dialog has three default buttons, **Yes**, **No**, and **Cancel**. If the user presses one of these three buttons, `button` is set to the name of the button pressed. If the user presses the close button on the dialog without making a choice, `button` is set to the empty string. If the user presses the **Return** key, button is set to `'Yes'`. `'qstring'` is a cell array or a string that automatically wraps to fit within the dialog box.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`button = questdlg('qstring','title')` displays a question dialog with `'title'` displayed in the dialog's title bar.

`button = questdlg('qstring','title',default)` specifies which push button is the default in the event that the **Return** key is pressed. `'default'` must be `'Yes'`, `'No'`, or `'Cancel'`.

`button = questdlg('qstring','title','str1','str2',default)` creates a question dialog box with two push buttons labeled `'str1'` and `'str2'`. *`default`* specifies the default button selection and must be `'str1'` or `'str2'`.

`button = questdlg('qstring','title','str1','str2','str3',default)` creates a question dialog box with three push buttons labeled `'str1'`, `'str2'`, and `'str3'`. *`default`* specifies the default button selection and must be `'str1'`, `'str2'`, or `'str3'`.

When *`default`* is specified, but is not set to one of the button names, pressing the **Enter** key displays a warning and the dialog remains open.

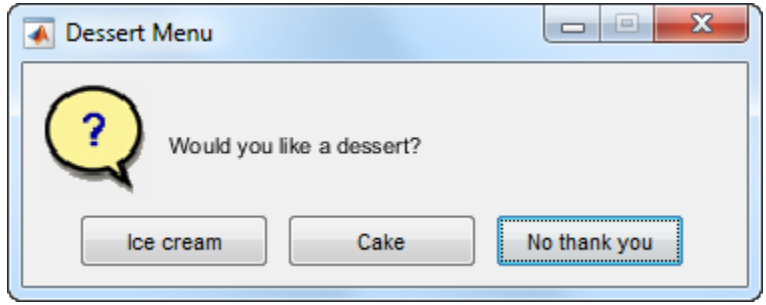
`button = questdlg('qstring','title', ..., options)` replaces the string *`default`* with a structure, `options`. The structure specifies which button string is the default answer, and whether to use TeX to interpret the question string, `qstring`. Button strings and dialog titles cannot use TeX interpretation. The `options` structure must include the fields `Default` and `Interpreter`, both strings. It can include other fields, but `questdlg` does not use them. You can set `Interpreter` to `'none'` or `'tex'`. If the `Default` field does not contain a valid button name, a command window warning is issued and the dialog box does not respond to pressing the **Enter** key.

Examples

Example 1

Create a dialog that requests a dessert preference and encode the resulting choice as an integer.

```
% Construct a questdlg with three options
choice = questdlg('Would you like a dessert?', ...
 'Dessert Menu', ...
 'Ice cream','Cake','No thank you','No thank you');
% Handle response
switch choice
    case 'Ice cream'
        disp([choice ' coming right up.'])
        dessert = 1;
    case 'Cake'
        disp([choice ' coming right up.'])
        dessert = 2;
    case 'No thank you'
        disp('I''ll bring you your check.')
        dessert = 0;
end
```



To access the return value assigned to `dessert`, save the example as a function, for example `choosedessert`, by inserting this line on top:

```
function dessert = choosedessert
```

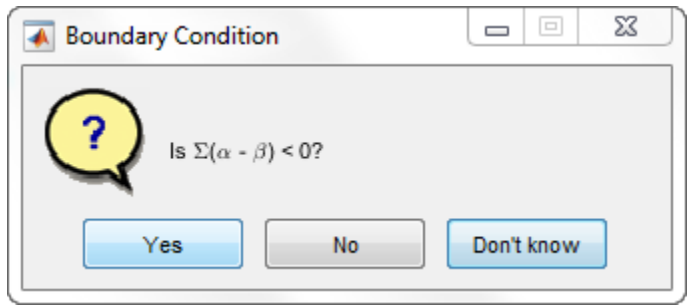
You can generalize the function by providing the cases as string or cell array calling arguments.

As the example shows, `case` statements can contain white space (but are case-sensitive).

Example 2

Specify an options structure to use the TeX interpreter to format a question.

```
options.Interpreter = 'tex';
% Include the desired Default answer
options.Default = 'Don''t know';
% Create a TeX string for the question
qstring = 'Is \Sigma(\alpha - \beta) < 0?';
choice = questdlg(qstring,'Boundary Condition',...
    'Yes','No','Don''t know',options)
```



See Also

`dialog` | `errordlg` | `helpdlg` | `inputdlg` | `listdlg` | `msgbox` | `warndlg` | `figure` | `textwrap` | `uiwait` | `uiresume`

Introduced before R2006a

quit

Terminate MATLAB program

Alternatives

As an alternative to the quit function, use the Close box in the MATLAB desktop.

Syntax

```
quit
quit cancel
quit force
```

Description

`quit` displays a confirmation dialog box if the confirm upon quitting preference is selected, and if confirmed or if the confirmation preference is not selected, terminates MATLAB after running `finish.m`, if `finish.m` exists. Call `quit` from the MATLAB command prompt. To interrupt a MATLAB command, see "Stop Execution".

The workspace is not automatically saved by `quit`. To save the workspace or perform other actions when quitting, create a `finish.m` file to perform those actions. For example, you can display a custom dialog box to confirm quitting using a `finish.m` file—see the following examples for details. If an error occurs while `finish.m` is running, `quit` is canceled so that you can correct your `finish.m` file without losing your workspace.

`quit cancel` is for use in `finish.m` and cancels quitting. It has no effect anywhere else.

`quit force` bypasses `finish.m` and terminates MATLAB. Use this to override `finish.m`, for example, if an errant `finish.m` will not let you quit.

Examples

Two sample `finish.m` files are included with MATLAB. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finishsav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m`—Displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code:

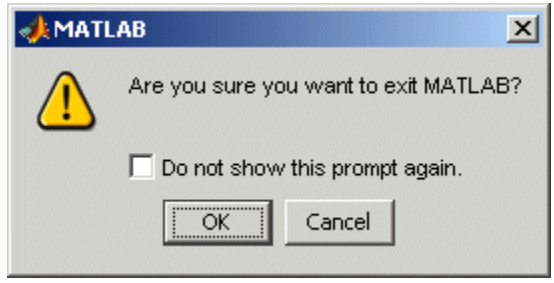
```
button = questdlg('Ready to quit?', ...
        'Exit Dialog','Yes','No','No');
switch button
            case 'Yes',
              disp('Exiting MATLAB');
              %Save variables to matlab.mat
              save
            case 'No',
              quit cancel;
end
```

More About

Tips

When using Handle Graphics objects in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

If you want MATLAB to display the following confirmation dialog box after running quit, select **Preferences** in the **Environment** section on the **Home** tab. Then select the check box for Confirm before exiting MATLAB, and click **OK**.



- “Stop Execution”

See Also

`exit` | `save` | `finish` | `startup`

Introduced before R2006a

Quit (COM)

Terminate MATLAB Automation server

Syntax

IDL Method Signature

```
void Quit(void)
```

Microsoft Visual Basic Client

```
Quit
```

MATLAB Client

```
Quit(h)
```

Description

`Quit(h)` terminates the MATLAB server session attached to handle `h`. The MATLAB object is active until all references have been released, such as when the variable in a function call goes out of scope, or by calling the MATLAB `clear h` command.

The function name is case-sensitive.

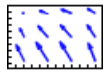
To release the MATLAB object, type:

```
clear h
```

Introduced before R2006a

quiver

Quiver or velocity plot



Syntax

```
quiver(x,y,u,v)
quiver(u,v)
quiver(...,scale)
quiver(...,LineSpec)
quiver(...,LineSpec,'filled')
quiver(...,'PropertyName',PropertyValue,...)
quiver(axes_handle,...)
h = quiver(...)
```

Description

A quiver plot displays velocity vectors as arrows with components `(u,v)` at the points `(x,y)`.

For example, the first vector is defined by components `u(1)`,`v(1)` and is displayed at the point `x(1)`,`y(1)`.

`quiver(x,y,u,v)` plots vectors as arrows at the coordinates specified in each corresponding pair of elements in `x` and `y`. The matrices `x`, `y`, `u`, and `v` must all be the same size and contain corresponding position and velocity components. However, `x` and `y` can also be vectors, as explained in the next section. By default, the arrows are scaled to just not overlap, but you can scale them to be longer or shorter if you want.

`quiver(u,v)` draws vectors specified by `u` and `v` at equally spaced points in the *x*-*y* plane.

`quiver(...,scale)` automatically scales the arrows to fit within the grid and then stretches them by the factor `scale`. `scale = 2` doubles their relative length, and `scale`

= 0.5 halves the length. Use scale = 0 to plot the velocity vectors without automatic scaling. You can also tune the length of arrows after they have been drawn by choosing the Plot Edit  tool, selecting the quiver object, opening the Property Editor, and adjusting the **Length** slider.

quiver(...,LineSpec) specifies line style, marker symbol, and color using any valid LineSpec. quiver draws the markers at the origin of the vectors.

quiver(...,LineSpec,'filled') fills markers specified by LineSpec.

quiver(...,'*PropertyName*',PropertyValue,...) specifies property name and property value pairs for the quiver objects the function creates.

quiver(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes (gca).

h = quiver(...) returns the handle to the quiver object.

Expanding x- and y-Coordinates

MATLAB expands x and y if they are not matrices. This expansion is equivalent to calling meshgrid to generate matrices from vectors:

```
[x,y] = meshgrid(x,y);
quiver(x,y,u,v)
```

In this case, the following must be true:

length(x) = n and length(y) = m, where [m,n] = size(u) = size(v).

The vector x corresponds to the columns of u and v, and vector y corresponds to the rows of u and v.

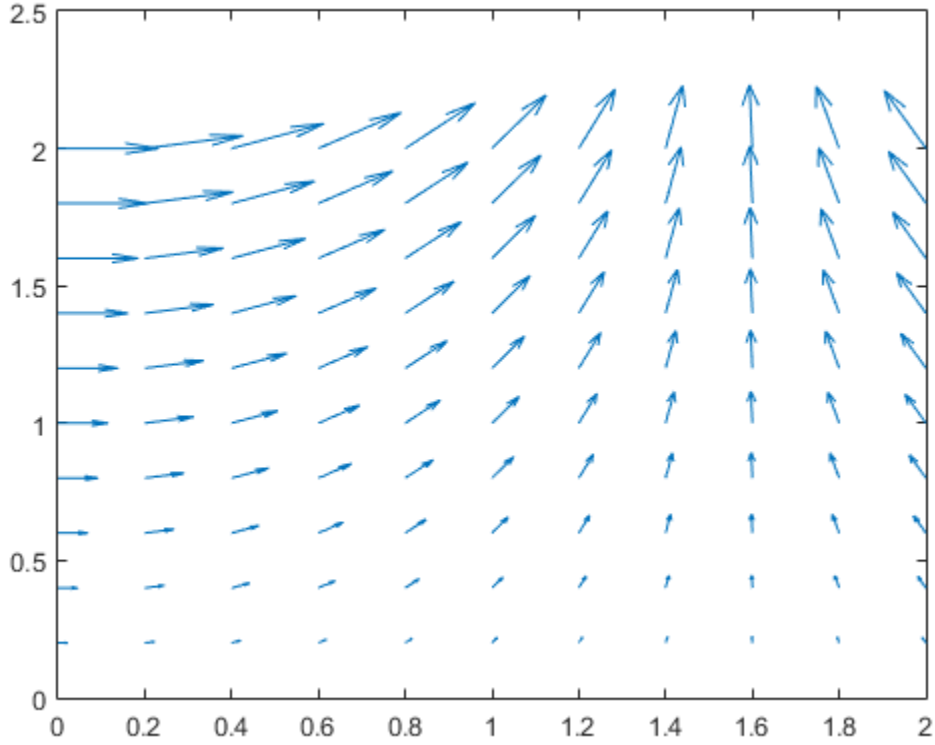
Examples

Plot Vector Velocities

Use quiver to display an arrow at each data point in x and y such that the arrow direction and length represent the corresponding values in u and v.

```
[x,y] = meshgrid(0:0.2:2,0:0.2:2);
u = cos(x).*y;
```

```
v = sin(x).*y;

figure
quiver(x,y,u,v)
```

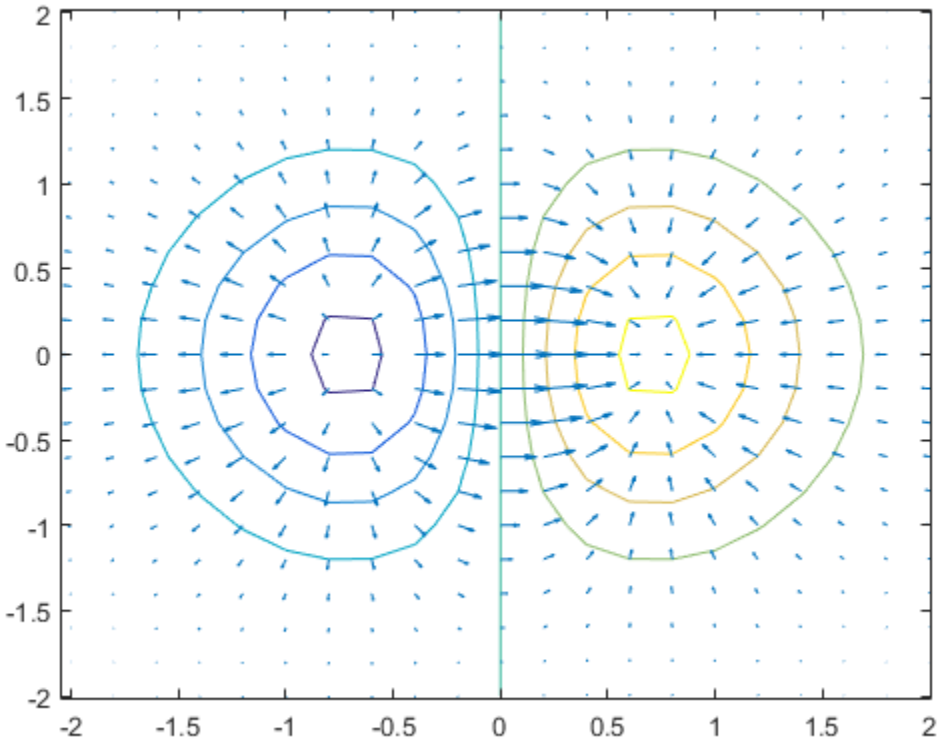


Show Gradient with Quiver Plot

Plot the gradient of the function $z = xe^{-x^2-y^2}$.

```
[X,Y] = meshgrid(-2:.2:2);
Z = X.*exp(-X.^2 - Y.^2);
[DX,DY] = gradient(Z,.2,.2);

figure
```

```
contour(X,Y,Z)
hold on
quiver(X,Y,DX,DY)
hold off
```



More About

- "Display Quiver Plot Over Contour Plot"

See Also

Functions

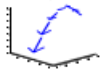
`contour` | `LineSpec` | `plot` | `quiver3`

Properties
Quiver Series Properties

Introduced before R2006a

quiver3

3-D quiver or velocity plot



Syntax

```
quiver3(x,y,z,u,v,w)
quiver3(z,u,v,w)
quiver3(...,scale)
quiver3(...,LineSpec)
quiver3(...,LineSpec,'filled')
quiver3(...,'PropertyName',PropertyValue,...)
quiver3(axes_handle,...)
h = quiver3(...)
```

Description

A three-dimensional quiver plot displays vectors with components `(u,v,w)` at the points `(x,y,z)`, where `u`, `v`, `w`, `x`, `y`, and `z` all have real (non-complex) values.

`quiver3(x,y,z,u,v,w)` plots vectors with directions determined by components `(u,v,w)` at points determined by `(x,y,z)`. The matrices `x`,`y`,`z`,`u`,`v`, and `w` must all be the same size and contain the corresponding position and vector components.

`quiver3(z,u,v,w)` plots vectors with directions determined by components `(u,v,w)` at equally spaced points along the surface `z`. For each vector `(u(i,j),v(i,j),w(i,j))`, the column index `j` determines the *x*-value of the point on the surface, the row index `i` determines the *y*-value, and `z(i,j)` determines the *z*-value. That is, `quiver3` locates the vector at the point on the surface `(j,i,z(i,j))`. The `quiver3` function automatically scales the vectors to prevent overlapping based on the distance between them.

`quiver3(...,scale)` automatically scales the vectors to prevent them from overlapping, and then multiplies them by `scale`. `scale = 2` doubles their relative length, and `scale = 0.5` halves them. Use `scale = 0` to plot the vectors without the automatic scaling.

`quiver3(...,LineSpec)` specifies line style, marker symbol, and color using any valid `LineSpec`. `quiver3` draws the markers at the origin of the vectors.

`quiver3(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

`quiver3(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the quiver series that the function creates.

`quiver3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = quiver3(...)` returns the quiver series handle.

Examples

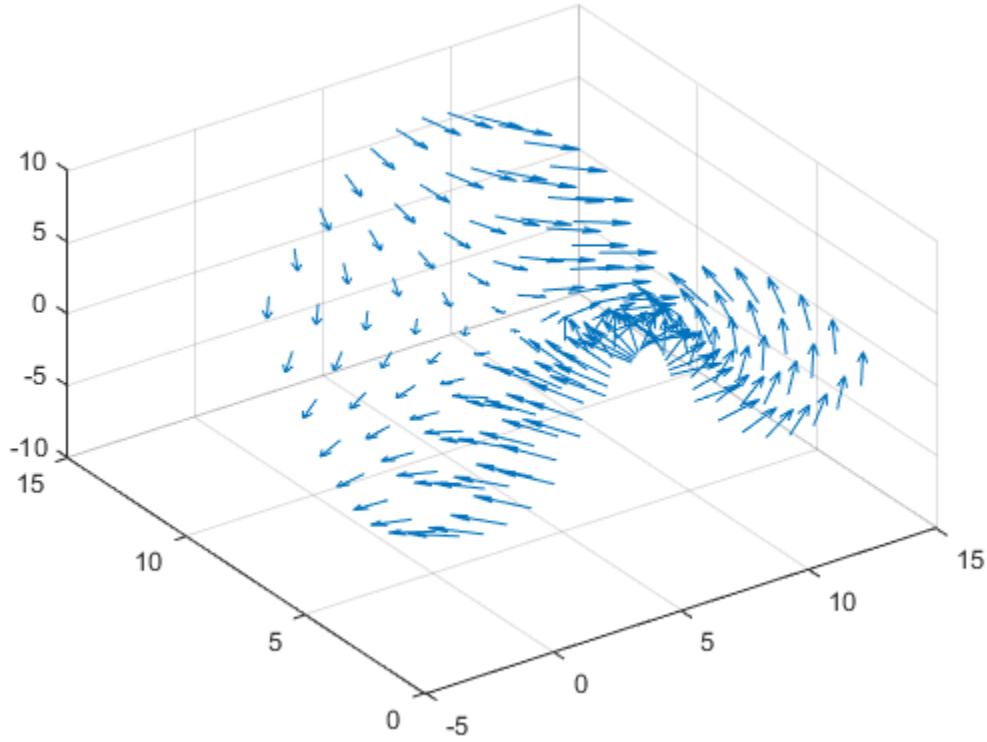
Create 3-D Quiver Plot

Define the data.

```
x = -3:0.5:3;
y = -3:0.5:3;
[X,Y] = meshgrid(x, y);
Z = Y.^2 - X.^2;
[U,V,W] = surfnorm(Z);
```

Plot vectors with components `(U,V,W)` at points that are equally spaced in the *x*-direction and *y*-direction with heights determined by `Z`.

```
figure
quiver3(Z,U,V,W)
view(-35,45)
```

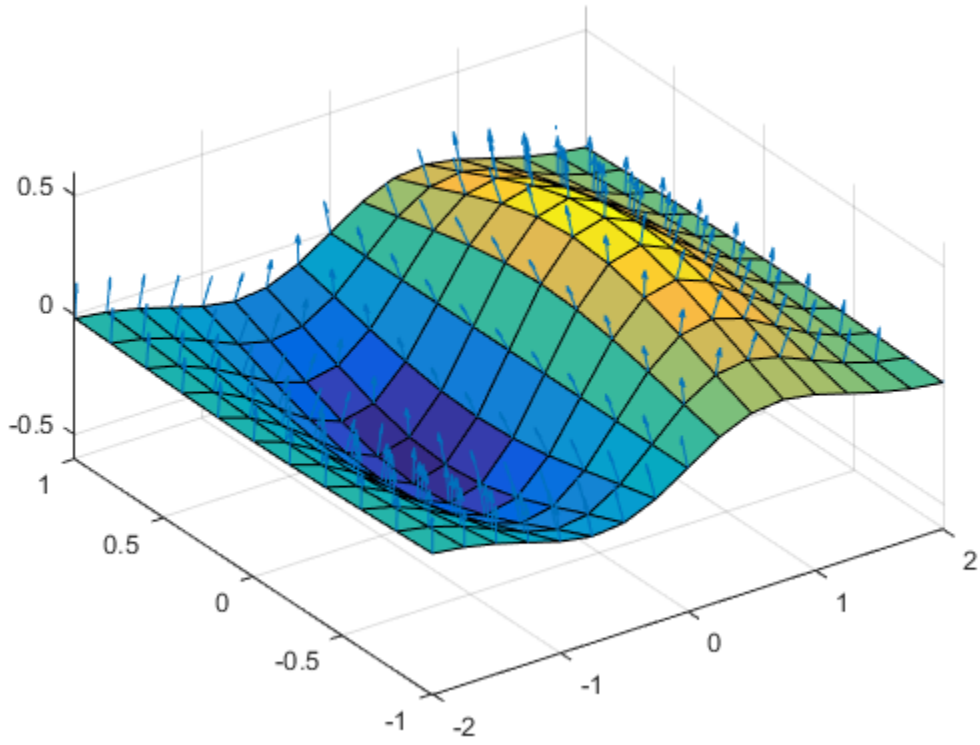


Plot Surface Normals

Plot the surface normals of the function $z = xe^{-x^2-y^2}$.

```
[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);

figure
quiver3(X,Y,Z,U,V,W,0.5)

hold on
surf(X,Y,Z)
view(-35,45)
```

```
axis([-2 2 -1 1 -.6 .6])
hold off
```



More About

- "Projectile Path Over Time"

See Also

Functions

`axis` | `contour` | `LineSpec` | `plot` | `plot3` | `quiver` | `surfnorm` | `view`

Properties
Quiver Series Properties

Introduced before R2006a

Quiver Series Properties

Control quiver series appearance and behavior

Quiver series properties control the appearance and behavior of a quiver series object. By changing property values, you can modify certain aspects of the quiver series.

Starting in R2014b, you can use dot notation to query and set properties.

```
q = quiver(1:10,1:10);
c = q.Color;
q.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Arrows

`Color` — Arrow color

`[0 0 1]` (default) | RGB triplet | color string | `'none'`

Arrow color, specified as a three-element RGB triplet, a color string, or `'none'`. If you set the color to `'none'`, then the arrow is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`

Long Name	Short Name	RGB Triplet
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

LineStyle — Style of arrow stem

'-' (default) | '--' | ':' | '-.' | 'none'

Style of arrow stem, specified as one of the strings listed in this table.

String	Line Style	Result
'-'	Solid line	————
'--'	Dashed line	- - - - -
':'	Dotted line	.....
'-.'	Dash-dotted line	—·—·—·—
'none'	No stem	No stem

LineWidth — Width of arrow stem

0.5 (default) | scalar numeric value

Width of arrow stem, specified as a scalar numeric value greater than zero in point units. One point equals 1/72 inch. The default value is 0.5 point.

Example: 0.75

ShowArrowHead — Arrowhead display

'on' (default) | 'off'

Arrowhead display, specified as one of these values:

- 'on' — Display the vectors with arrowheads.
- 'off' — Display the vectors without arrowheads.

`MaxHeadSize` — Maximum size of arrowhead

`0.2` (default) | scalar

Maximum size of arrowhead, specified as a scalar value in units relative to the length of the arrow.

Example: `0.1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`AutoScale` — Automatic scaling of arrow length

`'on'` (default) | `'off'`

Automatic scaling of arrow length, specified as one of these values:

- `'on'` — Scale the arrow length to fit within the grid-defined coordinate data and scale arrows so that they do not overlap. The `quiver` function then applies the `AutoScaleFactor` to the arrow length.
- `'off'` — Do not scale the arrow lengths.

`AutoScaleFactor` — Scale factor

`0.9` (default) | scalar

Scale factor, specified as a scalar. A value of `2` doubles the length of the arrows. A value of `0.5` halves the arrow lengths.

This property has an effect only if the `AutoScale` property is set to `'on'`.

Example: `2`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.

- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the quiver series object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Data

`UData` — Vector lengths in x-direction

vector | matrix

Vector lengths in *x*-direction, specified as a vector or a matrix. The `UData`, `VData`, and `WData` properties together specify the components of the vectors displayed as arrows in the quiver chart.

Example: `1:10`

`VData` — Vector lengths in y-direction

vector | matrix

Vector lengths in *y*-direction, specified as a vector or a matrix. The `UData`, `VData`, and `WData` properties together specify the components of the vectors displayed as arrows in the quiver chart.

Example: `1:10`

`WData` — Vector lengths in z-direction

vector | matrix

Vector lengths in *z*-direction, specified as a vector or a matrix. The `UData`, `VData`, and `WData` properties together specify the components of the vectors displayed as arrows in the quiver chart. For 2-D quiver charts, `WData` is an empty array.

Example: `1:10`

`XData` — x-coordinates

vector | matrix

x-coordinates, specified as a vector or matrix. The input argument `X` to the `quiver` function determines the *x*-coordinates. If you do not specify `X`, then `quiver` uses the indices of `UData` as the *x*-coordinates. `XData` must be equal in size to `YData`.

Setting this property sets the associated mode property to manual mode.

Example: `1:10`

`YData` — y-coordinates

vector | matrix

y-coordinates, specified as a vector or matrix. The input argument `Y` to the `quiver` function determines the *y*-coordinates. If you do not specify `Y`, then `quiver` uses the indices of `VData` as the *y*-coordinates. `YData` must be equal in size to `XData`.

Setting this property sets the associated mode property to manual mode.

Example: `1:10`

`ZData` — z-coordinates

vector | matrix

z-coordinates, specified as a vector or matrix. The input argument `Z` to the `quiver3` function determines the *z*-coordinates. For 2-D quiver charts, `ZData` is an empty array. For 3-D quiver charts, `ZData` must be equal in size to `XData` and `YData`.

Example: `1:10`

`UDataSource` — Variable linked to `UData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `UData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `UData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, MATLAB does not update the `UData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`VDataSource` — Variable linked to `VData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `VData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `VData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, MATLAB does not update the `VData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`WDataSource` — Variable linked to `WData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `WData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `WData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, MATLAB does not update the `WData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`XDataSource` — Variable linked to `XData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZDataSource` — Variable linked to `ZData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `ZData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

`XDataMode` — Selection mode for `XData`

`'auto' | 'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Automatically select the values.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or use the input argument `X` to the function.

`YDataMode` — Selection mode for `YData`

`'auto' | 'manual'`

Selection mode for `YData`, specified as one of these values:

- `'auto'` — Automatically select the values.
- `'manual'` — Use manually specified values. To specify the values, set the `YData` property or use the input argument `Y` to the function.

Visibility

`Visible` — Visibility of quiver series

`'on'` (default) | `'off'`

Visibility of quiver series, specified as one of these values:

- `'on'` — Display the quiver series.
- `'off'` — Hide the quiver series without deleting it. You still can access the properties of an invisible quiver series object.

`Clipping` — Clipping of quiver series to axes limits

`'on'` (default) | `'off'`

Clipping of quiver series to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the quiver series that are outside the axes limits.
- `'off'` — Display the entire quiver series, even if parts of it appear outside the axes limits. Parts of the quiver series might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the quiver series that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.

- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'quiver'`

Type of graphics object, returned as `'quiver'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

`Tag` — User-specified tag

`''` (default) | string

Tag to associate with the quiver series, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with quiver series

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the quiver series object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the quiver series.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the quiver series object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the quiver series from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the quiver series object in the legend as one entry (default).
   - `'off'` — Do not include the quiver series object in the legend.
   - `'children'` — Include only children of the quiver series object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of quiver series

axes object | group object | transform object

Parent of quiver series, specified as an axes, group, or transform object.

`Children` — Children of quiver series

empty `GraphicsPlaceholder` array

The quiver series has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of quiver series object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The quiver series object handle is always visible.
- `'off'` — The quiver series object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The quiver series object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the quiver series at the command-line, but allows callback functions to access it.

If the quiver series object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the quiver series. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The quiver series object — You can access properties of the quiver series object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the quiver series. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the quiver series when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the quiver series.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the quiver series that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the quiver series responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the quiver series passes the click to the object below it in the current view of the figure window. The HitTest property of the quiver series has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the quiver series. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the quiver series that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the quiver series object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for quiver series

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for quiver series, specified as one of these values:

- `'off'` — Click the quiver series plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the quiver series plot to select it, that is, anywhere within the rectangle that encloses the quiver series plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the quiver series is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the quiver series tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the quiver series. Setting the `CreateFcn` property on an existing quiver series has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during quiver series creation. MATLAB executes the callback after creating the quiver series and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The quiver series object — You can access properties of the quiver series object from within the callback function. You also can access the quiver series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the quiver series. MATLAB executes the callback before destroying the quiver series so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The quiver series object — You can access properties of the quiver series object from within the callback function. You also can access the quiver series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of quiver series

`'off'` (default) | `'on'`

Deletion status of quiver series, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the quiver series begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the quiver series no longer exists.

Check the value of the BeingDeleted property to verify that the quiver series is not about to be deleted before querying or modifying it.

See Also

`quiver` | `quiver3`

More About

- “Access Property Values”
- “Graphics Object Properties”

qz

QZ factorization for generalized eigenvalues

Syntax

```
[AA,BB,Q,Z] = qz(A,B)
[AA,BB,Q,Z,V,W] = qz(A,B)
qz(A,B,flag)
```

Description

The `qz` function gives access to intermediate results in the computation of generalized eigenvalues.

`[AA,BB,Q,Z] = qz(A,B)` for square matrices `A` and `B`, produces upper quasitriangular matrices `AA` and `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA`, and `Q*B*Z = BB`. For complex matrices, `AA` and `BB` are triangular.

`[AA,BB,Q,Z,V,W] = qz(A,B)` also produces matrices `V` and `W` whose columns are generalized eigenvectors.

`qz(A,B,flag)` for real matrices `A` and `B`, produces one of two decompositions depending on the value of `flag`:

`'complex'`	Produces a possibly complex decomposition with a triangular `AA`. For compatibility with earlier versions, `'complex'` is the default.
`'real'`	Produces a real decomposition with a quasitriangular `AA`, containing 1-by-1 and 2-by-2 blocks on its diagonal.

If `AA` is triangular, then the diagonal elements `a = diag(AA)` and `b = diag(BB)` are the generalized eigenvalues that satisfy

```
A*V*b = B*V*a
b'*W'*A = a'*W'*B
```

The eigenvalues produced by `lambda = eig(A,B)` are the ratios of the diagonal elements `a` and `b`, such that `lambda = a./b`.

If `AA` is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

See Also

`eig`

Introduced before R2006a

rand

Uniformly distributed random numbers

Syntax

```
X = rand
X = rand(n)
X = rand(sz1,...,szN)
X = rand(sz)

X = rand(___,typename)
X = rand(___,'like',p)
```

Description

`X = rand` returns a single uniformly distributed random number between 0 and 1.

`X = rand(n)` returns an `n`-by-`n` matrix of random numbers.

`X = rand(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of random numbers where `sz1,...,szN` indicate the size of each dimension. For example, `rand(3,4)` returns a 3-by-4 matrix.

`X = rand(sz)` returns an array of random numbers where size vector `sz` specifies `size(X)`. For example, `rand([3 4])` returns a 3-by-4 matrix.

`X = rand(___,typename)` returns an array of random numbers of data type `typename`. The `typename` input can be either `'single'` or `'double'`. You can use any of the input arguments in the previous syntaxes.

`X = rand(___,'like',p)` returns an array of random numbers like `p`; that is, of the same object type as `p`. You can specify either `typename` or `'like'`, but not both.

The sequence of numbers produced by `rand` is determined by the internal settings of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. You can control that shared random number generator using `rng`.

Note: Use the `rng` function instead of `rand` or `randn` with the `'seed'`, `'state'`, or `'twister'` inputs. For more information, see " Replace Discouraged Syntaxes of rand and randn"

Examples

Matrix of Random Numbers

Generate a 5-by-5 matrix of uniformly distributed random numbers between 0 and 1.

```
r = rand(5)
```

```
r =

    0.5468    0.6791    0.8852    0.3354    0.6538
    0.5211    0.3955    0.9133    0.6797    0.4942
    0.2316    0.3674    0.7962    0.1366    0.7791
    0.4889    0.9880    0.0987    0.7212    0.7150
    0.6241    0.0377    0.2619    0.1068    0.9037
```

Random Numbers Within Specified Interval

Generate a 10-by-1 column vector of uniformly distributed numbers in the interval `[-5,5]`.

```
r = -5 + (5+5)*rand(10,1)
```

```
r =

    3.1472
    4.0579
   -3.7301
    4.1338
    1.3236
   -4.0246
   -2.2150
    0.4688
    4.5751
    4.6489
```

In general, you can generate `N` random numbers in the interval `[a,b]` with the formula `r = a + (b-a).*rand(N,1)`.

Random Integers

Use the `randi` function (instead of `rand`) to generate 5 random integers from the uniform distribution between 10 and 50.

```
r = randi([10 50],1,5)
```

```
r =

    43    47    15    47    35
```

Random Complex Numbers

Generate a single random complex number with real and imaginary parts in the interval `[0,1]`.

```
a = rand + 1i*rand
```

```
a =

   0.8147 + 0.9058i
```

Reset Random Number Generator

Save the current state of the random number generator and create a 1-by-5 vector of random numbers.

```
s = rng;
r = rand(1,5)
```

```
r =

    0.0975    0.2785    0.5469    0.9575    0.9649
```

Restore the state of the random number generator to `s`, and then create a new 1-by-5 vector of random numbers. The values are the same as before.

```
rng(s);
r1 = rand(1,5)
```

```
r1 =
```

```
    0.0975    0.2785    0.5469    0.9575    0.9649
```

Always use the `rng` function (rather than the `rand` or `randn` functions) to specify the settings of the random number generator. For more information, see " Replace Discouraged Syntaxes of rand and randn".

3-D Array of Random Numbers

Create a 3-by-2-by-3 array of random numbers.

```
X = rand([3,2,3])
```

```
X(:,:,1) =

    0.8909    0.1978
    0.3342    0.0305
    0.6987    0.7441


X(:,:,2) =

    0.5000    0.6099
    0.4799    0.6177
    0.9047    0.8594


X(:,:,3) =

    0.8055    0.2399
    0.5767    0.8865
    0.1829    0.0287
```

Specify Data Type of Random Numbers

Create a 1-by-4 vector of random numbers whose elements are single precision.

```
r = rand(1,4,'single')
```

```
r =

    0.1270    0.9134    0.6324    0.0975
```

```
class(r)
```

```
ans =
```

```
single
```

Clone Size from Existing Array

Create a matrix of random numbers with the same size as an existing array.

```
A = [3 2; -2 1];
sz = size(A);
X = rand(sz)
```

```
X =

    0.4899    0.9787
    0.1679    0.7127
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = rand(size(A));
```

Clone Size and Data Type from Existing Array

Create a 2-by-2 matrix of single precision random numbers.

```
p = single([3 2; -2 1]);
```

Create an array of random numbers that is the same size and data type as `p`.

```
X = rand(size(p),'like',p)
```

```
X =

    0.5005    0.0596
    0.4711    0.6820
```

```
class(X)
```

```
ans =

single
```

Clone Distributed Array

If you have Parallel Computing Toolbox, create a 1000-by-1000 distributed array of random numbers with underlying data type `single`. For the `distributed` data type, the `'like'` syntax clones the underlying data type in addition to the primary data type.

```
p = rand(1000,'single','distributed');
```

Create an array of random numbers that is the same size, primary data type, and underlying data type as `p`.

```
X = rand(size(p),'like',p);

class(X)

ans =

distributed

classUnderlying(X)

ans =

single
```

- "Create Arrays of Random Numbers"
- "Random Numbers Within a Specific Range"
- "Random Numbers Within a Sphere"

Input Arguments

`n` — Size of square matrix

integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension (as separate arguments)

integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

- Beyond the second dimension, `rand` ignores trailing dimensions with a size of 1. For example, `rand(3,1,1,1)` produces a 3-by-1 vector of random numbers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)

integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `rand` ignores trailing dimensions with a size of 1. For example, `rand([3,1,1,1])` produces a 3-by-1 vector of random numbers.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`typename` — Data type (class) to create

`'double'` (default) | `'single'`

Data type (class) to create, specified as the string `'double'`, `'single'`, or the name of another class that provides `rand` support.

Example: `rand(5,'single')`

`p` — Prototype of array to create

numeric array

Prototype of array to create, specified as a numeric array.

Example: `rand(5,'like',p)`

Data Types: `single` | `double`
Complex Number Support: Yes

More About

- "Creating and Controlling a Random Number Stream"

- “Class Support for Array-Creation Functions”
- “ Replace Discouraged Syntaxes of rand and randn”
- “Why Do Random Numbers Repeat After Startup?”

See Also

`randi` | `randn` | `randperm` | `RandStream` | `rng` | `sprand` | `sprandn`

Introduced before R2006a

rand (RandStream)

Uniformly distributed random numbers

Class

`RandStream`

Syntax

```
r = rand(s,n)
r = rand(s,m,n)
r = rand(s,[m,n])
r = rand(s,m,n,p,...)
r = rand(s,[m,n,p,...])
r = rand(s)
r = rand(s,size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

`r = rand(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). The values are drawn from the random stream `s`.

`r = rand(s,m,n)` or `r = rand(s,[m,n])` returns an `m`-by-`n` matrix.

`r = rand(s,m,n,p,...)` or `r = rand(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array.

`r = rand(s)` returns a scalar.

`r = rand(s,size(A))` returns an array the same size as `A`.

`r = rand(..., 'double')` or `r = rand(..., 'single')` returns an array of uniform values of the specified class.

Note: The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `rand` is determined by the internal state of the random number stream `s`. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

`rand` | `randi (RandStream)` | `randn (RandStream)` | `@RandStream` | `randperm (RandStream)`

randi

Uniformly distributed pseudorandom integers

Syntax

```
X = randi(imax)
X = randi(imax,n)
X = randi(imax,sz1,...,szN)
X = randi(imax,sz)

X = randi(imax,classname)
X = randi(imax,n,classname)
X = randi(imax,sz1,...,szN,classname)
X = randi(imax,sz,classname)

X = randi(imax,'like',p)
X = randi(imax,n,'like',p)
X = randi(imax,sz1,...,szN,'like',p)
X = randi(imax,sz,'like',p)

X = randi([imin,imax], ___ )
```

Description

`X = randi(imax)` returns a pseudorandom scalar integer between `1` and `imax`.

`X = randi(imax,n)` returns an `n`-by-`n` matrix of pseudorandom integers drawn from the discrete uniform distribution on the interval [`1`,`imax`].

`X = randi(imax,sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array where `sz1,...,szN` indicates the size of each dimension. For example, `randi(10,3,4)` returns a 3-by-4 array of pseudorandom integers between 1 and 10.

`X = randi(imax,sz)` returns an array where size vector `sz` defines `size(X)`. For example, `randi(10,[3,4])` returns a 3-by-4 array of pseudorandom integers between 1 and 10.

`X = randi(imax,classname)` returns a pseudorandom integer where `classname` specifies the data type. `classname` can be `'single'`, `'double'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, or `'uint32'`.

`X = randi(imax,n,classname)` returns an `n`-by-`n` array of data type `classname`.

`X = randi(imax,sz1,...,szN,classname)` returns an `sz1`-by-...-by-`szN` array of data type `classname`..

`X = randi(imax,sz,classname)` returns an array where size vector `sz` defines `size(X)` and `classname` defines `class(X)`.

`X = randi(imax,'like',p)` returns a pseudorandom integer like `p`; that is, with the same data type (class).

`X = randi(imax,n,'like',p)` returns an `n`-by-`n` array like `p`.

`X = randi(imax,sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array like `p`.

`X = randi(imax,sz,'like',p)` returns an array like `p` where size vector `sz` defines `size(X)`.

`X = randi([imin,imax], ___ )` returns an array containing integers drawn from the discrete uniform distribution on the interval [`imin`,`imax`], using any of the above syntaxes.

The sequence of numbers produced by `randi` is determined by the settings of the uniform random number generator that underlies `rand`, `randn`, and `randi`. The `randi` function uses one uniform random value to create each integer random value. You can control that shared random number generator using `rng`.

Examples

Square Matrix of Random Integers

Generate a 5-by-5 matrix of random integers between 1 and 10. The first input to `randi` indicates the largest integer in the sampling interval (the smallest integer in the interval is 1).

```
r = randi(10,5)
```

```
r =

     4     3     2    10     5
     6     5     4    10     6
     5     1     2     1    10
     7    10     5     8     5
     7     2     4     3    10
```

Random Integers Within Specified Interval

Generate a 10-by-1 column vector of uniformly distributed random integers from the sample interval `[-5,5]`.

```
r = randi([-5,5],10,1)
```

```
r =

     3
     4
    -4
     5
     1
    -4
    -2
     1
     5
     5
```

Control Random Number Generation

Save the current state of the random number generator and create a 1-by-5 vector of random integers.

```
s = rng;
r = randi(10,1,5)
```

```
r =

     1     6     9     7     2
```

Restore the state of the random number generator to `s`, and then create a new 1-by-5 vector of random integers. The values are the same as before.

```
rng(s);
r1 = randi(10,1,5)
```

```
r1 =

     1     6     9     7     2
```

Always use the `rng` function (rather than the `rand` or `randn` functions) to specify the settings of the random number generator. For more information, see " Replace Discouraged Syntaxes of rand and randn".

3-D Array of Random Integers

Create a 3-by-2-by-3 array of uniformly distributed random integers between 1 and 500.

```
X = randi(500,[3,2,3])
```

```
X(:,:,1) =

   185    79
   231   428
   491   323


X(:,:,2) =

   189   242
    96    61
   215   295


X(:,:,3) =

   114   126
   193   146
   292   309
```

Random Integers of Other Data Types

Create a 1-by-4 vector of random numbers whose elements are of type `int16`.

```
r = randi(100,1,4,'int16')
```

```
r =

     28     55     96     97
```

```
class(r)
```

```
ans =

int16
```

Size Defined by Existing Array

Create a matrix of uniformly distributed random integers between 1 and 10 with the same size as an existing array.

```
A = [3 2; -2 1];
sz = size(A);
X = randi(10,sz)
```

```
X =

     3    10
     9     8
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = randi(10,size(A));
```

Size and Numeric Data Type Defined by Existing Array

Create a 2-by-2 matrix of 8-bit signed integers.

```
p = int8([3 2; -2 1]);
```

Create an array of random integers that is the same size and data type as `p`.

```
X = randi(10,size(p),'like',p)
```

```
X =

    4    2
    6   10
```

```
class(X)
```

```
ans =

int8
```

- “Random Integers”
- “Create Arrays of Random Numbers”

- “Generate Random Numbers That Are Repeatable”
- “Generate Random Numbers That Are Different”

Input Arguments

`imax` — Largest integer in sample interval

positive integer

Largest integer in sample interval, specified as a positive integer. `randi` draws values from the uniform distribution in the sample interval `[1,imax]`.

Example: `randi(10,5)`

`imin` — Smallest integer in sample interval

1 (default) | scalar integer

Smallest integer in sample interval, specified as a scalar integer.

Both `imin` and `imax` must be integers that satisfy `imin` ≤ `imax`.

For example, `randi([50,100],5)` returns a 5-by-5 matrix of random integers between (and including) 50 and 100.

`n` — Size of square matrix

integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension (as separate arguments)

two or more integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.

- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randi` ignores trailing dimensions with a size of 1. For example, `randi([5,10],3,1,1,1)` produces a 3-by-1 vector of random integers between 5 and 10.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)

integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randi` ignores trailing dimensions with a size of 1. For example, `randi([5,10],[3,1,1,1])` produces a 3-by-1 vector of random integers between 5 and 10.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`classname` — Data type (class) to create

`'double'` (default) | `'single'` | `'int8'` | `'uint8'` | ...

Output class, specified as the string `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, or the name of another class that provides `randi` support.

Example: `randi(5,5,'int8')`

Data Types: `char`

`p` — Prototype of array to create

numeric array

Prototype of array to create, specified as a numeric array.

Example: `randi(5,5,'like',p)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`
Complex Number Support: Yes

More About

Tips

- The arrays returned by `randi` might contain repeated integer values. This behavior is sometimes referred to as sampling with replacement. Use `randperm` if you require all unique values.

- "Creating and Controlling a Random Number Stream"
- "Class Support for Array-Creation Functions"
- "Why Do Random Numbers Repeat After Startup?"

See Also

`rand` | `randn` | `randperm` | `RandStream` | `rng`

randi (RandStream)

Uniformly distributed pseudorandom integers

Class

`RandStream`

Syntax

```
r = randi(s,imax,n)
r = randi(s,imax,m,n)
r = randi(s,imax,[m,n])
r = randi(s,imax,m,n,p,...)
r = randi(s,imax,[m,n,p,...])
r = randi(s,imax)
r = randi(s,imax,size(A))
r = randi(s,[imin,imax],...)
r = randi(...,classname)
```

Description

`r = randi(s,imax,n)` returns an `n`-by-`n` matrix containing pseudorandom integer values drawn from the discrete uniform distribution on `1:imax`. `randi` draws those values from the random stream `s`.

`r = randi(s,imax,m,n)` or `r = randi(s,imax,[m,n])` returns an `m`-by-`n` matrix.

`r = randi(s,imax,m,n,p,...)` or `r = randi(s,imax,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array.

`r = randi(s,imax)` returns a scalar.

`r = randi(s,imax,size(A))` returns an array the same size as `A`.

`r = randi(s,[imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on `imin:imax`.

`r = randi(...,classname)` returns an array of integer values of class `classname`. `classname` does not support 64-bit integers.

Note: The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The arrays returned by `randi` might contain repeated integer values. This is sometimes referred to as sampling with replacement. To get unique integer values, sometimes referred to as sampling without replacement, use `randperm (RandStream)`.

The sequence of numbers produced by `randi` is determined by the internal state of the random stream `s`. `randi` uses one uniform value from `s` to generate each integer value. Resetting `s` to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

`randi` | `RandStream` | `rand (RandStream)` | `randperm (RandStream)` | `randn (RandStream)`

randn

Normally distributed random numbers

Syntax

```
X = randn
X = randn(n)
X = randn(sz1,...,szN)
X = randn(sz)

X = randn(___,typename)
X = randn(___,'like',p)
```

Description

`X = randn` returns a random scalar drawn from the standard normal distribution.

`X = randn(n)` returns an `n`-by-`n` matrix of normally distributed random numbers.

`X = randn(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of random numbers where `sz1,...,szN` indicate the size of each dimension. For example, `randn(3,4)` returns a 3-by-4 matrix.

`X = randn(sz)` returns an array of random numbers where size vector `sz` defines `size(X)`. For example, `randn([3 4])` returns a 3-by-4 matrix.

`X = randn(___,typename)` returns an array of random numbers of data type `typename`. The `typename` input can be either `'single'` or `'double'`. You can use any of the input arguments in the previous syntaxes.

`X = randn(___,'like',p)` returns an array of random numbers like `p`; that is, of the same object type as `p`. You can specify either `typename` or `'like'`, but not both.

The sequence of numbers produced by `randn` is determined by the internal settings of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. You can control that shared random number generator using `rng`.

Note: Use the `rng` function instead of `rand` or `randn` with the `'seed'`, `'state'`, or `'twister'` inputs. For more information, see " Replace Discouraged Syntaxes of rand and randn"

Examples

Matrix of Random Numbers

Generate a 5-by-5 matrix of normally distributed random numbers.

```
r = randn(5)
```

```
r =

    1.0347    0.8884    1.4384   -0.1022   -0.0301
    0.7269   -1.1471    0.3252   -0.2414   -0.1649
   -0.3034   -1.0689   -0.7549    0.3192    0.6277
    0.2939   -0.8095    1.3703    0.3129    1.0933
   -0.7873   -2.9443   -1.7115   -0.8649    1.1093
```

Bivariate Normal Random Numbers

Generate values from a bivariate normal distribution with specified mean vector and covariance matrix.

```
mu = [1 2];
sigma = [1 0.5; 0.5 2];
R = chol(sigma);
z = repmat(mu,10,1) + randn(10,2)*R
```

```
z =

    1.3271    3.3688
    2.0826    3.3279
    2.0061    2.9663
    0.3491    3.3285
    1.2571    3.3585
    0.0556    1.8450
   -0.3218    0.4258
    1.9248    1.6005
    1.0000    3.5770
```

```
    0.9451   -0.1597
```

Random Complex Numbers

Generate a single random complex number with normally distributed real and imaginary parts.

```
a = randn + 1i*randn
```

```
a =

   0.5377 + 1.8339i
```

Reset Random Number Generator

Save the current state of the random number generator and create a 1-by-5 vector of random numbers.

```
s = rng;
r = randn(1,5)
```

```
r =

   -0.0245   -1.9488    1.0205    0.8617    0.0012
```

Restore the state of the random number generator to `s`, and then create a new 1-by-5 vector of random numbers. The values are the same as before.

```
rng(s);
r1 = randn(1,5)
```

```
r1 =

   -0.0245   -1.9488    1.0205    0.8617    0.0012
```

Always use the `rng` function (rather than the `rand` or `randn` functions) to specify the settings of the random number generator. For more information, see " Replace Discouraged Syntaxes of rand and randn".

3-D Array of Random Numbers

Create a 3-by-2-by-3 array of random numbers.

```
X = randn([3,2,3])
```

```
X(:,:,1) =
```

```
   -0.0708   -2.1924
   -2.4863   -2.3193
    0.5812    0.0799


X(:,:,2) =

   -0.9485    0.8577
    0.4115   -0.6912
    0.6770    0.4494


X(:,:,3) =

    0.1006    0.8979
    0.8261   -0.1319
    0.5362   -0.1472
```

Specify Data Type of Random Numbers

Create a 1-by-4 vector of random numbers whose elements are single precision.

```
r = randn(1,4,'single')
```

```
r =

    0.5377    1.8339   -2.2588    0.8622
```

```
class(r)
```

```
ans =

single
```

Clone Size from Existing Array

Create a matrix of normally distributed random numbers with the same size as an existing array.

```
A = [3 2; -2 1];
sz = size(A);
X = randn(sz)
```

```
X =
```

```
    1.0078   -0.5046
   -2.1237   -1.2706
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = randn(size(A));
```

Clone Size and Data Type from Existing Array

Create a 2-by-2 matrix of single precision random numbers.

```
p = single([3 2; -2 1]);
```

Create an array of random numbers that is the same size and data type as `p`.

```
X = randn(size(p),'like',p)
```

```
X =

   -0.3826    0.8257
    0.6487   -1.0149
```

```
class(X)
```

```
ans =

single
```

Clone Distributed Array

If you have Parallel Computing Toolbox, create a 1000-by-1000 distributed array of random numbers with underlying data type `single`. For the `distributed` data type, the `'like'` syntax clones the underlying data type in addition to the primary data type.

```
p = randn(1000,'single','distributed');
```

Create an array of random numbers that is the same size, primary data type, and underlying data type as `p`.

```
X = randn(size(p),'like',p);
```

```
class(X)
```

```
ans =

distributed
```

```
classUnderlying(X)

ans =

single
```

- "Create Arrays of Random Numbers"
- "Random Numbers Within a Specific Range"
- "Random Numbers Within a Sphere"
- "Random Numbers from Normal Distribution with Specific Mean and Variance"

Input Arguments

n — Size of square matrix

integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN — Size of each dimension (as separate arguments)

integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randn` ignores trailing dimensions with a size of 1. For example, `randn(3,1,1,1)` produces a 3-by-1 vector of random numbers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Size of each dimension (as a row vector)

integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randn` ignores trailing dimensions with a size of 1. For example, `randn([3,1,1,1])` produces a 3-by-1 vector of random numbers.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`typename` — Data type (class) to create

`'double'` (default) | `'single'`

Data type (class) to create, specified as the string `'double'`, `'single'`, or the name of another class that provides `randn` support.

Example: `randn(5,'single')`

`p` — Prototype of array to create

numeric array

Prototype of array to create, specified as a numeric array.

Example: `randn(5,'like',p)`

Data Types: `single` | `double`
Complex Number Support: Yes

More About

- "Creating and Controlling a Random Number Stream"
- "Class Support for Array-Creation Functions"
- " Replace Discouraged Syntaxes of rand and randn"
- "Why Do Random Numbers Repeat After Startup?"

See Also

`rand` | `randi` | `randperm` | `RandStream` | `rng` | `sprand` | `sprandn`

Introduced before R2006a

randn (RandStream)

Normally distributed pseudorandom numbers

Class

`RandStream`

Syntax

```
r = randn(s,m,n)
r = randn(s,[m,n])
r = randn(s,m,n,p,...)
r = randn(s,[m,n,p,...])
r = randn(s)
r = randn(s,size(A))
r = randn(...,'double')
r = randn(...,'single')
```

Description

`r = randn(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard normal distribution. `randn` draws those values from the random stream `s`.

`r = randn(s,m,n)` or `r = randn(s,[m,n])` returns an `m`-by-`n` matrix.

`r = randn(s,m,n,p,...)` or `r = randn(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array.

`r = randn(s)` returns a scalar.

`r = randn(s,size(A))` returns an array the same size as `A`.

`r = randn(...,'double')` or `r = randn(...,'single')` returns an array of uniform values of the specified class.

Note: The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randn` is determined by the internal state of the random stream `s`. `randn` uses one or more uniform values from `s` to generate each normal value. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

`randn` | `RandStream` | `rand (RandStream)` | `randi (RandStream)`

randperm

Random permutation

Syntax

```
p = randperm(n)
p = randperm(n,k)
```

Description

`p = randperm(n)` returns a row vector containing a random permutation of the integers from 1 to `n` inclusive.

`p = randperm(n,k)` returns a row vector containing `k` unique integers selected randomly from 1 to `n` inclusive.

Examples

`randperm(6)`
might be the vector

```
[3  2  6  4  1  5]
```

or it might be some other permutation of the integers from 1 to 6, depending on the state of the random number generator. Two successive calls to `randperm` would in most cases return two different vectors:

```
randperm(6)
ans =
     5     2     6     4     1     3

randperm(6)
ans =
     4     1     6     2     3     5
```

`randperm(6,3)`
might be the vector

```
[4 2 5]
```
or it might be some other permutation of any three integers from 1 to 6 inclusive, depending on the state of the random number generator.

More About

Tips

For `p = randperm(n,k)`, `p` contains `k` *unique* values. `randperm` performs k-permutations (sampling without replacement). To allow repeated values in the output (sampling with replacement), use `randi(n,1,k)`.

`randperm` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

See Also

`permute` | `randi` | `nchoosek` | `randperm(RandStream)` | `perms` | `rng`

Introduced before R2006a

randperm (RandStream)

Random permutation

Class

`RandStream`

Syntax

```
p = randperm(s,n)
p = randperm(s,n,k)
```

Description

`p = randperm(s,n)` returns a row vector containing a random permutation of integers from 1 to `n` inclusive. `randperm(s,n)` uses random values drawn from the random stream `s`.

`p = randperm(s,n,k)` returns a row vector containing `k` unique integers selected randomly from 1 to `n` inclusive.

Examples

Create a random stream `s` and generate a random permutation of the integers from 1 to 6 based on `s`:

```
s = RandStream('mt19937ar','Seed',0);
randperm(s,6)
```

MATLAB returns the vector

```
[6     3     5     1     2     4]
```

Use the random stream `s` to generate three integers between 1 and 10:

```
randperm(s,10,3)
```

```
ans =
     1     8     9
```

More About

Tips

For `p = randperm(s,n,k)`, `p` contains `k` *unique* values. `randperm` performs k-permutations (sampling without replacement). To allow repeated values in the output (sampling with replacement), use `randi(s,n,1,k)`.

See Also

`permute` | `randi (RandStream)` | `randperm` | `nchoosek` | `perms` | `rand`

RandStream

Random number stream

Constructor

`RandStream`

Description

Pseudorandom numbers in MATLAB come from one or more random number streams. The simplest way to generate arrays of random numbers is to use `rand`, `randn`, or `randi`. These functions all rely on the same stream of uniform random numbers, known as the *global stream*. You can create other streams that act separately from the global stream, and you can use their `rand`, `randi`, or `randn` methods to generate arrays of random numbers. You can also create a random number stream and make it the global stream.

To create a single random number stream, use the `RandStream` constructor. To create multiple independent random number streams, use `RandStream.create`. The `rng` function provides a simple interface to create a new global stream.

`stream = RandStream.getGlobalStream` returns the global random number stream, that is, the one currently used by the `rand`, `randi`, and `randn` functions.

`prevstream = RandStream.setGlobalStream(stream)` designates the random number stream `stream` as the new global stream to be used by the `rand`, `randi`, and `randn` functions, and returns the previous global stream.

A random number stream `s` has properties that control its behavior. Access or assign to a property using `p = s.Property` or `s.Property = p`. The following table lists defined properties:

Properties

Property	Description
`Type`	(Read-only) Generator algorithm used by the stream. The list of possible generators is given by `RandStream.list`.
`Seed`	(Read-only) Seed value used to create the stream.
`NumStreams`	(Read-only) Number of streams in the group in which the current stream was created.
`StreamIndex`	(Read-only) Index of the current stream from among the group of streams with which it was created.
`State`	Internal state of the generator. You should not depend on the format of this property. The value you assign to `S.State` must be a value read from `S.State` previously. Use `reset` to return a stream to a predictable state without having previously read from the `State` property. The sequence of random numbers produced by a random number stream `s` is determined by the internal state of its random number generator. Saving and restoring the generator's internal state with the `State` property allows you to reproduce a sequence of random numbers.
`Substream`	Index of the substream to which the stream is currently set. The default is 1. Multiple substreams are not supported by all generator types; the multiplicative lagged Fibonacci generator (`mlfg6331_64`) and combined multiple recursive generator (`mrg32k3a`) support substreams.
`NormalTransform`	Transformation algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Property	Description
`Antithetic`	Logical value indicating whether `S` generates antithetic pseudorandom values, that is, the usual values subtracted from 1. The default is false.
`FullPrecision`	Logical value indicating whether `S` generates values using its full precision. Some generators can create pseudorandom values faster, but with fewer random bits, if `FullPrecision` is false. The default is true.

Methods

Method	Description
`RandStream`	Create a random number stream.
`RandStream.create`	Create multiple independent random number streams.
`get`	Get the properties of a random stream object.
`list`	List available random number generator algorithms.
`set`	Set random stream property.
`RandStream.getGlobalStream`	Get the global random number stream.
`RandStream.setGlobalStream`	Set global random number stream.
`reset`	Reset a stream to its initial internal state
`rand`	Pseudorandom numbers from a uniform distribution
`randn`	Pseudorandom numbers from a standard normal distribution
`randi`	Pseudorandom integers from a uniform discrete distribution
`randperm`	Random permutation of a set of values

Examples

Example 1

Create a single stream and designate it as the current global stream:

```
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);
```

Example 2

Create three independent streams:

```
[s1,s2,s3] = RandStream.create('mrg32k3a','NumStreams',3);
r1 = rand(s1,100000,1);
r2 = rand(s2,100000,1);
r3 = rand(s3,100000,1);
corrcoef([r1,r2,r3])
```

Example 3

Create only one stream from a set of three independent streams, and designate it as the current global stream:

```
s2 = RandStream.create('mrg32k3a','NumStreams',3,...
     'StreamIndices',2);
RandStream.setGlobalStream(s2);
```

Example 4

Reset the global random number stream that underlies `rand`, `randi`, and `randn` back to its beginning, to reproduce previous results:

```
stream = RandStream.getGlobalStream;
reset(stream);
```

Example 5

Save and restore the current global stream's state to reproduce the output of `rand`:

```
stream = RandStream.getGlobalStream;
savedState = stream.State;
u1 = rand(1,5)
u1 =
    0.8147    0.9058    0.1270    0.9134    0.6324

stream.State = savedState;
u2 = rand(1,5)
u2 =
    0.8147    0.9058    0.1270    0.9134    0.6324
```

`u2` contains exactly the same values as `u1`.

Example 6

Reset the global random number stream to its initial settings. This causes `rand`, `randi`, and `randn` to start over, as if in a new MATLAB session:

```
s = RandStream('mt19937ar','Seed',0);
RandStream.setGlobalStream(s);
```

Example 7

Reinitialize the global random number stream using a seed based on the current time. This causes `rand`, `randi`, and `randn` to return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session as it may affect the statistical properties of the random numbers MATLAB produces:

```
s = RandStream('mt19937ar','Seed','shuffle');
RandStream.setGlobalStream(s);
```

Example 8

Change the transformation algorithm that `randn` uses to create normal pseudorandom values from uniform values. This does not replace or reset the global stream.

```
stream = RandStream.getGlobalStream;
stream.NormalTransform = 'inversion'
```

See Also

`rand` | `rng` | `randn` | `randi`

RandStream constructor

Random number stream

Class

`RandStream`

Syntax

```
s = RandStream('gentype')
s = RandStream('gentype',Name,Value)
```

Description

`s = RandStream('gentype')` creates a random number stream that uses the uniform pseudorandom number generator algorithm specified by `gentype`. `RandStream.list` returns all possible values for `gentype`, or see " Choosing a Random Number Generator" for details on generator algorithms.

`s = RandStream('gentype',Name,Value)` allows you to specify one or more optional `Name,Value` pairs to control creation of the stream.

Once you have created a random, you can use `RandStream.setGlobalStream` to make it the global stream, so that the functions `rand`, `randi`, and `randn` draw values from it.

Parameters for `RandStream` are:

Parameter	Description
`Seed`	Nonnegative scalar integer with which to initialize all streams. Seeds must be an integer between 0 and $2^{32} - 1$ or `'shuffle'` to create a seed based on the current time. Default is 0.

Parameter	Description
`NormalTransform`	Transformation algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Examples

Example 1

Create a random number stream, make it the global stream, and save and restore its state to reproduce the output of `randn`:

```
s = RandStream('mrg32k3a');
RandStream.setGlobalStream(s);
savedState = s.State;
z1 = randn(1,5)
z1 =
   -0.1894   -1.4426   -0.3592    0.8883   -0.4337
s.State = savedState;
z2 = randn(1,5)
z2 =
   -0.1894   -1.4426   -0.3592    0.8883   -0.4337
```

`z2` contains exactly the same values as `z1`.

Example 2

Return `rand`, `randi`, and `randn` to their default startup settings:

```
s = RandStream('mt19937ar','Seed',0)
RandStream.setGlobalStream(s);
```

Example 3

Replace the current global random number stream with a stream whose seed is based on the current time, so `rand`, `randi`, and `randn` will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB

session as it may affect the statistical properties of the random numbers MATLAB produces:

```
s = RandStream('mt19937ar','Seed','shuffle');
RandStream.setGlobalStream(s);
```

More About

Tips

- Streams created using `RandStream` might not be independent from each other. Use `RandStream.create` to create multiple streams that are independent.

See Also

`RandStream` | `RandStream.rand` | `RandStream.randn` | `RandStream.randi` | `RandStream.getGlobalStream` | `RandStream.setGlobalStream` | `RandStream.list` | `rng` | `RandStream.create`

RandStream.getGlobalStream

Current global random number stream

Class

`@RandStream`

Syntax

```
stream = RandStream.getGlobalStream
```

Description

`stream = RandStream.getGlobalStream` returns the current global random number stream.

`rand`, `randi`, and `randn` all rely on the same stream of uniform pseudorandom numbers, known as the *global stream*. `rand` draws one value from that stream to generate each uniform value it returns. `randi` draws one uniform value from that stream to generate each integer value it returns. And `randn` draws one or more uniform values to generate each normal value it returns. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

Note: The `rng` function is a shorter alternative for many common uses of `RandStream.getGlobalStream`.

More About

- "Creating and Controlling a Random Number Stream"
- "Managing the Global Stream"

See Also

`rand` | `randi` | `randn` | `RandStream` | `RandStream.setGlobalStream` | `rng`

RandStream.setGlobalStream

Set global random number stream

Syntax

```
prevstream = RandStream.setGlobalStream(stream)
```

Description

`prevstream = RandStream.setGlobalStream(stream)` designates the random number stream, specified as `stream`, to be the global stream that the `rand`, `randi`, and `randn` functions draw values from. It returns the previous global random number stream as `prevstream`.

`rand`, `randi`, and `randn` all rely on the same stream of uniform pseudorandom numbers, known as the *global stream*. `rand` draws one value from that stream to generate each uniform value it returns. `randi` draws one uniform value from that stream to generate each integer value it returns. And `randn` draws one or more uniform values to generate each normal value it returns. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

Note: The `rng` function is a shorter alternative for many common uses of `RandStream.setGlobalStream`.

More About

- "Creating and Controlling a Random Number Stream"
- "Managing the Global Stream"

See Also

`rand` | `randi` | `randn` | `RandStream` | `RandStream.getGlobalStream` | `rng`

rank

Rank of matrix

Syntax

```
k = rank(A)
k = rank(A,tol)
```

Description

The `rank` function provides an estimate of the number of linearly independent rows or columns of a full matrix.

`k = rank(A)` returns the number of singular values of `A` that are larger than the default tolerance, `max(size(A))*eps(norm(A))`.

`k = rank(A,tol)` returns the number of singular values of `A` that are larger than `tol`.

More About

Tips

Use `sprank` to determine the structural rank of a sparse matrix.

Algorithms

There are a number of ways to compute the rank of a matrix. MATLAB software uses the method based on the singular value decomposition, or SVD. The SVD algorithm is the most time consuming, but also the most reliable.

The `rank` algorithm is

```
s = svd(A);
tol = max(size(A))*eps(max(s));
r = sum(s > tol);
```

See Also

sprank

Introduced before R2006a

rat

Rational fraction approximation

Syntax

```
R = rat(X)
R = rat(X,tol)

[N,D] = rat( ___ )
```

Description

`R = rat(X)` returns the rational fraction approximation of `X` to within the default tolerance, `1e-6*norm(X(:),1)`. The approximation is a string containing the truncated continued fractional expansion.

`R = rat(X,tol)` approximates `X` to within the tolerance, `tol`.

`[N,D] = rat( ___ )` returns two arrays, `N` and `D`, such that `N./D` approximates `X`, using any of the above syntaxes.

Examples

Approximate Value of π

Approximate the value of π using a rational representation of the quantity `pi`.

The mathematical quantity π is not a rational number, but the quantity `pi` that approximates it *is* a rational number since all floating-point numbers are rational.

Find the rational representation of `pi`.

```
format rat
pi
```

```
ans =
```

```
355/113
```

The resulting expression is a string. You also can use `rats(pi)` to get the same answer.

Use `rat` to see the continued fractional expansion of `pi`.

```
R = rat(pi)
```

```
R =

3 + 1/(7 + 1/(16))
```

The resulting string is an approximation by continued fractional expansion. If you consider the first two terms of the expansion, you get the approximation $3+\frac{1}{7}=\frac{22}{7}$, which only agrees with `pi` to 2 decimals.

However, if you consider all three terms printed by `rat`, you can recover the value `355/113`, which agrees with `pi` to 6 decimals.

$$3+\frac{1}{7+\frac{1}{16}}=\frac{355}{113}.$$

Specify a tolerance for additional accuracy in the approximation.

```
R = rat(pi,1e-7)
```

```
R =

3 + 1/(7 + 1/(16 + 1/(-294)))
```

The resulting approximation, `104348/33215`, agrees with `pi` to 9 decimals.

Express Array Elements as Ratios

Create a 4-by-4 matrix.

```
format short;
X = hilb(4)
```

```
X =

    1.0000    0.5000    0.3333    0.2500
```

```
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429
```

Express the elements of `X` as ratios of small integers using `rat`.

```
[N,D] = rat(X)
```

```
N =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


D =

     1     2     3     4
     2     3     4     5
     3     4     5     6
     4     5     6     7
```

The two matrices, `N` and `D`, approximate `X` with `N./D`.

View the elements of `X` as ratios using `format rat`.

```
format rat
X
```

```
X =

       1            1/2          1/3          1/4
       1/2          1/3          1/4          1/5
       1/3          1/4          1/5          1/6
       1/4          1/5          1/6          1/7
```

In this form, it is clear that `N` contains the numerators of each fraction and `D` contains the denominators.

Input Arguments

`X` — Input array

numeric array

Input array, specified as a numeric array of class `single` or `double`.

Data Types: `single | double`
Complex Number Support: Yes

`tol` — Tolerance
scalar

Tolerance, specified as a scalar. `N` and `D` approximate `X`, such that `N./D - X < tol`. The default tolerance is `1e-6*norm(X(:),1)`.

Output Arguments

`R` — Continued fraction
string

Continued fraction, returned as a string. The accuracy of the rational approximation via continued fractions increases with the number of terms.

`N` — Numerator
numeric array

Numerator, returned as a numeric array. `N./D` approximates `X`.

`D` — Denominator
numeric array

Denominator, returned as a numeric array. `N./D` approximates `X`.

More About

Algorithms

Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. Rational approximations are generated by truncating continued fraction expansions.

The `rat` function approximates each element of `X` by a continued fraction of the form

$$\frac{N}{D} = D_1 + \frac{1}{D_2 + \frac{1}{\left(D_3 + \ldots + \frac{1}{D_k} \right)}}.$$

The *D*s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when `X = sqrt(2)`. For `X = sqrt(2)` , the error with `k` terms is about `2.68*(.173)^k`, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

See Also

`format | rats`

Introduced before R2006a

rats

Rational output

Syntax

```
S = rats(X)
S = rats(X,strlen)
```

Description

`S = rats(X)` returns a string containing the rational approximations to the elements of `X` using the default string length of `13`.

`rats` returns asterisks for elements that cannot be printed in the allotted space, but which are not negligible compared to the other elements in `X`.

`S = rats(X,strlen)` returns a string of length `strlen`. The rational approximation uses a tolerance that is inversely proportional to string length.

Examples

Rational Representation of Matrix

Create a 4-by-4 matrix.

```
format short
X = hilb(4)
```

```
X =

    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429
```

View the rational representation of the matrix using `rats`. The result is the same as using `format rat`.

```
R = rats(X)

R =

       1          1/2        1/3        1/4
      1/2         1/3        1/4        1/5
      1/3         1/4        1/5        1/6
      1/4         1/5        1/6        1/7
```

Adjust Output String Length

Find the rational representation of `pi` with the default string length and approximation tolerance. The result is the same as using `format rat`.

```
rats(pi)

ans =

    355/113
```

Adjust the string length of the output, which also adjusts the approximation tolerance.

```
rats(pi,20)

ans =

    104348/33215
```

The resulting rational approximation has greater accuracy. As the string length increases, the tolerance decreases.

Adjust the string length again to achieve greater accuracy.

```
rats(pi,25)

ans =

      1146408/364913
```

The resulting approximation agrees with `pi` to 10 decimal places.

Input Arguments

`X` — Input array

numeric array

Input array, specified as a numeric array of class `single` or `double`.

Data Types: `single | double`
Complex Number Support: Yes

`strlen` — String length
positive integer

String length, specified as a positive integer. The default string length is 13, which allows for 6 elements in 78 spaces.

Output Arguments

`S` — Rational output
string

Rational output, returned as a string.

More About

Algorithms

`rats` obtains rational approximations with `[N,D] = rat(X,tol)`, where `tol` is `min(10^(-(strlen-1)/2)*norm(X(isfinite(X)),1),.1)`. Thus, the tolerance is inversely proportional to the string length, `strlen`.

See Also
`format | rat`

Introduced before R2006a

rbbox

Create rubberband box for area selection

Syntax

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

Description

`rbbox` initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure's `CurrentPoint`, and begins tracking from this point.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` figure units, `rbbox` redraws the rubberband box. The default stepsize is `1`.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the *x* and *y* components of the lower left corner of the box, and `width` and `height` are the dimensions of the box.

Examples

Create an annotation rectangle by rubber banding the rectangle size in the figure.

```
set(gcf,'Units','normalized')
k = waitforbuttonpress;
rect_pos = rbbox;
annotation('rectangle',rect_pos,'Color','red')
```

Execute the code, click down and drag a rectangle within the figure. Releasing the mouse button draws a rectangle in the figure.

More About

Tips

`rbbox` is useful for defining and resizing a rectangular region:

- For box definition, `initialRect` is `[x y 0 0]`, where `(x,y)` is the figure's `CurrentPoint`.
- For box resizing, `initialRect` defines the rectangular region that you resize (e.g., a legend). `fixedPoint` is the corner diametrically opposite the tracking point.

`rbbox` returns immediately if a button is not currently pressed. Therefore, you use `rbbox` with `waitforbuttonpress` so that the mouse button is down when `rbbox` is called. `rbbox` returns when you release the mouse button.

See Also

`axis` | `dragrect` | `waitforbuttonpress`

Introduced before R2006a

rcond

Reciprocal condition number

Syntax

```
C = rcond(A)
```

Description

`C = rcond(A)` returns an estimate for the reciprocal condition of `A` in 1-norm. If `A` is well conditioned, `rcond(A)` is near 1.0. If `A` is badly conditioned, `rcond(A)` is near 0.

Examples

Sensitivity of Badly Conditioned Matrix

Examine the sensitivity of a badly conditioned matrix.

A notable matrix that is symmetric and positive definite, but badly conditioned, is the Hilbert matrix. The elements of the Hilbert matrix are $H(i,j) = 1/(i + j - 1)$.

Create a 10-by-10 Hilbert matrix.

```
A = hilb(10);
```

Find the reciprocal condition number of the matrix.

```
C = rcond(A)
```

```
C =

   2.8286e-14
```

The reciprocal condition number is small, so `A` is badly conditioned.

The condition of `A` has an effect on the solutions of similar linear systems of equations. To see this, compare the solution of $Ax = b$ to that of the perturbed system, $Ax = b + 0.01$.

Create a column vector of ones and solve $Ax = b$.

```
b = ones(10,1);
x = A\b;
```

Now change b by `0.01` and solve the perturbed system.

```
b1 = b + 0.01;
x1 = A\b1;
```

Compare the solutions, `x` and `x1`.

```
norm(x-x1)
```

```
ans =

   1.1250e+05
```

Since `A` is badly conditioned, a small change in `b` produces a very large change (on the order of 1e5) in the solution to $x = A\backslash b$. The system is sensitive to perturbations.

Find Condition of Identity Matrix

Examine why the reciprocal condition number is a more accurate measure of singularity than the determinant.

Create a 5-by-5 multiple of the identity matrix.

```
A = eye(5)*0.01;
```

This matrix is full rank and has five equal singular values, which you can confirm by calculating `svd(A)`.

Calculate the determinant of `A`.

```
det(A)
```

```
ans =

   1.0000e-10
```

Although the determinant of the matrix is close to zero, `A` is actually very well conditioned and *not* close to being singular.

Calculate the reciprocal condition number of `A`.

```
rcond(A)

ans =

     1
```

The matrix has a reciprocal condition number of `1` and is, therefore, very well conditioned. Use `rcond(A)` or `cond(A)` rather than `det(A)` to confirm singularity of a matrix.

Input Arguments

`A` — Input matrix

square numeric matrix

Input matrix, specified as a square numeric matrix.

Data Types: `single` | `double`

Output Arguments

`C` — Reciprocal condition number

scalar

Reciprocal condition number, returned as a scalar. The data type of `C` is the same as `A`.

The reciprocal condition number is a scale-invariant measure of how close a given matrix is to the set of singular matrices.

- If `C` is near 0, the matrix is nearly singular and badly conditioned.
- If `C` is near 1.0, the matrix is well conditioned.

More About

Tips

- `rcond` is a more efficient but less reliable method of estimating the condition of a matrix compared to the condition number, `cond`.

See Also

`cond` | `condest` | `norm` | `normest` | `rank` | `svd`

Introduced before R2006a

rdivide, ./

Right array division

Syntax

```
x = A./B
x = rdivide(A,B)
```

Description

`x = A./B` divides each element of A by the corresponding element of B. Inputs `A` and `B` must have the same size unless one is a scalar value. A scalar value is expanded into an array of the same size as the other input.

`x = rdivide(A,B)` is an alternative way to divide A by B, but is rarely used. It enables operator overloading for classes.

Examples

Divide Two Numeric Arrays

Create two numeric arrays, `A` and `B`, and divide the second array, `B`, into the first, `A`.

```
A = [2 4 6 8; 3 5 7 9];
B = 10*ones(2,4);
x = A./B
```

```
x =

    0.2000    0.4000    0.6000    0.8000
    0.3000    0.5000    0.7000    0.9000
```

Integer Division

Divide an `int16` scalar value by each element of an `int16` vector.

```
a = int16(10);
```

```
b = int16([3 4 6]);
x = a./b

x =

      3      3      2
```

MATLAB rounds the results when dividing integer data types.

Divide Scalar by Array

Create an array and divide it into a scalar.

```
C = 5;
D = magic(3);
x = C./D

x =

    0.6250    5.0000    0.8333
    1.6667    1.0000    0.7143
    1.2500    0.5556    2.5000
```

When you specify a scalar value to be divided by an array, the scalar value expands into an array of the same size, then element-by-element division is performed.

Input Arguments

A — Numerator

scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix, or multidimensional array. If `B` is an integer data type, `A` must be the same integer type or a scalar double.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`
Complex Number Support: Yes

B — Denominator

scalar | vector | matrix | multidimensional array

Denominator, specified as a scalar, vector, matrix, or multidimensional array. If `A` is an integer data type, `B` must be the same integer type or a scalar double.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`
Complex Number Support: Yes

Output Arguments

x — Solution

scalar | vector | matrix | multidimensional array

Solution, returned as a scalar, vector, matrix or multidimensional array. If either `A` or `B` is an integer data type, then `x` is that same integer data type.

More About

Tips

- When dividing integers, use `idivide` for more rounding options.
- MATLAB does not support complex integer division.

- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`idivide` | `ldivide` | `mldivide` | `mrdivide`

Introduced before R2006a

read

Class: VideoReader

Read video frame data from file

Note: `VideoReader.read` will be removed in a future release. Use `VideoReader.readFrame` instead.

Syntax

```
video = read(obj)
video = read(obj,index)
video = read( ___ ,'native')
```

Description

`video = read(obj)` reads in all video frames from the file associated with obj. The `read` method returns a `H`-by-`W`-by-`B`-by-`F` matrix, `video`, where `H` is the image frame height, `W` is the image frame width, `B` is the number of bands in the image (for example, 3 for RGB), and `F` is the number of frames read.

`video = read(obj,index)` reads only the specified frames. index can be a single number or a two-element array representing an index range of the video stream.

`video = read( ___ ,'native')` returns data in the format specified by the `VideoFormat` property, and can include any of the input arguments in the previous syntaxes.

Input Arguments

`obj`

Name of multimedia object created with `VideoReader`.

`index`

Frames to read, where the first frame number is 1. Use `Inf` to represent the last frame of the file.

For example:

```
video = read(obj, 1);         % first frame only
video = read(obj, [1 10]);    % first 10 frames
video = read(obj, Inf);       % last frame only
video = read(obj, [50 Inf]);  % frame 50 thru end
```

MATLAB cannot determine the number of frames in a variable frame rate file until you read the last frame. If the requested *`index`* extends beyond the end of the file, `read` returns either a warning or an error.

Default: `[1 Inf]`

Output Arguments

`video`

Array of data representing video frames, returned in a format dependent on the `VideoFormat` property of `obj`. For most files, the data type and dimensions of `video` are as follows. Note that when the `VideoFormat` property of `obj` is `'Indexed'`, the data type and dimensions of `video` depend on whether you call `read` with the `'native'` argument.

Value of `obj.VideoFormat`	Data Type of `video`	Dimensions of `video`	Description
`'RGB24'`	`uint8`	H-by-W-by-3-by-F	RGB24 image
`'Grayscale'`, without specifying `'native'`	`uint8`	H-by-W-by-1-by-F	Grayscale image
`'Indexed'`, without specifying `'native'`	`uint8`	H-by-W-by-3-by-F	RGB24 image
`'Grayscale'` or `'Indexed'`, specifying `'native'`	`struct`	1-by-F	MATLAB movie, which is an array of frame structure

Value of `obj.VideoFormat`	Data Type of `video`	Dimensions of `video`	Description
			arrays, each containing the fields `cdata` and `colormap`.

`H` is the image frame height, `W` is the image frame width, and `F` is the number of frames read.

For Motion JPEG 2000 files, the data type and dimensions of `video` are as follows.

Value of `obj.VideoFormat`	Data Type of `video`	Dimensions of `video`	Description
`'Mono8'`	`uint8`	H-by-W-by-1-by-F	Mono image
`'Mono8 Signed'`	`int8`	H-by-W-by-1-by-F	Mono signed image
`'Mono16'`	`uint16`	H-by-W-by-1-by-F	Mono image
`'Mono16 Signed'`	`int16`	H-by-W-by-1-by-F	Mono signed image
`'RGB24'`	`uint8`	H-by-W-by-3-by-F	RGB24 image
`'RGB24 Signed'`	`int8`	H-by-W-by-3-by-F	RGB24 signed image
`'RGB48'`	`uint16`	H-by-W-by-3-by-F	RGB48 image
`'RGB48 Signed'`	`int16`	H-by-W-by-3-by-F	RGB48 signed image

Examples

Read and Play Back Movie File

Read and play back the movie file `xylophone.mp4`.

```
xyloObj = VideoReader('xylophone.mp4');

nFrames = xyloObj.NumberOfFrames;
vidHeight = xyloObj.Height;
vidWidth = xyloObj.Width;
```

Preallocate the movie structure.

```
mov(1:nFrames) = ...
    struct('cdata',zeros(vidHeight,vidWidth, 3,'uint8'),...
           'colormap',[]);
```

Read one frame at a time.

```
for k = 1 : nFrames
    mov(k).cdata = read(xyloObj,k);
end
```

Size a figure based on the video's width and height.

```
hf = figure;
set(hf, 'position', [150 150 vidWidth vidHeight])
```

Play back the movie once at the video's frame rate.

```
movie(hf, mov, 1, xyloObj.FrameRate);
```

See Also

`movie` | `VideoReader`

How To

- “Read Video Files”

readFrame

Class: VideoReader

Read video frame from video file

Syntax

```
video = readFrame(obj)
video = readFrame(obj,'native')
```

Description

`video = readFrame(obj)` reads the next available video frame from the file associated with `obj`.

`video = readFrame(obj,'native')` returns data in the format specified by the `VideoFormat` property.

Input Arguments

`obj` — Object associated with video file to read

`VideoReader` object

Object associated with the video file to read, specified as a `VideoReader` object.

Output Arguments

`video` — Video frame data

array

Video frame data, returned as an array. The dimensions and data type of `video` depend on the `VideoFormat` property of `obj`.

For most files, the data type and dimensions of `video` are as follows. When the `VideoFormat` property of `obj` is `'Indexed'`, the data type and dimensions of `video` depend on whether you call `read` with the `'native'` argument.

Value of `obj.VideoFormat`	Data Type of `video`	Dimensions of `video`	Description
`'RGB24'`	`uint8`	H-by-W-by-3	RGB24 image
`'Grayscale'`, without specifying `'native'`	`uint8`	H-by-W-by-1	Grayscale image
`'Indexed'`, without specifying `'native'`	`uint8`	H-by-W-by-3	RGB24 image
`'Grayscale'` or `'Indexed'`, specifying `'native'`	`struct`	1-by-1	MATLAB movie, which is an array of frame structure arrays, each containing the fields `cdata` and `colormap`.

`H` is the image frame height and `W` is the image frame width.

For Motion JPEG 2000 files, the data type and dimensions of `video` are as follows.

Value of `obj.VideoFormat`	Data Type of `video`	Dimensions of `video`	Description
`'Mono8'`	`uint8`	H-by-W-by-1	Mono image
`'Mono8 Signed'`	`int8`	H-by-W-by-1	Mono signed image
`'Mono16'`	`uint16`	H-by-W-by-1	Mono image
`'Mono16 Signed'`	`int16`	H-by-W-by-1	Mono signed image
`'RGB24'`	`uint8`	H-by-W-by-3	RGB24 image
`'RGB24 Signed'`	`int8`	H-by-W-by-3	RGB24 signed image
`'RGB48'`	`uint16`	H-by-W-by-3	RGB48 image
`'RGB48 Signed'`	`int16`	H-by-W-by-3	RGB48 signed image

Examples

Read and Play Back Movie File

Read and play back the sample movie file, `xylophone.mp4`.

Construct a `VideoReader` object to read data from the sample file. Then, determine the width and height of the video.

```
xyloObj = VideoReader('xylophone.mp4');

vidWidth = xyloObj.Width;
vidHeight = xyloObj.Height;
```

Create a movie structure array, `mov`.

```
mov = struct('cdata',zeros(vidHeight,vidWidth,3,'uint8'),...
    'colormap',[]);
```

Read one frame at a time until the end of the video is reached.

```
k = 1;
while hasFrame(xyloObj)
    mov(k).cdata = readFrame(xyloObj);
    k = k+1;
end
```

Size a figure based on the video's width and height. Then, play back the movie once at the video's frame rate.

```
hf = figure;
set(hf,'position',[150 150 vidWidth vidHeight]);

movie(hf,mov,1,xyloObj.FrameRate);
```

- "Read Video Files"

See Also

`movie` | `VideoReader`

read

Class: Tiff

Read entire image

Syntax

```
imageData = read(tiffobj)
[Y,Cb,Cr] = read(tiffobj)
```

Description

`imageData = read(tiffobj)` reads the image data from the current image file directory (IFD) in the TIFF file associated with the `Tiff` object, `tiffobj`.

`[Y,Cb,Cr] = read(tiffobj)` reads the YCbCr component data from the current directory in the TIFF file. Depending upon the values of the `YCbCrSubSampling` tag, the size of the `Cb` and `Cr` channels might differ from the `Y` channel.

Examples

Open a `Tiff` object and read data from the TIFF file:

```
t = Tiff('example.tif','r');
imageData = read(t);
close(t)
```

See Also

`Tiff.write` | `imread`

read

Read data from remote host over TCP/IP interface

Syntax

```
read(t)
read(t,size)
read(t,size,datatype)
```

Description

`read(t)` reads all available bytes of data from `tcpclient` object `t` connected to the remote host and returns the data. The number of values read is determined by the `BytesAvailable` property.

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

`read(t,size)` reads the specified number of values, `size`, from `tcpclient` object `t` connected to the remote host and returns the data. If `size` is greater than the object's `BytesAvailable` property, then the function waits until the specified amount of data is read or the `timeout` is reached.

`read(t,size,datatype)` reads the specified number of values, `size`, with the specified precision, `datatype`, from `tcpclient` object `t` connected to the remote host and returns the data. The `datatype` argument is a character string of a standard MATLAB data type.

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

Examples

Read All Available Data from Host

Create a TCP/IP object called `t`, using the IP address shown, and Port of `4012`.

```
t = tcpclient('172.28.154.231', 4012)
```

```
t =

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 10
   BytesAvailable: 0
```

Read all the bytes of data available.

```
read(t)
```

The `read` function used with no arguments reads all available bytes of data from `tcpclient` object `t` connected to the remote host and returns the data. The number of values read is determined by the `BytesAvailable` property, which is equal to the numbers of bytes available in the input buffer.

Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Specify the Size and Data Type to Read

Create a TCP/IP object called `t`, connecting to a TCP/IP echo server, with Port of `7`.

```
t = tcpclient('localhost', 7)
```

```
t =

   tcpclient with properties:

          Address: 'local host'
             Port: 7
          Timeout: 10
   BytesAvailable: 0
```

Assign 10 bytes of data to the variable `data`.

```
data = (1:10)
```

Check the data.

```
whos data
```

```
Name      Size      Bytes      Class      Attributes

data      1x10         10      double
```

Write data to the echo server.

```
write(t, data)
```

Check that the data was written using the `BytesAvailable` property.

```
t.BytesAvailable
```

```
ans =

    80
```

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type. Since 1 `double` equals 8 `uint8` bytes, there are 80 bytes available.

Read 10 doubles from the server. The object name is always the first argument. The `size` argument must be the second argument, and `datatype` must be the third argument.

```
read(t, 10, 'double')
```

```
ans =

  1    2    3    4    5    6    7    8    9    10
```

Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Input Arguments

`size` — Number of bytes to read

numeric scalar

Number of bytes to read, specified as a numeric scalar. Size cannot be set to `inf`. If `size` is greater than the object's `BytesAvailable` property, the function waits until the specified amount of data is read. The first argument must be the object name, and the second argument is the size. The `size` argument is optional.

Example: `read(t, 5)`

Data Types: `double`

`datatype` — MATLAB data type

`'uint8'` (default) | `'int8'` | `'uint16'` | `'int16'` | `'uint32'` | `'int32'` | `'uint64'` | `'int64'` | `'single'` | `'double'`

MATLAB data type, specified as a character string. Size cannot be set to `inf`. The `datatype` must be set to one of the 10 values shown above. The first argument must be the object name, the second argument is the size, and the third argument is the data type. The `size` and `datatype` arguments are optional.

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

Example: `read(t, 10, 'double')`

Data Types: `char`

readasync

Read data asynchronously from device

Syntax

```
readasync(obj)
readasync(obj,size)
```

Description

`readasync(obj)` initiates an asynchronous read operation on the serial port object, `obj`.

`readasync(obj,size)` asynchronously reads, at most, the number of bytes given by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Examples

This example creates the serial port object `s` on a Windows platform. It connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
```

```
ans =
     15

out = fscanf(s)

out =
2.0399999619E0

fclose(s)
```

More About

Tips

Before you can read data, you must connect `obj` to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

Only use `readasync` to configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute a callback function when the terminator or the specified amount of data is read.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled (if `size` is not specified).

Because `readasync` checks for the terminator, this function can be slow. To increase speed, you might want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the device.

See Also

`fopen` | `stopasync` | `BytesAvailable` | `BytesAvailableFcn` | `ReadAsyncMode` | `Status` | `TransferStatus`

Introduced before R2006a

readEncodedStrip

Class: Tiff

Read data from specified strip

Syntax

```
stripData = readEncodedStrip(tiffobj,stripNumber)
[Y,Cb,Cr] = readEncodedStrip(tiffobj,stripNumber)
```

Description

`stripData = readEncodedStrip(tiffobj,stripNumber)` reads data from the strip specified by `stripNumber`. Strip numbers are one-based numbers.

`[Y,Cb,Cr] = readEncodedStrip(tiffobj,stripNumber)` reads YCbCr component data from the specified strip. The size of the chrominance components `Cb` and `Cr` might differ from the size of the luminance component `Y` depending on the value of the `YCbCrSubSampling` tag.

`readEncodeStrip` clips the last strip, if the strip extends past the `ImageLength` boundary.

Examples

Read a Strip

Read the first strip in the second image of a TIFF file.

Create a `Tiff` object associated with the example file, `example.tif`, and make the second image the current directory.

```
t = Tiff('example.tif','r');
setDirectory(t,2)
```

Read the data in the first strip. Then, close the `Tiff` object.

```
data = readEncodedStrip(t,1);
close(t)
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

References

This method corresponds to the `TIFFReadEncodedStrip` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.readEncodedTile` | `Tiff.isTiled`

readEncodedTile

Class: Tiff

Read data from specified tile

Syntax

```
tileData = readEncodedTile(tiffobj,tileNumber)
[Y,Cb,Cr] = readEncodedTile(tiffobj,tileNumber)
```

Description

`tileData = readEncodedTile(tiffobj,tileNumber)` reads data from the tile specified by `tileNumber`. Tile numbers are one-based numbers.

`[Y,Cb,Cr] = readEncodedTile(tiffobj,tileNumber)` reads YCbCr component data from the specified tile. The size of the chrominance components `Cb` and `Cr` might differ from the size of the luminance component `Y`, depending on the value of the `YCbCrSubSampling` tag.

`readEncodedTile` clips tiles on the last row or right-most column of an image if the tile extends past the `ImageLength` and `ImageLength` boundaries.

Examples

Read a Tile

Create a `Tiff` object associated with the example file, `example.tif`. Then, read the first tile of data.

```
t = Tiff('example.tif','r');
data = readEncodedTile(t,1);
```

```
close(t)
```

References

This method corresponds to the `TIFFReadEncodedTile` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.readEncodedStrip` | `Tiff.isTiled`

readRGBAImage

Class: Tiff

Read image using RGBA interface

Syntax

```
[RGB,alpha] = readRGBAImage()
```

Description

`[RGB,alpha] = readRGBAImage()` reads an entire image using the RGBA interface. `RGB` consists of an `m`-by-`n`-by-3 colormetric image, where `m` and `n` are the height and width of the tile, respectively. `alpha` is the associated alpha matting. If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to 255 (transparent).

The pixel values may be transformed depending upon the values of the following tags:

`PhotometricInterpretation`

`BitsPerSample`

`SamplesPerPixel`

`Orientation`

`ExtraSamples`

`ColorMap`

Examples

Return all image data as RGB, with associated alpha matting.

```
t = Tiff('example.tif','r');
t.setDirectory(2);
[RGB,A] = t.readRGBAImage();
t.close();
```

References

This method corresponds to the `TIFFReadRGBAImage` function in the LibTIFF C API.

To use this method, you must be familiar with LibTIFF version 4.0.0, as well as the TIFF specification and technical notes. View this documentation at LibTiff - TIFF Library and Utilities.

See Also

`Tiff.readRGBAStrip` | `Tiff.read` | `Tiff.readRGBATile`

readRGBAStrip

Class: Tiff

Read strip data using RGBA interface

Syntax

```
[RGB,alpha] = readRGBAStrip(row)
```

Description

`[RGB,alpha] = readRGBAStrip(row)` reads a strip using the RGBA interface. `row` is a one-based number of any row contained by the strip. `RGB` consists of an `m`-by-`n`-by-3 colormetric image, where `m` and `n` are the height and width of the strip, respectively. `alpha` is the associated alpha matting. If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to 255 (transparent).

The strip is clipped if the strip boundary extends past the end of the image.

The pixel values may be transformed depending upon the values of the following tags:

`PhotometricInterpretation`
`BitsPerSample`
`SamplesPerPixel`
`Orientation`
`ExtraSamples`
`ColorMap`

Examples

Open a `Tiff` object and read the strip of data that contains the first row, using the RGBA interface, from the example file, `example.tif`

```
t = Tiff('example.tif','r');
```

```
t.setDirectory(2);
[RGB,A] = t.readRGBAStrip(1);
t.close();
```

References

This method corresponds to the `TIFFReadRGBAStrip` function in the LibTIFF C API.

To use this method, you must be familiar with LibTIFF version 4.0.0, as well as the TIFF specification and technical notes. View this documentation at LibTiff - TIFF Library and Utilities.

See Also

`Tiff.readRGBATile` | `Tiff.readRGBAImage`

readRGBATile

Class: Tiff

Read tile data using RGBA interface

Syntax

```
[RGB,alpha] = readRGBATile(row,col)
```

Description

`[RGB,alpha] = readRGBATile(row,col)` reads a tile using the RGBA interface. `row` and `col` are the one-based row and column numbers of any pixel in the requested tile. `RGB` consists of an `m`-by-`n`-by-3 colormetric image, where `m` and `n` are the height and width of the tile, respectively. `alpha` is the associated alpha matting. If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to 255 (transparent).

The tile is clipped if the tile boundaries extend past the edges of the image.

The pixel values may be transformed depending upon the values of the following tags:

```
PhotometricInterpretation
BitsPerSample
SamplesPerPixel
Orientation
ExtraSamples
ColorMap
```

Examples

Open a `Tiff` object and read the first tile using the RGBA interface.

```
t = Tiff('example.tif','r');
```

```
t.setDirectory(1);
[RGB,A] = t.readRGBATile(1,1);
t.close();
```

References

This method corresponds to the `TIFFReadRGBATile` function in the LibTIFF C API.

To use this method, you must be familiar with LibTIFF version 4.0.0, as well as the TIFF specification and technical notes. View this documentation at LibTiff - TIFF Library and Utilities.

See Also

`Tiff.readRGBAStrip` | `Tiff.readRGBAImage`

Remove

Convenience function for static .NET System.Delegate Remove method

Syntax

```
result = Remove(combinedDelegate,removedDelegate)
```

Description

`result = Remove(combinedDelegate,removedDelegate)` removes last instance of the `removedDelegate` delegate from the `combinedDelegate` delegate.

Input Arguments

`combinedDelegate`

.NET `System.Delegate` object. The combined delegate from which to remove the `removedDelegate` delegate.

Default:

`removedDelegate`

.NET `System.Delegate` object. The delegate to remove from the `combinedDelegate` delegate.

Default:

Output Arguments

`result`

.NET `System.Delegate` object. A new delegate which is the same as the `combinedDelegate` delegate except without the last instance of the `removedDelegate` delegate.

Alternatives

Use the static `Remove` method of the `System.Delegate` class.

More About

- "Combine and Remove .NET Delegates"
- MSDN System.Delegate.Remove Method reference page

See Also

`RemoveAll` | `Combine`

Introduced in R2011a

RemoveAll

Convenience function for static .NET System.Delegate RemoveAll method

Syntax

```
result = RemoveAll(combinedDelegate,removedDelegate)
```

Description

`result = RemoveAll(combinedDelegate,removedDelegate)` removes all instances of `removedDelegate` from `combinedDelegate`.

Input Arguments

`combinedDelegate`

.NET `System.Delegate` object. The combined delegate from which to remove all instances of the `removedDelegate` delegate.

Default:

`removedDelegate`

.NET `System.Delegate` object. The delegate to remove from the `combinedDelegate` delegate.

Default:

Output Arguments

`result`

.NET `System.Delegate` object. A new delegate which is the same as the `combinedDelegate` delegate except without all instances of the `removedDelegate` delegate.

Alternatives

Use the static `RemoveAll` method of the `System.Delegate` class.

More About

- "Combine and Remove .NET Delegates"
- MSDN System.Delegate.RemoveAll Method reference page

See Also

`Remove` | `Combine`

Introduced in R2011a

timeseries class

Create `timeseries` object

Description

Time series are data vectors sampled over time, in order, often at regular intervals. They are distinguished from randomly sampled data that form the basis of many other data analyses. Time series represent the time-evolution of a dynamic population or process. The linear ordering of time series gives them a distinctive place in data analysis, with a specialized set of techniques. Time series analysis is concerned with:

- Identifying patterns
- Modeling patterns
- Forecasting values

Construction

`ts = timeseries` creates an empty time-series object.

`ts = timeseries(tsname)` creates an empty time-series object using the name, tsname, for the time-series object. This name can differ from the time-series variable name.

`ts = timeseries(data)` creates the time-series object using the specified data.

`ts = timeseries(data,time)` creates the time-series object using the specified data and time.

`ts = timeseries(data,time,quality)` specifies quality in terms of codes defined by `QualityInfo.Code`.

`ts = timeseries(data,'Name',tsname)` creates the time-series object using the specified data and the name, tsname.

`ts = timeseries(data,time,'Name',tsname)` creates the time-series object using the specified data, time, and the name, tsname.

`ts = timeseries(data,time,quality,'Name',tsname)` uses the specified quality and the name, tsname.

Input Arguments

`data`

The time-series data, which can be an array of samples

`tsname`

Time-series name specified as a string

Default: `''`

`time`

The time vector.

When time values are date strings, you must specify `Time` as a cell array of date strings. When the time vector contains duplicate values:

- Duplicated values must occupy contiguous elements.
- Time values must not be decreasing.

Interpolating time-series data using methods like `resample` and `synchronize` can produce different results depending on whether the input `timeseries` contains duplicate times.

Default: A time vector that ranges from `0` to `N-1` with a 1-second interval, where `N` is the number of samples.

`quality`

An integer vector with values `-128` to `127` that specifies the quality in terms of codes defined by `QualityInfo.Code`

When `Quality` is a vector:

- `Quality` must have the same length as the time vector.

- Each `Quality` value applies to the corresponding data sample.

When `Quality` is an array:

- `Quality` must have the same size as the data array.
- Each `Quality` value applies to the corresponding data value of the `ts.data` array.

Properties

`Data`

Time-series data, where each data sample corresponds to a specific time

The data can be a scalar, a vector, or a multidimensional array. Either the first or last dimension of the data must align with `Time`.

By default, `NaN`s represent missing or unspecified data. Set the `TreatNaNasMissing` property to determine how missing data is treated in calculations.

Attributes:

`Dependent`	`true`

`DataInfo`

Contains fields for storing contextual information about `Data`:

- `Unit` — String that specifies data units
- `Interpolation` — A `tsdata.interpolation` object that specifies the interpolation method for this `timeseries` object.

  Fields of the `tsdata.interpolation` object include:

  - `Fhandle` — Function handle to a user-defined interpolation function
  - `Name` — String that specifies the name of the interpolation method. Predefined methods include `'linear'` and `'zoh'` (zero-order hold). `'linear'` is the default.
- `UserData` — Any user-defined information entered as a string

Events

An array of `tsdata.event` objects that stores event information for this `timeseries` object.

You add events by using the `addevent` method. Fields of the `tsdata.event` object include the following:

- `EventData` — Any user-defined information about the event
- `Name` — String that specifies the name of the event
- `Time` — Time value when this event occurs, specified as a real number or a date string
- `Units` — Time units
- `StartDate` — A reference date specified in MATLAB date-string format. `StartDate` is empty when you have a numerical (non-date-string) time vector.

IsTimeFirst

Logical value (`true` or `false`) specifies whether the time vector is aligned with the first or last dimension of the `Data` array. The value is `false` for 3-D and higher dimensional data and `true` otherwise.

- `true` — The first dimension of the data array is aligned with the time vector. For example, `ts = timeseries(rand(3,3),1:3);`
- `false` — The last dimension of the data array is aligned with the time vector. For example: `ts = timeseries(rand(3,4,5),1:5);`

Attributes:

`Dependent`	`true`
`SetAccess`	`'protected'`

Length

Length of the time vector in the `timeseries` object

Attributes:

`Dependent`	`true`

```
SetAccess                       'protected'
```

Name

The `timeseries` object name entered as a string, tsname.

This name can differ from the name of the `timeseries` variable in the MATLAB workspace.

Quality

An integer vector or array containing values `-128` to `127` that specify the quality in terms of codes defined by `QualityInfo.Code`.

When `Quality` is a vector, it must have the same length as the time vector. In this case, each `Quality` value applies to a corresponding data sample.

When `Quality` is an array, it must have the same size as the data array. In this case, each `Quality` value applies to the corresponding value of the data array.

Attributes:

```
Dependent                       true
```

QualityInfo

Provides a lookup table that converts numerical `Quality` codes to readable descriptions.

`QualityInfo` fields include the following:

- `Code` — Integer vector containing values `-128` to `127` that define the "dictionary" of quality codes. You can assign one of these integer values to each `Data` value by using the `Quality` property.
- `Description` — Cell vector of strings, where each element provides a readable description of the associated quality `Code`.
- `UserData` — Stores any additional user-defined information.

Lengths of `Code` and `Description` must match.

Time

Array of time values.

When `TimeInfo.StartDate` is empty, the numerical `Time` values are measured relative to `0` in specified units. When `TimeInfo.StartDate` is defined, the time values are date strings measured relative to the `StartDate` in specified units.

The length of `Time` must be the same as either the first or the last dimension of `Data`. When the data contains three or more dimensions, the length of `Time` matches the size of the last data dimension. Otherwise, the length of `Time` matches the size of the first data dimension.

Attributes:

`Dependent`	`true`

`TimeInfo`

Uses the following fields for storing contextual information about `Time`:

- `Units` — Time units having any of following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, or `'nanoseconds'`
- `Start` — Start time
- `End` — End time (read only)
- `Increment` — Interval between two subsequent time values
- `Length` — Length of the time vector (read only)
- `Format` — String defining the date string display format. See the MATLAB `datestr` function reference page for more information.
- `StartDate` — Date string defining the reference date. See the MATLAB `setabstime` function reference page for more information.
- `UserData` — Stores any additional user-defined information

`TreatNaNasMissing`

Logical value that specifies how to treat `NaN` values in `Data`:

- `true` — (Default) Treats all `NaN` values as missing data except during statistical calculations.
- `false` — Includes `NaN` values in statistical calculations, in which case `NaN` values are propagated to the result.

`UserData`

Generic field for data of any class that you want to add to the object.

Default: [ ]

Methods

Time-Series Methods

.
.
.
.
.

Methods to Query and Set Object Properties and Plot the Data

`get`	Query `timeseries` object property values.
`getdatasamplesize`	Return the size of each data sample in a `timeseries` object.
`getqualitydesc`	Return data quality descriptions based on the `Quality` property values assigned to a `timeseries` object.
`plot`	Plot the `timeseries` object.
`set`	Set `timeseries` property values.

Methods to Manipulate Data and Time

`addsample`	Add a data sample to a `timeseries` object.
`append`	Concatenate `timeseries` objects in the time dimension.
`delsample`	Delete a sample from a `timeseries` object.
`detrend`	Subtract the mean or best-fit line and remove all `NaN`s from time-series data.
`filter`	Shape frequency content of time-series data using a 1-D digital filter.

`getabstime`	Extract a date-string time vector from a `timeseries` object into a cell array.
`getdatasamples`	Extract a subset of data samples from an existing `timeseries` object into an array using a subscripted indexed array.
`getsamples`	Extract a subset of data samples from an existing `timeseries` object into a new `timeseries` object using a subscript indexed array.
`getinterpmethod`	Get the interpolation method for a `timeseries` object.
`getsampleusingtime`	Extract data samples from an existing `timeseries` object into a new `timeseries` object based on specified start and end time values.
`idealfilter`	Apply an ideal pass or notch (noncausal) filter to a `timeseries` object.
`resample`	Select or interpolate data in a `timeseries` object using a new time vector.
`setabstime`	Set the time values in the time vector as date strings.
`setinterpmethod`	Set interpolation method for a `timeseries` object.
`setuniformtime`	Assign uniform time vector to `timeseries` object.
`synchronize`	Synchronize and resample two `timeseries` objects using a common time vector.

Event Methods

To construct an event object, use the constructor `tsdata.event`. For an example of defining events for a time-series object, see "Defining Events".

`addevent`	Add one or more events to a `timeseries` object.
`delevent`	Delete one or more events from a `timeseries` object.
`gettsafterateevent`	Create a new `timeseries` object by extracting the samples from an existing time series that occur after or at a specified event.
`gettsafterevent`	Create a new `timeseries` object by extracting the samples that occur after a specified event from an existing time series.

`gettsatevent`	Create a new `timeseries` object by extracting the samples that occur at the same time as a specified event from an existing time series.
`gettsbeforeatevent`	Create a new `timeseries` object by extracting the samples that occur before or at a specified event from an existing time series.
`gettsbeforeevent`	Create a new `timeseries` object by extracting the samples that occur before a specified event from an existing time series.
`gettsbetweenevents`	Create a new `timeseries` object by extracting the samples that occur between two specified events from an existing time series.

Methods to Arithmetically Combine `timeseries` Objects

`+`	Addition of the corresponding data values of `timeseries` objects.
`-`	Subtraction of the corresponding data values of `timeseries` objects.
`.*`	Element-by-element multiplication of `timeseries` data.
`*`	Matrix-multiply `timeseries` data.
`./`	Right element-by-element division of `timeseries` data.
`/`	Right matrix division of `timeseries` data.
`.\`	Element-by-element left-array divide of `timeseries` data.
`\`	Left matrix division of `timeseries` data.

Methods to Calculate Descriptive Statistics for a `timeseries` Object

`iqr`	Return the interquartile range of `timeseries` data.
`max`	Return the maximum value of `timeseries` data.
`mean`	Return the mean of `timeseries` data.
`median`	Return the median of `timeseries` data.
`min`	Return the minimum of `timeseries` data.
`std`	Return the standard deviation of `timeseries` data.

`sum`	Return the sum of `timeseries` data.
`var`	Return the variance of `timeseries` data.

Definitions

timeseries

The time-series object, called `timeseries`, is a MATLAB variable that contains time-indexed data and properties in a single, coherent structure. For example, in addition to data and time values, you can also use the time-series object to store events, descriptive information about data and time, data quality, and the interpolation method.

Data Sample

A time-series *data sample* consists of one or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

For example, suppose that `ts.data` has the size 3-by-4-by-5 and the time vector has the length 5. Then, the number of samples is `5` and the total number of data values is `3 x 4 x 5 = 60`.

Time Vector

A time vector of a `timeseries` object can be either numerical (`double`) values or valid MATLAB date strings.

When the `timeseries` `TimeInfo.StartDate` property is empty, the numerical `time` values are measured relative to 0 (or another numerical value) in specified units. In this case, the time vector is described as *relative* (that is, it contains time values that are not associated with a specific start date).

When `TimeInfo.StartDate` is nonempty, the time values are date strings measured relative to `StartDate` in specified units. In this case, the time vector is described as *absolute* (that is, it contains time values that are associated with a specific calendar date).

MATLAB supports the following date-string formats for time-series applications.

Date-String Format	Usage Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `timeseries` object called `'LaunchData'` that contains four data sets, each stored as a column of length `5` and using the default time vector:

```
b = timeseries(rand(5, 4),'Name','LaunchData')
```

Create a `timeseries` object containing a single data set of length `5` and a time vector starting at `1` and ending at `5`:

```
b = timeseries(rand(5,1),[1 2 3 4 5])
```

Create a `timeseries` object called `'FinancialData'` containing five data points at a single time point:

```
b = timeseries(rand(1,5),1,'Name','FinancialData')
```

See Also

`tscollection` | `tsdata.event`

Introduced before R2006a

addsample

Class: timeseries

Add data sample to `timeseries` object

Syntax

```
ts1 = addsample(ts, s)
ts1 = addsample(ts, 'Data', data-value, 'Time',time-value,...,
Name,Value)
```

Description

`ts1 = addsample(ts, s)` adds one or more new data samples stored in a structure s to the `timeseries` object ts.

`ts1 = addsample(ts, 'Data', data-value, 'Time',time-value,...,Name,Value)` adds one or more data samples to the `timeseries` object ts along with additional options specified by one or more Name,Value pair arguments.

Tips

- If `N` is the number of data samples, you can get the sample size of each time with `SampleSize = getsamplesize(ts)`.

  When `ts.IsTimeFirst` is `true`, the size of the data is `N`-by-`SampleSize`. When `ts.IsTimeFirst` is `false`, the size of the data is `SampleSize`-by-`N`.

Input Arguments

s

A structure that you must define before passing as an argument to `addsample`. It consists of the following optional fields:

- `s.data`
- `s.time`
- `s.quality`
- `s.overwriteflag`

`data-value`

A numeric data value.

`time-value`

A valid time vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Quality'`

Array of data quality codes.

Default: `[ ]`

`'OverwriteFlag'`

Logical value that controls whether to overwrite a data sample at the same time with the new sample you are adding to your `timeseries` object. When set to `true`, the new sample overwrites the old sample at the same time.

Default: `false`

Output Arguments

`ts1`

The `timeseries` object that results when you add the specified samples to the original `timeseries` object.

Definitions

data sample

One or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

Examples

Add a data value of `420` at time `3`:

```
ts = ts.addsample('Time',3,'Data',420);
```

Add a data value of `420` at time `3` and specify quality code `1` for this data value. Set the `OverwriteFlag` to overwrite an existing value at time `3`.

```
ts = ts.addsample('Data',3.2,'Quality',1,'OverwriteFlag',...
        true,'Time',3);
```

See Also

`timeseries` | `delsample` | `getdatasamples`

Introduced before R2006a

append

Class: timeseries

Concatenate time series objects in time dimension

Syntax

```
ts = append(ts1,ts2, ... tsn)
```

Description

`ts = append(ts1,ts2, ... tsn)` creates a new `timeseries` object by concatenating `timeseries` `ts1`, `ts2`, and so on, along the time dimension.

Tips

- A single overlapping time between each input time series is valid, as long as the overlapping samples are identical.
- The time vectors must not overlap by a nonzero amount. That is, the last time in `ts1` must be earlier than or equal to the first time in `ts2`.
- The sample size of the time series must be the same.

Input Arguments

`ts1`

The first `timeseries` object that you want to append.

`ts2`

The second `timeseries` object that you want to append.

`tsn`

The nth `timeseries` object that you want to append.

Output Arguments

`ts`

The `timeseries` object that results from appending the input `timeseries` objects.

Examples

After creating `timeseries` objects, `ts1` and `ts2`, append them:

```
ts1 = timeseries(rand(5,1),[1 2 3 4 5]);
ts2 = timeseries(rand(5,1),[6 7 8 9 10]);
ts3 = append(ts1, ts2)
```

See Also

`timeseries`

ctranspose

Class: timeseries

Transpose `timeseries` object

Syntax

```
ts1 = ctranspose(ts)
```

Description

`ts1 = ctranspose(ts)` returns a new `timeseries` object ts1 with the `IsTimeFirst` value set to opposite of what it is for ts. For example, if `ts` has the first data dimension aligned with the time vector, `ts1` has the last data dimension aligned with the time vector as a result of this operation.

Tips

- The overloaded `ctranspose` method for `timeseries` objects does not transpose the data. Instead, this method changes whether the first or the last dimension of the data aligns with the time vector. To transpose the data, you must transpose the `Data` property of the `timeseries` object. For example, you can use the syntax `ctranspose(ts.Data)` or `(ts.Data)'`. The `Data` property value must be a 2-D array.
- Consider a `timeseries` object with 10 samples with the property `IsTimeFirst = True`. When you transpose this object, the data size changes from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

  The following table summarizes the size for Data property of the `timeseries` object (up to three dimensions) before and after transposing.

  Data Size Before and After Transposing

Size of Original Data	Size of Transposed Data
N-by-1	1-by-1-by-N

Size of Original Data	Size of Transposed Data
N-by-M	M-by-1-by-N
N-by-M-by-L	M-by-L-by-N

Input Arguments

`ts`

The `timeseries` object you want to transpose.

Output Arguments

`ts1`

The transposed `timeseries` object.

Examples

Suppose that a `timeseries` object `ts` has `ts.data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is set to `true`, which means that the first dimension of the data is aligned with the time vector. `ctranspose(ts)` modifies `ts`, such that the last dimension of the data is now aligned with the time vector. This permutes the data, such that the size of `ts.Data` becomes 3-by-2-by-10.

See Also

`timeseries` | `transpose`

delsample

Class: timeseries

Remove sample from `timeseries` object

Syntax

```
ts1 = delsample(ts,Name,Value)
```

Description

`ts1 = delsample(ts,Name,Value)` deletes samples from the `timeseries` object ts based on the specified Name,Value pair arguments.

Input Arguments

`ts`

A `timeseries` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Index'`

The indices of the time vector that correspond to the samples you want to delete.

`'Value'`

The time values that correspond to the samples you want to delete.

Output Arguments

`ts1`

The `timeseries` object that results from removing the specified samples.

Examples

Create a `timeseries` object, and then remove samples:

```
ts = timeseries(rand(5,1),[10 20 30 40 50]);

% Remove data sample at time index 1

ts1 = delsample(ts,'Index', 1)

% Remove data sample at time value 20:

ts2 = delsample(ts,'Value', [20])
```

See Also

`set` | `timeseries`

Introduced before R2006a

detrend

Class: timeseries

Subtract mean or best-fit line and all NaNs from `timeseries` object

Syntax

```
ts1 = detrend(ts, method)
ts1 = detrend(ts, method, index)
```

Description

`ts1 = detrend(ts, method)` subtracts either a mean or a best-fit line from time-series data, using the specified method. Usually for FFT processing.

`ts1 = detrend(ts, method, index)` uses the optional index to specify the columns or rows to detrend.

Tips

- You cannot apply `detrend` to`timeseries` data with more than two dimensions.

Input Arguments

`ts`

The `timeseries` object from which you want to subtract the mean or best-fit line and all NaNs.

Default:

`method`

A string that specifies one of the following detrend methods:

- `'constant'` — Subtracts the mean.
- `'linear'` — Subtracts the best-fit line.

`index`

An integer array that specifies the columns or rows to detrend when `ts.IsTimeFirst` is true.

Output Arguments

`ts1`

The `timeseries` object resulting from detrending the input `timeseries` object.

See Also

`timeseries`

Introduced before R2006a

filter

Class: timeseries

Shape frequency content of time-series

Syntax

```
ts1 = filter(ts, numerator, denominator)
ts1=filter(ts, numerator, denominator, index)
```

Description

`ts1 = filter(ts, numerator, denominator)` applies the transfer function filter $b(z^{-1})/a(z^{-1})$ to the data in the `timeseries` object ts. b and a are the coefficient arrays of the transfer function numerator and denominator, respectively.

`ts1=filter(ts, numerator, denominator, index)` uses the optional index integer array to specify either the columns or rows to filter, depending on the value of `ts.IsTimeFirst`.

Tips

- The time-series data must be uniformly sampled to use this filter.
- The following function

  ```
  y = filter(b,a,x)
  ```

  creates filtered data `y` by processing the data in vector `x` with the filter described by vectors `a` and `b`.
- The `filter` function is a general tapped delay-line filter, described by the difference equation:
  $a(1)y(n) = b(1)x(n) + b(2)x(n-1) + ... + b(nb)x(n-nb+1) - a\,(2)y(n-1) - ... - a(N_a)y(n-N_b+1)$.

  Here, n is the index of the current sample, N_a is the order of the polynomial described by vector `a`, and N_b is the order of the polynomial described by vector `b`. The output

$y(n)$ is a linear combination of current and previous inputs, $x(n)$ $x(n-1)$..., and previous outputs, $y(n-1)$ $y(n-2)$... .

- You use the discrete filter to shape the data by applying a transfer function to the input signal.

  Depending on your objectives, the transfer function you choose might alter both the amplitude and the phase of the variations in the data at different frequencies to produce either a smoother or a rougher output.

- In digital signal processing (DSP), it is customary to write transfer functions as rational expressions in z^{-1} and to order the numerator and denominator terms in ascending powers of z^{-1}.

  Taking the z-transform of the difference equation
  $a(1)y(n) = b(1)x(n) + b(2)x(n-1) + ... + b(nb)x(n - nb + 1) - a(2)y(n-1) - ... - a(na)y(na + 1)$,

  results in the transfer function

$$Y(z) = H(z^{-1})X(z) = \frac{b(1) + b(2)z^{-1} + ... + b(nb)z^{-nb+1}}{a(1) + a(2)z^{-1} + ... + a(na)z^{-na+1}} X(z),$$

  where $Y(z)$ is the z-transform of the filtered output $y(n)$. The coefficients b and a are unchanged by the z-transform.

Input Arguments

`ts`

The first `timeseries` object for which you want to shape the frequency content.

`numerator`

The coefficient array of the transfer function numerator.

`denominator`

The coefficient array of the transfer function denominator.

index

An integer array that specifies the columns or rows to filter when `ts.IsTimeFirst` is true.

Output Arguments

ts1

The `timeseries` object that results from filtering the input `timeseries` object.

Examples

Apply Transfer Function to Time Series Data

This example applies the following transfer function to the data in `count.dat`:

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}$$

Load the matrix `count` into the workspace:

```
load count.dat
```

Create a time-series object based on this matrix:

```
count1 = timeseries(count(:,1),[1:24]);
```

Enter the coefficients of the denominator ordered in ascending powers of z^{-1} to represent $1 + 0.2x$:

```
a = [1 0.2];
```

Enter the coefficients of the numerator to represent $2 - 3z^{-1}$:

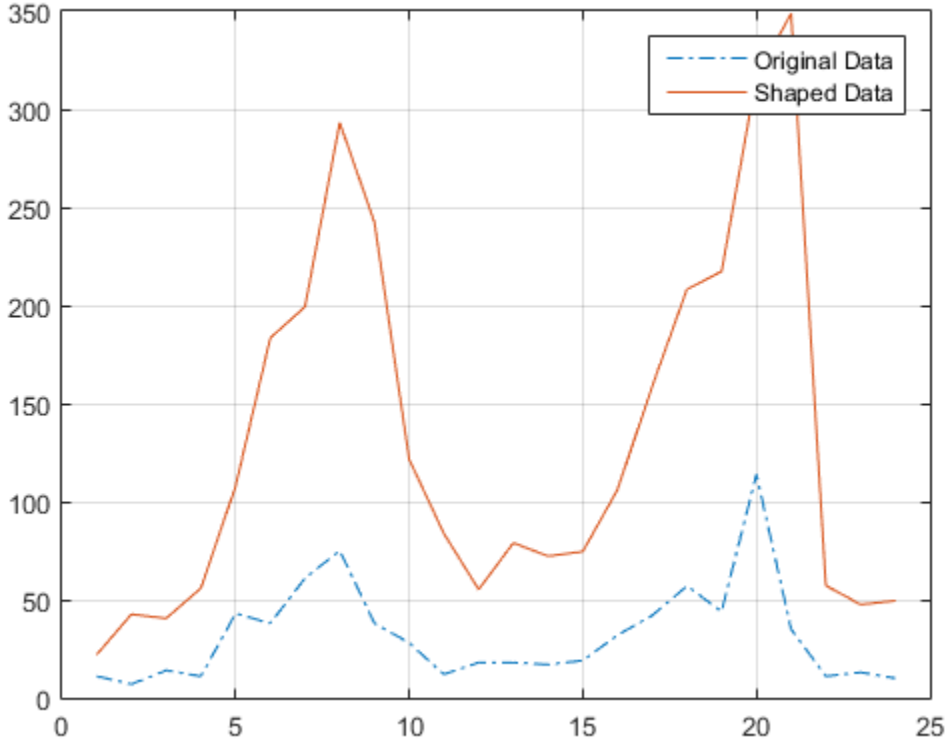
```
b = [2 3];
```

Call the filter method:

```
filter_count = filter(count1, b, a);
```

Compare the original data and the shaped data with an overlaid plot of the two curves:

```
figure
plot(count1,'-.')
grid on
hold on
plot(filter_count,'-')
legend('Original Data','Shaped Data',2)
```



See Also

`timeseries` | `idealfilter`

Introduced before R2006a

get

Class: timeseries

Query `timeseries` object property values

Syntax

```
get(ts)
value = get(ts, PropertyName)
```

Description

`get(ts)` displays all properties and values of the `timeseries` object, ts.

`value = get(ts, PropertyName)` returns the property value for the specified `timeseries` object. The following syntax is equivalent:

```
value = ts.PropertyName
```

Input Arguments

`ts`

A `timeseries` object.

`PropertyName`

String specifying the name of a `timeseries` property. For a list of `timeseries` properties, see `timeseries`.

Output Arguments

`value`

String containing the value associated with the specified property.

Examples

Create a `timeseries` object, and then get the name. This example gets the name three different ways:

```
ts1 = timeseries(rand(5,1),[1 2 3 4 5], 'Name', 'MyTimeseries');
get(ts1)
get(ts1, 'Name')
value = ts1.Name
```

See Also

`timeseries` | `set`

Introduced before R2006a

getabstime

Class: timeseries

Extract date-string time vector into cell array

Syntax

```
getabstime(ts)
```

Description

`getabstime(ts)` extracts the time vector from the `timeseries` object ts as a cell array of date strings.

Tips

- To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the `timeseries` object. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Input Arguments

`ts`

The `timeseries` object from which you want to extract the time vector.

Default:

Examples

The following example extracts a time vector as a cell array of date strings from a `timeseries` object.

First, create a `timeseries` object.

```
ts = timeseries([3 6 8 0 10]);
```

The default time vector for `ts` is `[0 1 2 3 4]`, which starts at 0 and increases in 1-second increments. The length of the time vector is equal to the length of the data.

Next, set the `StartDate` property.

```
ts.TimeInfo.StartDate = '10/27/2005 07:05:36';
```

Extract the time vector.

```
getabstime(ts)
```

MATLAB returns:

```
'27-Oct-2005 07:05:36'
'27-Oct-2005 07:05:37'
'27-Oct-2005 07:05:38'
'27-Oct-2005 07:05:39'
'27-Oct-2005 07:05:40'
```

Change the date-string format of the time vector, and then extract the time vector with the new date-string format:

```
ts.TimeInfo.Format = 'mm/dd/yy';
getabstime(ts)
```

MATLAB returns:

```
'10/27/05'
'10/27/05'
'10/27/05'
'10/27/05'
'10/27/05'
```

See Also

`timeseries` | `setabstime` | `datestr`

Introduced before R2006a

getdatasamples

Class: timeseries

Returns subset of time series samples using subscripted index array

Syntax

```
datasamples = getdatasamples(ts, i)
```

Description

`datasamples = getdatasamples(ts, i)` returns an array corresponding to the samples indicated by the array, i.

Input Arguments

`ts`

The `timeseries` object from which you want to exact samples.

`i`

An array of linear indices or logical values that specifies the time value or values for which you want to extract the corresponding samples.

Output Arguments

`datasamples`

The array that results from extracting the samples corresponding to the time value or values, `ts.time(i)`.

Examples

After creating a `timeseries` object, `ts`, extract the second and third data samples into an array:

```
ts = timeseries(rand(5,1),[1 2 3 4 5]);
samples = getdatasamples(ts, [2 3])
```

See Also

timeseries | getsamples | resample

getdatasamplesize

Class: timeseries

Size of data sample in `timeseries` object

Syntax

```
getdatasamplesize(ts)
```

Description

`getdatasamplesize(ts)` returns the size of each data sample in a `timeseries` object.

Input Arguments

`ts`

String specifying the name of a `timeseries` object.

Definitions

data sample

One or more scalar values recorded at a specific time. The number of data samples is the same as the length of the time vector.

Examples

After loading data and creating a `timeseries` object, get the size of a data sample:

```
% Load a 24-by-3 data array:
load count.dat
```

```
% Create a timeseries object with 24 time values:
count_ts = timeseries(count,[1:24],'Name','VehicleCount')

% Get the size of the data sample for this timeseries object:
getdatasamplesize(count_ts)
```

MATLAB returns the following, which indicates that the size of each data sample in `count_ts` is 1-by-3. In other words, MATLAB stores each data sample as a row with three values.

```
ans =

     1     3
```

See Also

`timeseries` | `set`

Introduced before R2006a

getinterpmethod

Class: timeseries

Interpolation method for `timeseries` object

Syntax

```
getinterpmethod(ts)
```

Description

`getinterpmethod(ts)` returns the interpolation method that the `timeseries` object ts, uses as a string.

Input Arguments

`ts`

The `timeseries` object from which you want to extract the interpolation method.

Definitions

interpolation method

Predefined interpolation methods are zero-order hold, `zoh`, and linear interpolation, `linear`. Linear interpolation is the default.

Examples

Create a `timeseries` object, and then get its interpolation method:

```
ts = timeseries(rand(5));
```

```
getinterpmethod(ts)
```

MATLAB returns:

```
linear
```

See Also

`timeseries` | `setinterpmethod`

Introduced before R2006a

getqualitydesc

Class: timeseries

Data quality descriptions

Syntax

```
getqualitydesc(ts)
```

Description

`getqualitydesc(ts)` returns a cell array of data quality descriptions based on the `Quality` values you assigned to a `timeseries` object, ts.

Input Arguments

`ts`

A `timeseries` object.

Examples

Create a `timeseries` object, and then get the data quality description strings for `ts`:

```
% Create a timeseries object, ts, with Data, Time, and Quality
% values, respectively:

ts = timeseries([3; 4.2; 5; 6.1; 8], 1:5, [1; 0; 1; 0; 1]);

% Set the QualityInfo property, including Code and Description:

ts.QualityInfo.Code = [0 1];
ts.QualityInfo.Description = {'good' 'bad'};

% Get the data quality description strings for ts:
```

```
getqualitydesc(ts)
```

MATLAB returns:

```
ans =

    'bad'
    'good'
    'bad'
    'good'
    'bad'
```

See Also

`timeseries`

Introduced before R2006a

getsamples

Class: timeseries

Subset of time series samples using subscripted index array

Syntax

```
ts1 = getsamples(ts, i)
```

Description

`ts1 = getsamples(ts, i)` returns a new `timeseries` object by extracting samples from `timeseries` `ts` corresponding to the time or times indicated by the subscripted index array, i.

Input Arguments

`ts`

The `timeseries` object from which you want to exact samples.

`i`

A subscripted index array that specifies the time value or values for which you want to extract the corresponding samples.

Output Arguments

`ts1`

The `timeseries` object that results from extracting the samples corresponding to the time value or values `ts.time(i)` .

Examples

After creating a `timeseries` object, `ts`, extract the data samples at times 2 and 3 into a new `timeseries` object, `ts1`:

```
ts = timeseries(rand(5,1),[1 2 3 4 5]);
ts1 = getdatasamples(ts, ts.time([2 3]))
```

See Also

`timeseries` | `getdatasamples` | `resample`

getsampleusingtime

Class: timeseries

Extract data samples into new `timeseries` object

Syntax

```
ts1 = getsampleusingtime(ts,time)
ts1 = getsampleusingtime(ts,time,'AllowDuplicateTimes',true)
ts1 = getsampleusingtime(ts,starttime,endtime)
```

Description

`ts1 = getsampleusingtime(ts,time)` returns a new `timeseries` object, ts1, with a single sample corresponding to specified time in ts.

`ts1 = getsampleusingtime(ts,time,'AllowDuplicateTimes',true)` returns a new `timeseries` object, ts1, with multiple samples when the specified time occurs more than once in ts.

`ts1 = getsampleusingtime(ts,starttime,endtime)` returns a new `timeseries` object, ts1, with samples between the times starttime and endtime in ts.

Tips

- If the time vector in `ts` is not relative to a calendar date, then `starttime` and `endtime` must be numeric.
- If the time vector in `ts` is relative to a calendar date, then `starttime` and `endtime` values must be dates—either strings or `datenum` values.

Input Arguments

`ts`

The `timeseries` object from which you want to extract data samples.

`time`

The time corresponding to data sample you want to extract.

`starttime`

The time corresponding to the first data sample you want to extract.

`endtime`

The time corresponding to the last data sample you want to extract.

Output Arguments

`ts1`

A `timeseries` object that contains the subset of data samples from the original `timeseries` object.

See Also

`timeseries`

Introduced before R2006a

idealfilter

Class: timeseries

Apply ideal (noncausal) filter to `timeseries` object

Syntax

```
ts1 = idealfilter(ts, interval, filtertype)
ts1 = idealfilter(ts, interval, filtertype, index)
```

Description

`ts1 = idealfilter(ts, interval, filtertype)` applies an ideal filter of filtertype to one or more frequency intervals that interval specifies for the `timeseries` object, ts.

`ts1 = idealfilter(ts, interval, filtertype, index)` applies an ideal filter and uses the optional index integer array to specify the columns or rows to filter.

Tips

- Ideal filters require data to have a mean of zero and prepare the data by subtracting its mean. You can restore the filtered signal amplitude by adding the mean of the input data to the filter output values.
- Use the ideal *notch* filter when you want to remove variations in a specific frequency range. Alternatively, use the ideal *pass* filter to allow only the variations in a specific frequency range.
- If the time-series data is sampled nonuniformly, filtering resamples this data on a uniform time vector.
- All `NaN`s in the time series are interpolated before filtering, using the interpolation method you assigned to the `timeseries` object.

Input Arguments

`ts`

The `timeseries` object to which you want to apply an ideal filter.

`interval`

The frequency interval (specified in cycles per time unit) at which you want the ideal filter applied. To specify several frequency intervals, use an n-by-2 array of start and end frequencies, where `n` represents the number of intervals.

`filtertype`

A string specifying the type of filter you want to apply, either `pass` or `notch`.

`index`

An integer array that specifies the columns or rows to filter when `ts.IsTimeFirst` is true.

Output Arguments

`ts1`

The `timeseries` object that results when you apply an ideal filter to the original `timeseries` object.

Definitions

ideal filter

Filters are *ideal* in the sense that they are *not realizable*. An ideal filter is noncausal and the ends of the filter amplitude are perfectly flat in the frequency domain.

Examples

Apply Ideal Notch and Pass Filters

This example first applies an ideal notch filter to the data in `count.dat`. Then, it applies a pass filter to the data.

Load the count matrix into the workspace:

```
load count.dat
```

Create a timeseries object from column one of this matrix. Specify a time vector that ranges from 1 to 24 s in 1-s intervals.

```
count1 = timeseries(count(:,1),1:24);
```

Obtain the mean of the data:

```
countmean = mean(count1);
```

Enter the frequency interval, in hertz, for filtering the data:

```
interval = [0.08 0.2];
```

Invoke an ideal notch filter:

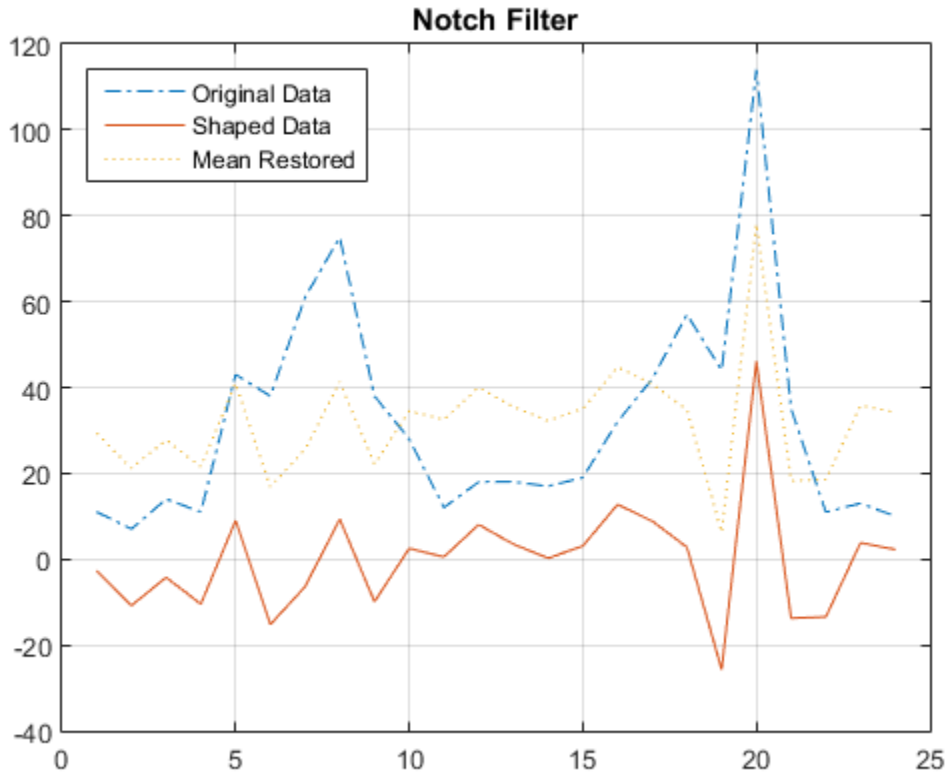
```
idealfilter_countn = idealfilter(count1,interval,'notch');
```

Compare the original data and the shaped data on a line plot:

```
plot(count1,'-.')
grid on
hold on
plot(idealfilter_countn,'-')
```

Restore the mean to the filtered data and show it on the line plot, adding a legend and a title:

```
countn_restored = idealfilter_countn + countmean;
plot(countn_restored,':')
title('Notch Filter')
legend('Original Data','Shaped Data','Mean Restored',...
       'Location','NorthWest')
```



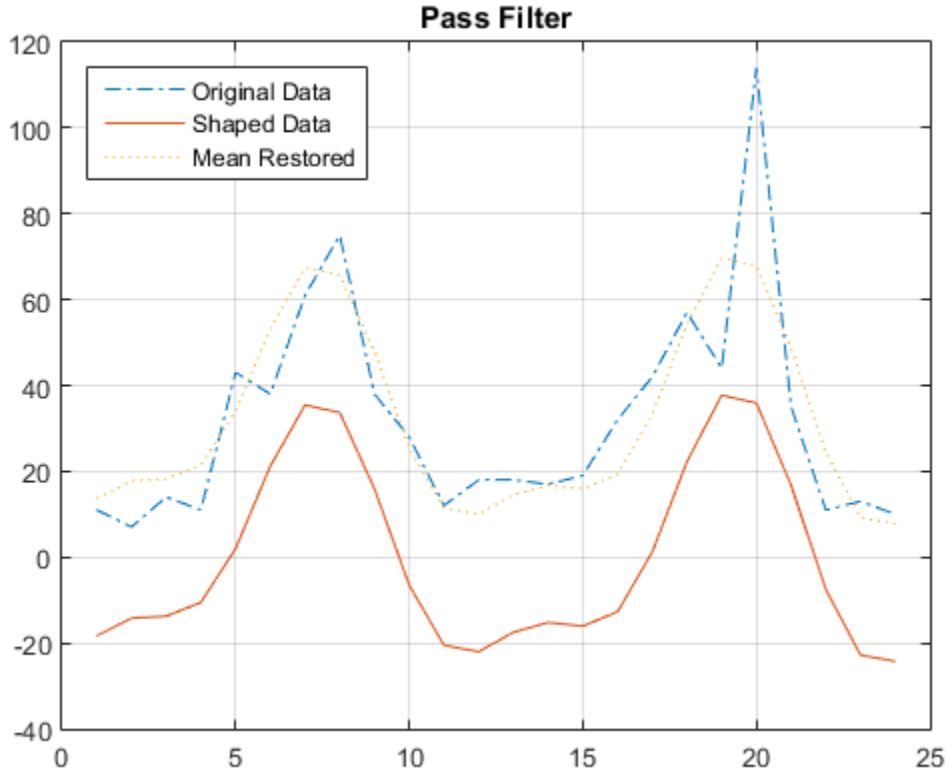
Close the Figure window:

```
close
```

Then, repeat the process using a `pass` rather than a `notch` filter:

```
figure

plot(count1,'-.')
grid on
hold on

idealfilter_countp = idealfilter(count1,interval,'pass');
plot(idealfilter_countp,'-')
```

```
countp_restored = idealfilter_countp + countmean;
plot(countp_restored,':')

title('Pass Filter')
legend('Original Data','Shaped Data','Mean Restored',...
       'Location','NorthWest')
```



See Also

`timeseries` | `filter`

Introduced before R2006a

iqr

Class: timeseries

Interquartile range of `timeseries` data

Syntax

```
ts_iqr = iqr(ts)
iqr(ts, Name, Value)
```

Description

`ts_iqr = iqr(ts)` returns the interquartile range of `ts.Data`.

`iqr(ts, Name, Value)` reruns the interquatrile range of `ts.Data` with the specified Name, Value pairs.

Input Arguments

`ts`

The `timeseries` object for which you want the interquartile range of `timeseries` data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

Output Arguments

ts_iqr

The interquartile range of `ts.Data`, as follows:

- When `ts.Data` is a vector, `ts_iqr` is the difference between the 75th and the 25th percentiles of the `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_iqr` is a row vector containing the interquartile range of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `iqr` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

Create a time series with a missing value, represented by `NaN`, and then calculate the interquartile range of `ts.Data` after removing the missing value from the calculation:

```
ts = timeseries([3.0 NaN 5 6.1 8], 1:5);
iqr(ts,'MissingData','remove')
```

MATLAB returns:

```
    3.0500
```

See Also

`mean` | `min` | `sum` | `timeseries` | `max` | `median` | `std` | `var`

Introduced before R2006a

max

Class: timeseries

Maximum value of `timeseries` data

Syntax

```
ts_max = max(ts)
ts_max = max(ts,Name,Value)
```

Description

`ts_max = max(ts)` returns the maximum value in the `timeseries` data.

`ts_max = max(ts,Name,Value)` returns the maximum value in the `timeseries` data with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`ts`

The `timeseries` object for which you want to determine the maximum data value.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

`'Quality'`

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

Output Arguments

`ts_max`

The maximum data value in the specified `timeseries` object, as follows:

- When `ts.Data` is a vector, `ts_max` is the maximum value of `ts.Data` values.
- When `ts.Data` is a matrix, `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_max` is a row vector containing the maximum value of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `max` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example illustrates how to find the maximum values in multivariate time-series data:

```
% Load a 24-by-3 data array:
load count.dat

% Create a timeseries object with 24 time values:
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')

% Find the maximum in each data column for this timeseries object:

max(count_ts)
```

MATLAB returns:

```
114   145   257
```

See Also

`mean` | `min` | `sum` | `timeseries` | `iqr` | `median` | `std` | `var`

Introduced before R2006a

mean

Class: timeseries

Mean value of `timeseries` data

Syntax

```
ts_mn = mean(ts)
ts_mn = mean(ts,Name,Value)
```

Description

`ts_mn = mean(ts)` returns the mean value of `ts.Data`.

`ts_mn = mean(ts,Name,Value)` returns the mean value of `ts.Data` with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`ts`

The `timeseries` object for which you want the mean value of data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

`'Quality'`

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

`'Weighting'`

A string specifying one of two possible values, `none` or `time`.
When you specify `time`, larger time values correspond to larger weights.

Output Arguments

`ts_mn`

The mean value of `ts.Data`, as follows:

- When `ts.Data` is a vector, `ts_mn` is the mean value of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time, then `ts_mn` is a row vector containing the mean value of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `mean` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

Find the mean values in multivariate time-series data:

```
% Load a 24-by-3 data array:

load count.dat

%  Create a timeseries object with 24 time values:

count_ts = timeseries(count,[1:24],'Name','CountPerSecond')

% Find the mean of each data column for this timeseries object:
```

```
mean(count_ts)
```

MATLAB returns:

```
32.0000   46.5417   65.5833
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note: If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`timeseries` | `timeseries.max` | `timeseries.min` | `timeseries.sum` | `timeseries.iqr` | `timeseries.median` | `timeseries.std` | `timeseries.var`

Introduced before R2006a

median

Class: timeseries

Median value of `timeseries` data

Syntax

```
ts_med = median(ts)
ts_med = method(ts,Name,Value)
```

Description

`ts_med = median(ts)` returns the median value of `ts.Data`.

`ts_med = method(ts,Name,Value)` returns the median value of `ts.Data` with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`ts`

The `timeseries` object for which you want the median data value.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, `none` or `time`.
When you specify `time`, larger time values correspond to larger weights.

Output Arguments

ts_med

The median value of `ts.Data`, as follows:

- When `ts.Data` is a vector, `ts_med` is the mean value of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_med` is a row vector containing the median value of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `median` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example finds the median values in multivariate time-series data. MATLAB finds the median independently for each data column in the `timeseries` object:

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,[1:24],'Name','CountPerSecond');

% Find the median of each data column for this timeseries object:
```

```
median(count_ts)
```

MATLAB returns:

```
23.5000   36.0000   39.0000
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note: If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `mean` | `std` | `var` | `max` | `min` | `sum` | `timeseries`

Introduced before R2006a

min

Class: timeseries

Minimum value of `timeseries` data

Syntax

```
ts_min = min(ts)
ts_min = method(ts,Name,Value)
```

Description

`ts_min = min(ts)` returns the minimum value in the `timeseries` data.

`ts_min = method(ts,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Input Arguments

`ts`

The `timeseries` object for which you want the minimum data value.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

Output Arguments

ts_min

The minimum value of `ts.Data`, as follows:

- When `ts.Data` is a vector, `ts_min` is the minimum value of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_min` is a row vector containing the minimum value of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `min` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example finds the minimum values in multivariate time-series data. MATLAB finds the minimum independently for each data column in the `timeseries` object.

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,[1:24],'Name','CountPerSecond');

% Find the minimum in each data column for this timeseries object:

min(count_ts)
```

MATLAB returns:

```
7     9     7
```

See Also

`max` | `median` | `sum` | `timeseries` | `iqr` | `mean` | `std` | `var`

Introduced before R2006a

plot

Class: timeseries

Plot time series

Syntax

```
plot(ts)
plot(tsc.tsname)
plot(ts,linespec)
plot(tsc.tsname,linespec)
plot(ts,Name, Value)
plot(tsc.tsname,Name, Value)
```

Description

`plot(ts)` plots the `timeseries` data ts against time and interpolates values between samples by using either zero-order-hold (`'zoh'`) or linear interpolation (the default). The plot displays in the current axes. MATLAB creates a title and axes, if none exists.

`plot(tsc.tsname)` plots the `timeseries` object, tsname that is part of the `tscollection`, tsc.

`plot(ts,linespec)` plots the `timeseries` data using a line graph and applies the specified `linespec` to lines, markers, or both.

`plot(tsc.tsname,linespec)` plots the `timeseries` object that is part of a `timeseries` collection as a line graph and applies the specified `linespec` to lines, markers, or both.

`plot(ts,Name, Value)` plots a line graph of the time series data using the values specified for Chart Line Properties.

`plot(tsc.tsname,Name, Value)` plots a line graph of the `timeseries` object that is part of the specified `timeseries` collection using the values specified for Chart Line Properties.

Tips

- The `timeseries/plot` method generates titles and axis labels automatically. These labels are:
  - Plot Title — `'Time Series Plot: <name>'`

    where `<name>` is the string assigned to `ts.Name`, or by default, `'unnamed'`
  - X-Axis Label — `'Time (<units>)'`

    where `<units>` is the value of the `ts.TimeInfo.Units` field, which defaults to `'seconds'`
  - Y-Axis Label — `'<name>'`

    where `<name>` is the string assigned to `ts.Name`, or by default, `'unnamed'`
- You can place new time series data on a time series plot (by setting `hold on`, for example, and issuing another `timeseries/plot` command). When you add data to a plot, the title and axis labels become blank strings to avoid labeling confusion. You can add your own labels after plotting using the `title`, `xlabel`, and `ylabel` commands.
- Time series events, when defined, are marked in the plot with a circular marker with red fill. You can also specify markers for all data points using a `linespec` or name/value syntax in addition to any event markers your data defines. The event markers plot on top of the markers you define.
- The value assigned to `ts.DataInfo.Interpolation.Name` controls the type of interpolation the `plot` method uses when plotting and resampling time series data. Invoke the `timeseries` method `setinterpmethod` to change default linear interpolation to zero-order hold interpolation (staircase). This method creates a new `timeseries` object, with which you can overwrite the original one if you want. For example, to cause time series `ts` to use zero-order hold interpolation, type the following:

  ```
  ts = ts.setinterpmethod('zoh');
  ```

Input Arguments

ts

A `timeseries` object.

tsc

A `tscollection`.

tsname

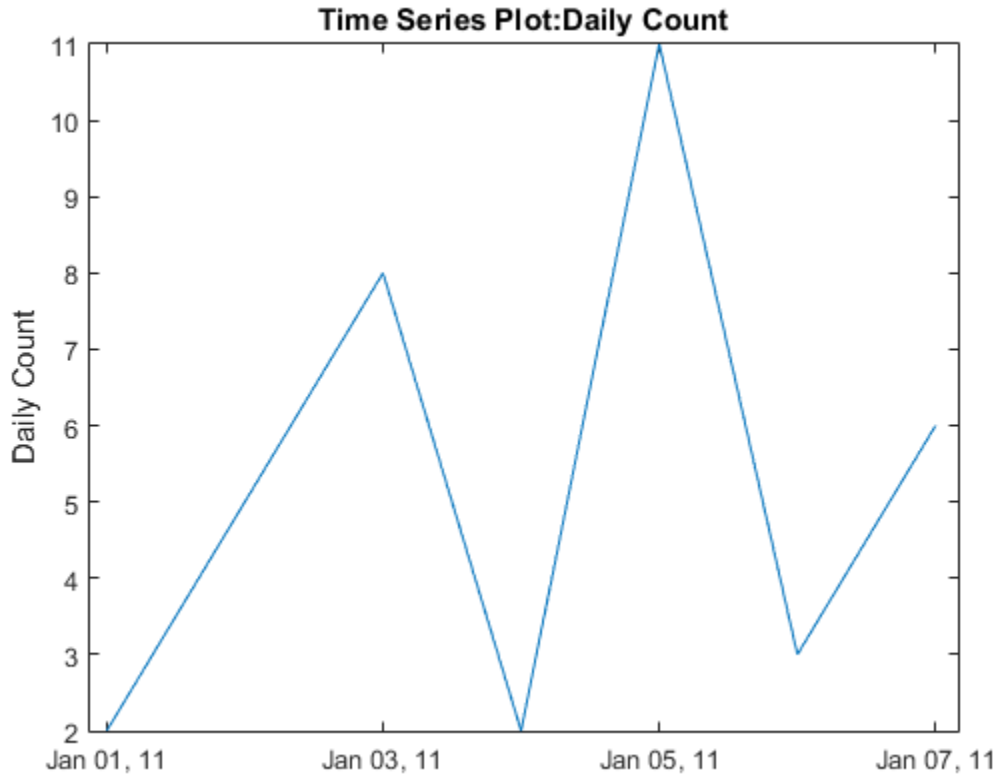
The name of a `timeseries` object within the `tscollection` .

Examples

Plot Time Series Object with Specified Start Date

Create a time series object, set the start date, and then plot the time vector relative to the start date.

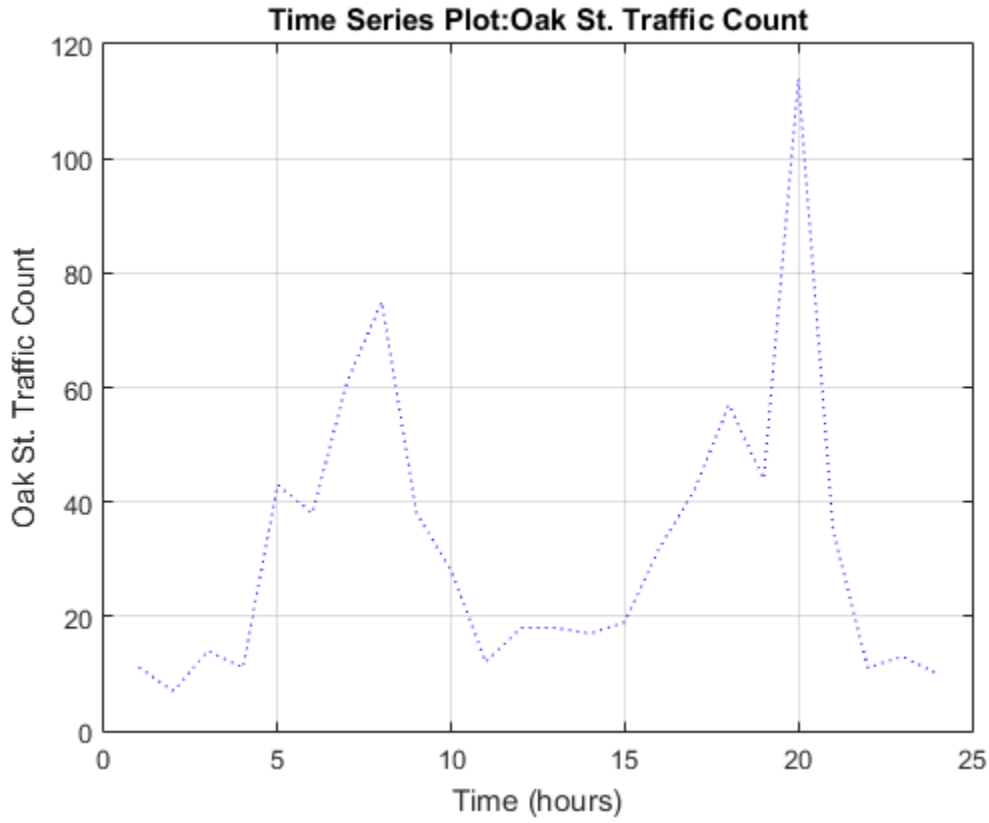
```
x = [2 5 8 2 11 3 6];
ts1 = timeseries(x,1:7);

ts1.Name = 'Daily Count';
ts1.TimeInfo.Units = 'days';
ts1.TimeInfo.StartDate = '01-Jan-2011';     % Set start date.
ts1.TimeInfo.Format = 'mmm dd, yy';       % Set format for display on x-axis.

ts1.Time = ts1.Time - ts1.Time(1);        % Express time relative to the start date.

plot(ts1)
```



Plot Two Time Series Objects on the Same Axes

Create two time series objects from traffic count data, and then plot them in sequence on the same axes. Add an event to one series, which is automatically displayed with a red marker.

```
load count.dat;
count1 = timeseries(count(:,1),1:24);
count1.Name = 'Oak St. Traffic Count';
count1.TimeInfo.Units = 'hours';
plot(count1,':b')
grid on
```



Obtain time of maximum value and add it as an event:

```
[~,index] = max(count1.Data);
max_event = tsdata.event('peak',count1.Time(index));
max_event.Units = 'hours';
```

Add the event to the time series:

```
count1 = addevent(count1,max_event);
```

Replace plot with new one showing the event:

```
plot(count1,'.-b')
grid on
```

Make a new time series object from column 2 of the same data source:

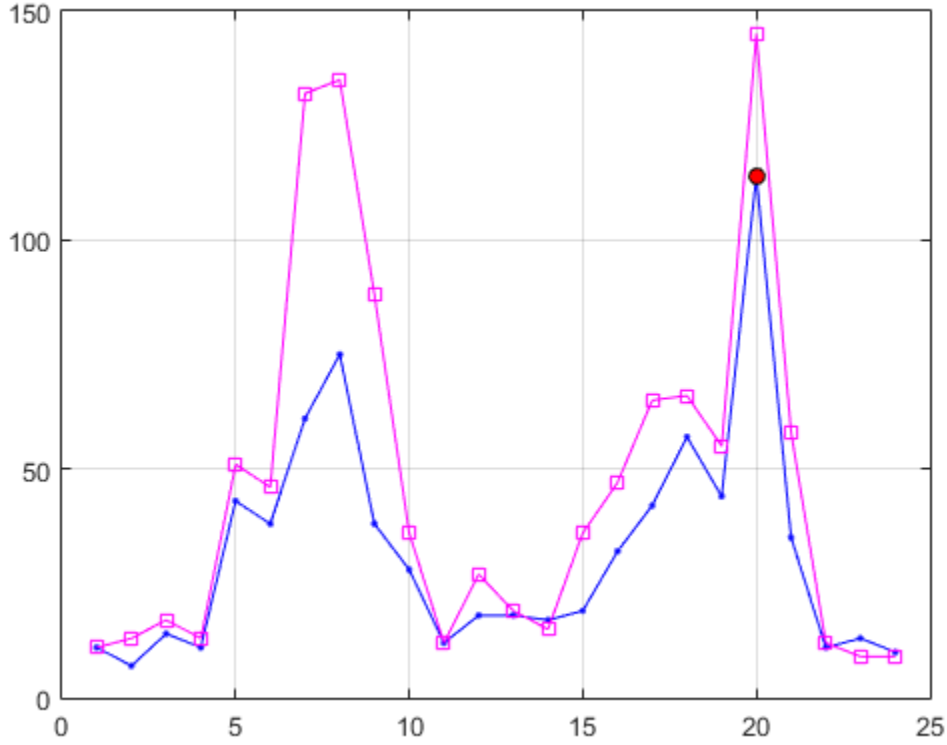
```
count2 = timeseries(count(:,2),1:24);
count2.Name = 'Maple St. Traffic Count';
count2.TimeInfo.Units = 'Hours';
```

Turn hold on to add the new data to the plot:

```
hold on
```

The plot method does not add labels to a held plot. Use property/value pairs to customize markers:

```
plot(count2,'s-m','MarkerSize',6),
```

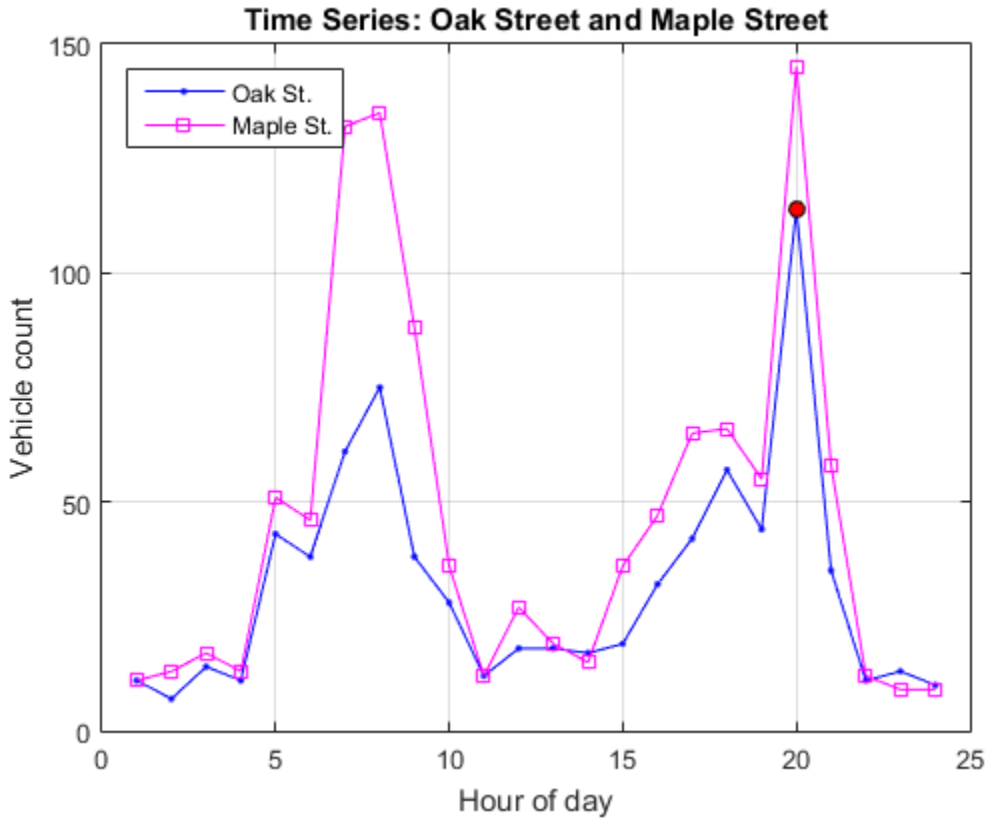


Labels are erased, so generate them manually:

```
title('Time Series: Oak Street and Maple Street')
xlabel('Hour of day')
ylabel('Vehicle count')
```

Add a legend in the upper left:

```
legend('Oak St.','Maple St.','Location','northwest')
```



See Also

`timeseries` | `setinterpmethod` | `tscollection` | `tsdata.event` | `plot`

Introduced before R2006a

resample

Class: timeseries

Select or interpolate `timeseries` data using new time vector

Syntax

```
ts1 = resample(ts, time)
ts1 = resample(ts, time, interp_method)
ts1 = resample(ts, time, interp_method, code)
```

Description

`ts1 = resample(ts, time)` resamples the `timeseries` object, ts, using the new time vector. The `resample` method uses the default interpolation method, which you can view by using the `getinterpmethod(ts)` syntax.

`ts1 = resample(ts, time, interp_method)` resamples the `timeseries` object ts using the specified interpolation method, interp_method.

`ts1 = resample(ts, time, interp_method, code)` resamples the `timeseries` object ts using the interpolation method given by the string interp_method. MATLAB applies the code to all samples.

Input Arguments

`ts`

The `timeseries` object that you want to resample.

`time`

The time vector you want to use to resample the `timeseries` object.

When `ts` uses date strings and `time` is numeric, then `time` is treated as specified relative to the `ts.TimeInfo.StartDate` property and in the same units that `ts` uses.

`interp_method`

A string specifying the interpolation method. Valid interpolation methods are `linear` and zero-order hold, `zoh` .

Default: `linear`

`code`

An integer value that specifies the user-defined `Quality` code for resampling. MATLAB applies this `Quality` code to all samples.

Output Arguments

`ts1`

The `timeseries` object that results when you interpolate the original `timeseries` object with a new time vector.

Examples

This example shows how to resample a `timeseries` object.

Create a `timeseries` object.

```
ts1 = timeseries([1.1; 2.9; 3.7; 4.0; 3.0],1:5,'Name','speed');
```

View the time, data, and interpolation method.

```
ts1.time
ts1.data
ts1.getinterpmethod
```

Resample `ts1` using its default interpolation method.

```
res_ts=resample(ts1,[1 1.5 3.5 4.5 4.9]);
```

View the time, data, and interpolation method for the resampled object.

```
res_ts.time
res_ts.data
```

```
res_ts.getinterpmethod
```

See Also

timeseries | getinterpmethod | setinterpmethod | synchronize

Introduced before R2006a

set

Class: timeseries

Set properties of `timeseries` object

Syntax

```
set(ts, PropertyName, Value)
set(ts, PropertyName)
set(ts)
```

Description

`set(ts, PropertyName, Value)` sets the named property, `Name`, of the `timeseries` object, `ts`, to the value, `Value`. The following syntax is equivalent:

```
ts.Property = Value
```

`set(ts, PropertyName)` displays the value of the named property for the `timeseries` object, `ts`.

`set(ts)` displays all properties and values of the `timeseries` object `ts`.

Input Arguments

`ts`

A `timeseries` object.

`PropertyName`

A string specifying the name of a `timeseries` property. For a list of `timeseries` properties, see `timeseries`.

`Value`

The value to which you want to set the named property.

Examples

Create a `timeseries`, set its name to `mytimeseries`, and then view the `timeseries` properties:

```
ts1 = timeseries(rand(5,1),[1 2 3 4 5]);
set(ts1, 'Name', 'mytimeseries')
set(ts1)
```

See Also

`timeseries` | `get`

Introduced before R2006a

setabstime

Class: timeseries

Set times of `timeseries` object as date strings

Syntax

```
ts1=setabstime(ts, times)
ts1=setabstime(ts, times, format)
```

Description

`ts1=setabstime(ts, times)` sets the times in ts to the date strings specified in times.

`ts1=setabstime(ts, times, format)` explicitly specifies the date-string format, format, used in times.

Input Arguments

`ts`

The `timeseries` object for which you want to set times as date strings.

`times`

A cell array of strings or a `char` array containing valid date or time values in the same date format.

`format`

The date-string format used for the time values.

Output Arguments

`ts1`

The `timeseries` object that results from setting times as date strings on the original `timeseries` object.

Examples

Create a `timeseries` object, and then set the absolute time vector:

```
ts = timeseries(rand(3,1))
ts1 = setabstime(ts,{'12-DEC-2005 12:34:56',...
     '12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})

% View each timeseries object in the Variable Edtior to see the
% differences in the time vectors for each.
```

See Also

`datestr` | `getabstime` | `timeseries`

Introduced before R2006a

setinterpmethod

Class: timeseries

Set default interpolation method for `timeseries` object

Syntax

```
ts = setinterpmethod(ts, method)
ts = setinterpmethod(ts, fhandle)
ts = setinterpmethod(ts, interpobj)
```

Description

`ts = setinterpmethod(ts, method)` sets the default interpolation method, method, for `timeseries` object, ts, and outputs it to ts1.

`ts = setinterpmethod(ts, fhandle)` sets the default interpolation method for `timeseries` object ts, where fhandle is a function handle to the interpolation method.

`ts = setinterpmethod(ts, interpobj)` sets the default interpolation method for `timeseries` object ts, where interpobj is a `tsdata.interpolation` object that directly replaces the interpolation object stored in ts.

Tips

- This method is case sensitive.

Input Arguments

`ts`

The `timeseries` object for which you want to set the default interpolation method.

`method`

A string specifying the interpolation method. Valid values are `linear` and zero-order hold, `zoh`.

Default: `linear`

`fhandle`

A function handle to the interpolation method. The order of input arguments defining the function handle must be `new_time`, `time`, and `data`. The single output argument must be the interpolated data only.

`interpobj`

A `tsdata.interpolation` object that directly replaces the interpolation object stored in ts.

Output Arguments

`ts1`

The `timeseries` object that results when you set the interpolation method for the original `timeseries` object.

Examples

Set the default interpolation method for `timeseries` object `ts` to zero order hold:

```
ts = timeseries(rand(100,1),1:100);
ts = setinterpmethod(ts,'zoh');
plot(ts);
```

Set the default interpolation method for `timeseries` object `ts`, where fhandle is a function handle to the interpolation method defined by function handle `myFuncHandle`:

```
ts = timeseries(rand(100,1),1:100);
myFuncHandle = @(new_time, time, data)...
               interp1(time, data, new_time,...
                       'linear','extrap');
```

```
ts = setinterpmethod(ts, myFuncHandle);
ts = resample(ts, [-5:0.1:10]);
plot(ts);
```

Set the default interpolation method for `timeseries` object `ts` to a `tsdata.interpolation` object:

```
ts = timeseries(rand(100,1),1:100);
myFuncHandle = @(new_time, time, data)...
               interp1(time, data, new_time,...
                       'linear','extrap');
myInterpObj = tsdata.interpolation(myFuncHandle);
ts = setinterpmethod(ts,myInterpObj);
plot(ts);
```

See Also

`timeseries` | `getinterpmethod`

Introduced before R2006a

setuniformtime

Class: timeseries

Modify uniform time vector of `timeseries` object

Syntax

```
ts2 = setuniformtime(ts1,'StartTime',StartTime)
ts2 = setuniformtime(ts1,'Interval',Interval)
ts2 = setuniformtime(ts1,'EndTime',EndTime)
ts2 = setuniformtime(ts1,'StartTime',StartTime,'Interval',Interval)
ts2 = setuniformtime(ts1,'StartTime',StartTime,'EndTime',EndTime)
ts2 = setuniformtime(ts1,'Interval',Interval,'EndTime',EndTime)
```

Description

`ts2 = setuniformtime(ts1,'StartTime',StartTime)` returns the time series with a modified uniform time vector, determined from the StartTime and Interval `1`. `EndTime = StartTime + (length(ts1) - 1)`. The unit of time is unchanged.

`ts2 = setuniformtime(ts1,'Interval',Interval)` sets the `StartTime` to `0`, and uses `EndTime = (length(ts1) - 1)*Interval`.

`ts2 = setuniformtime(ts1,'EndTime',EndTime)` sets the StartTime to `0`, and uses `Interval = EndTime/(length(ts1) -1)`.

`ts2 = setuniformtime(ts1,'StartTime',StartTime,'Interval',Interval)` uses `EndTime = StartTime + (length(ts1) - 1) * Interval`.

`ts2 = setuniformtime(ts1,'StartTime',StartTime,'EndTime',EndTime)` uses `Interval = (EndTime - StartTime)/(length(ts1) - 1)`.

`ts2 = setuniformtime(ts1,'Interval',Interval,'EndTime',EndTime)` uses `StartTime = EndTime - (length(ts1) - 1) * Interval`.

Input Arguments

`ts1`

`timeseries` object to which you want to assign a uniform time vector.

`StartTime`

Start time of uniform time vector, specified as a numeric value.

`Interval`

Time interval of uniform time vector, specified as a numeric scalar value.

`EndTime`

End time of uniform time vector specified as a numeric scalar value.

Output Arguments

`ts2`

Time series with uniform time vector, returned as a `timeseries` object.

Examples

Specify New Start Time

Modify the uniform time vector of time series data by specifying a new start time.

1. Load the sample data.

   ```
   load count.dat;
   ```

2. Create a `timeseries` object.

   ```
   count_ts = timeseries(count,1:length(count),'Name','CountPerSecond');
   ```

3. Modify uniform time vector of `count_ts`.

   ```
   count_ts = setuniformtime(count_ts,'StartTime',10);
   ```

The start time of the time vector is `10` seconds. `setuniformtime` uses a default time interval of `1` and computes the end time using: `EndTime = StartTime + (length(count_ts) - 1)* Interval`.

Specify New Start and End Times

Modify the uniform time vector of time series data by specifying a new start time and end time.

1. Load sample data.

   ```
   load count.dat;
   ```

2. Create `timeseries` object.

   ```
   count_ts = timeseries(count,1:length(count),'Name','CountPerSecond');
   ```

3. Assign a uniform time vector to `count_ts`.

   ```
   count_ts2 = setuniformtime(count_ts,'StartTime',10,'EndTime',20);
   ```

   The start time of the time vector is now 10 seconds, and the end time is now 20 seconds. MATLAB computes the time interval using: `Interval = (EndTime - StartTime)/(length(count_ts) - 1)`

See Also

`timeseries`

synchronize

Class: timeseries

Synchronize and resample two `timeseries` objects using common time vector

Syntax

```
[ts1 ts2] = synchronize(ts1,ts2,synchronizemethod)
[ts1 ts2] = synchronize(___,Name,Value)
```

Description

`[ts1 ts2] = synchronize(ts1,ts2,synchronizemethod)` creates two new `timeseries` objects by synchronizing ts1 and ts2 using a common time vector and the specified method.

`[ts1 ts2] = synchronize(___,Name,Value)` creates the two new `timeseries` objects with additional options specified by one or more Name,Value pair arguments for the previous syntax.

Input Arguments

`ts1`

One of the `timeseries` objects that you want to synchronize and resample.

`ts2`

The other `timeseries` object that you want to synchronize and resample.

`synchronizemethod`

A string that defines the method for synchronizing the `timeseries` object. It can be any one of the following:

- `Union` — Resample `timeseries` objects using a time vector that is a union of the time vectors of `ts1` and `ts2` on the time range where the two time vectors overlap.

- `Intersection` — Resample `timeseries` objects on a time vector that is the intersection of the time vectors of `ts1` and `ts2`.
- `Uniform` — Requires an additional argument as follows:

  ```
  [ts1 ts2] = synchronize(ts1,ts2,'Uniform','Interval',value)
  ```

  This method resamples time series on a uniform time vector, where `value` specifies the time interval between two consecutive samples. The uniform time vector is the overlap of the time vectors of `ts1` and `ts2`. The interval units are the smaller units of `ts1` and `ts2`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'InterpMethod'`

Forces the specified interpolation method (over the default method) for this synchronize operation. Can be either a string, `linear` or `zoh`, or a `tsdata.interpolation` object that contains a user-defined interpolation method.

Default: `linear`

`'QualityCode'`

Integer (between -128 and 127) used as the quality code for both time series after the synchronization.

`'KeepOriginalTimes'`

Logical value (`true` or `false`) indicating whether the new time series should keep the original time values.

`'tolerance'`

Real number used as the tolerance for differentiating two time values when comparing the `ts1` and `ts2` time vectors. The default tolerance is `1e-10`. For example, when the sixth time value in `ts1` is `5+(1e-12)` and the sixth time value in `ts2` is `5-(1e-13)`,

both values are treated as 5 by default. To differentiate those two times, you can set `'tolerance'` to a smaller value such as `1e-15`, for example.

Output Arguments

`ts1`

One of the `timeseries` objects that you synchronized and resampled.

`ts2`

The other `timeseries` object that you synchronized and resampled.

Examples

This example illustrates how the `KeepOriginalTime` property affects synchronization.

```
% Create two timeseries, such that ts1.timeinfo.StartDate
% is one day after ts2.timeinfo.StartDate:

ts1 = timeseries([1 2],[datestr(now); datestr(now+1)]);
ts2 = timeseries([1 2],[datestr(now-1); datestr(now)]);

% If you use this code, then ts1.timeinfo.StartDate
% is changed to match ts2.TimeInfo.StartDate
% and ts1.Time changes to 1:

[ts1 ts2] = synchronize(ts1,ts2,'union');

% But if you use this code, then ts1.timeinfo.StartDate
% is unchanged and ts1.Time is still 0:

[ts1 ts2] = synchronize(ts1,ts2,'union','KeepOriginalTimes',true);
```

See Also

`set` | `timeseries`

Introduced before R2006a

transpose

Class: timeseries

Transpose `timeseries` object

Syntax

```
ts1 = transpose(ts)
```

Description

`ts1 = transpose(ts)` returns a new `timeseries` object, ts1, with `IsTimeFirst` value set to the opposite of what it is for ts. For example, if `ts` has the first data dimension aligned with the time vector, `ts1` has the last data dimension aligned with the time vector.

Tips

- The `transpose` function that is overloaded for `timeseries` objects does not transpose the data. Instead, this function changes whether the first or the last dimension of the data aligns with the time vector. To transpose the data, transpose the `Data` property of the time series. For example, you can use the syntax `transpose(ts.Data)` or `(ts.Data).'`. The value of the `Data` property must be a 2-D array.
- Consider a time series with 10 samples with the property `IsTimeFirst = True`. When you transpose this time series, the data size changes from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

  The following table summarizes how the size for timeseries data (up to three dimensions) display before and after transposing.

  Data Size Before and After Transposing

Size of Original Data	Size of Transposed Data
N-by-1	1-by-1-by-N

Size of Original Data	Size of Transposed Data
N-by-M	M-by-1-by-N
N-by-M-by-L	M-by-L-by-N

Input Arguments

`ts`

The `timeseries` object that you want to transpose.

Output Arguments

`ts1`

The `timeseries` object that is the result of transposing the original `timeseries` object.

Examples

Suppose that a `timeseries` object, `ts`, has `ts.Data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is `true`, which means that the first dimension of the data aligns with the time vector. `transpose(ts)` modifies the `timeseries` object, such that the last dimension of the data now aligns with the time vector. This permutes the data, such that the size of `ts.Data` becomes 3-by-2-by-10.

See Also

`timeseries` | `transpose`

Introduced before R2006a

std

Class: timeseries

Standard deviation of `timeseries` data

Syntax

```
ts_std = std(ts)
ts_std = std(ts,Name,Value)
```

Description

`ts_std = std(ts)` returns the standard deviation of the `timeseries` data.

`ts_std = std(ts,Name,Value)` specifies additional options specified with one or more Name,Value pair arguments.

Input Arguments

`ts`

The `timeseries` object for which you want the standard deviation of the data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, `none` or `time`.
When you specify `time`, larger time values correspond to larger weights.

Output Arguments

ts_std

The standard deviation of `ts.Data` values as follows:

- When `ts.Data` is a vector, `ts_std` is the standard deviation of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_std` is the standard deviation of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `std` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example finds the standard deviation for a `timeseries` object. MATLAB calculates the standard deviation for each data column in the `timeseries` object.

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,1:24,'Name','CountPerSecond');

% Calculate the standard deviation of each data column for this
% timeseries object:
```

```
std(count_ts)
```

MATLAB returns:

```
25.3703   41.4057   68.0281
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note: If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `mean` | `min` | `var` | `max` | `median` | `sum` | `timeseries`

Introduced before R2006a

sum

Class: timeseries

Sum of `timeseries` data

Syntax

```
ts_sm = sum(ts)
ts_sm = sum(ts,Name,Value)
```

Description

`ts_sm = sum(ts)` returns the sum of the `timeseries` data.

`ts_sm = sum(ts,Name,Value)` specifies additional options with one or more Name,Value pair arguments.

Input Arguments

`ts`

The `timeseries` object for which you want the sum of the data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

`'Quality'`

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

`'Weighting'`

A string specifying one of two possible values, `none` or `time`.
When you specify `time`, larger time values correspond to larger weights.

Output Arguments

`ts_sm`

The sum of the `timeseries` data, as follows:

- When `ts.Data` is a vector, ts_sm is the sum of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then ts_sm is a row vector containing the sum of each column of `ts.Data`.

When `ts.Data` is a N-dimensional array, `sum` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

Calculate the sum of each data column for a `timeseries` object:

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,1:24,'Name','CountPerSecond');

% Calculate the sum of each data column for this timeseries object:
```

```
sum(count_ts)
```

MATLAB returns:

```
768         1117         1574
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note: If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `mean` | `min` | `var` | `max` | `median` | `std` | `timeseries`

Introduced before R2006a

var

Class: timeseries

Variance of `timeseries` data

Syntax

```
ts_var = var(ts)
ts_var = var(ts,Name,Value)
```

Description

`ts_var = var(ts)` returns the variance of `ts.data`.

`ts_var = var(ts,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Input Arguments

`ts`

The `timeseries` object for which you want the variance of the data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MissingData'`

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, `none` or `time`.
When you specify `time`, larger time values correspond to larger weights.

Output Arguments

ts_var

The variance of `ts.data`, as follows:

- When `ts.Data` is a vector, then `ts_var` is the variance of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_var` is a row vector containing the variance of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `var` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example calculates the variance values of a multivariate `timeseries` object. MATLAB calculates the variance independently for each data column in the `timeseries` object.

Load a 24-by-3 data array. Then create a timeseries object with 24 time values.

```
load count.dat
count_ts = timeseries(count,[1:24],'Name','CountPerSecond');
```

Calculate the variance of each data column.

```
var(count_ts)
```

MATLAB returns:

```
1.0e+03 *
  0.6437    1.7144    4.6278
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note: If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `mean` | `min` | `sum` | `max` | `median` | `std` | `timeseries`

Introduced before R2006a

triangulation class

Triangulation in 2-D or 3-D

Description

Use `triangulation` to create an in-memory representation of any 2-D or 3-D triangulation data that is in matrix format, such as the matrix output from the `delaunay` function or other software tools. When your data is represented using `triangulation`, you can perform topological and geometric queries, which you can use to develop geometric algorithms. For example, you can find the triangles or tetrahedra attached to a vertex, those that share an edge, their circumcenters, and other features.

Construction

`TR = triangulation(T,P)` creates a 2-D or 3-D triangulation representation using the triangulation connectivity list, `T`, and the points in matrix `P`.

`TR = triangulation(T,x,y)` creates a 2-D triangulation representation with the point coordinates specified as column vectors, `x` and `y`.

`TR = triangulation(T,x,y,z)` creates a 3-D triangulation representation with the point coordinates specified as column vectors, `x`, `y`, and `z`.

Input Arguments

`T`

Triangulation connectivity list, specified as an `m`-by-`n` matrix, where `m` is the number of triangles or tetrahedra, and `n` is the number of vertices per triangle or tetrahedron. Each element in `T` is a "Vertex ID" on page 1-6833. Each row of `T` contains the vertex IDs that define a triangle or tetrahedron.

`P`

Points, specified as a matrix whose columns are the `x`, `y`, (and possibly `z`) coordinates of the triangulation points. The row numbers of `P` are the vertex IDs in the triangulation.

`x`

x-coordinates vector, specified as a column vector containing the *x*-coordinates of the triangulation points.

`y`

y-coordinates vector, specified as a column vector containing the *y*-coordinates of the triangulation points.

`z`

z-coordinates vector, specified as a column vector containing the *z*-coordinates of the triangulation points.

Properties

`Points`

Points in the triangulation, represented as a matrix containing the following information:

- Each row in `TR.Points` contains the coordinates of a vertex.
- Each row number of `TR.Points` is a vertex ID.

`ConnectivityList`

Triangulation connectivity list, represented as a matrix. This matrix contains the following information:

- Each element in `TR.ConnectivityList` is a vertex ID.
- Each row represents a triangle or tetrahedron in the triangulation.
- Each row number of `TR.ConnectivityList` is a "Triangle or Tetrahedron ID" on page 1-6833.

Methods

barycentricToCartesian	Converts point coordinates from barycentric to Cartesian

cartesianToBarycentric	Converts point coordinates from Cartesian to barycentric
circumcenter	Circumcenter of triangle or tetrahedron
edgeAttachments	Triangles or tetrahedra attached to specified edge
edges	Triangulation edges
faceNormal	Triangulation face normal
featureEdges	Triangulation sharp edges
freeBoundary	Triangulation facets referenced by only one triangle or tetrahedron
incenter	Incenter of triangle or tetrahedron
isConnected	Test if two vertices are connected by edge
nearestNeighbor	Vertex closest to specified location
neighbors	Neighbors to specified triangle or tetrahedron
pointLocation	Triangle or tetrahedron containing specified point
size	Size of triangulation connectivity list
vertexAttachments	Triangles or tetrahedra attached to specified vertex

vertexNormal	Triangulation vertex normal

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

2-D Triangulation

Define the points in the triangulation.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];
```

Define the triangles. This is the triangulation connectivity list.

```
T = [5  3  1;
```

```
     3  2  1;
     3  4  2;
     4  6  2];
```

Create the triangulation representation.

```
TR = triangulation(T,P)
```

```
TR =

  triangulation with properties:

             Points: [6x2 double]
    ConnectivityList: [4x3 double]
```

Examine the coordinates of the vertices of the first triangle.

```
TR.Points(TR.ConnectivityList(1,:),:)
```

```
ans =

    1.0000    6.5000
    2.5000    5.0000
    2.5000    8.0000
```

See Also

`delaunayTriangulation`

barycentricToCartesian

Class: triangulation

Converts point coordinates from barycentric to Cartesian

Syntax

```
PC = barycentricToCartesian(TR,ti,B)
```

Description

`PC = barycentricToCartesian(TR,ti,B)` returns the Cartesian coordinates of the points in `B`. Each row, `B(j,:)`, contains the barycentric coordinates of a point with respect to the triangle or tetrahedron, `ti(j)`. The point, `PC(j,:)`, is the `j`th point represented in Cartesian coordinates.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`ti`

Triangle or tetrahedron IDs, specified as a column vector.

`B`

Barycentric coordinates, specified as a matrix. Each row, `B(j,:)`, contains the barycentric coordinates of a point with respect to the triangle or tetrahedron, `ti(j)`.

Output Arguments

`PC`

Cartesian coordinates, returned as a matrix. The point, `PC(j,:)`, is the `j`th point represented in Cartesian coordinates.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Barycentric Coordinates Converted to Cartesian Coordinates

Create a triangulation from a set of points, `P`, and triangulation connectivity list, `T`.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];

TR = triangulation(T,P);
```

Specify the first triangle.

```
ti = 1;
```

Specify the barycentric coordinates of the second point in the triangle.

```
B = [0 1 0];
```

Convert the point to Cartesian coordinates.

```
PC = barycentricToCartesian(TR,ti,B)
```

```
PC =

    2.5000    5.0000
```

Mapped Incenters of Deformed Triangulation

Create a Delaunay triangulation from a set of points.

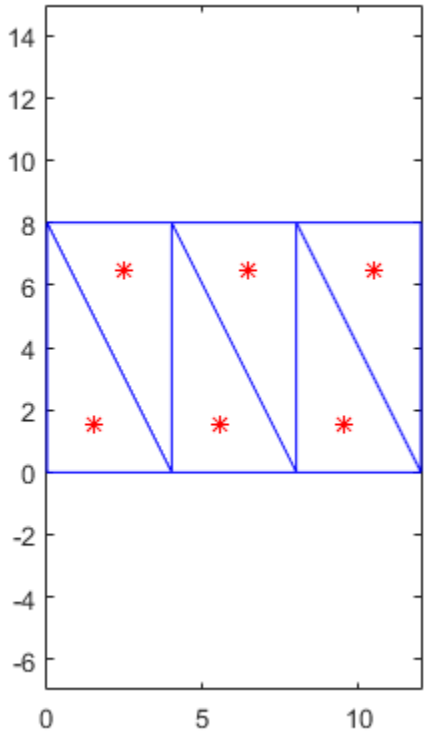
```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
DT = delaunayTriangulation(x,y);
```

Calculate the Cartesian coordinates of the incenters.

```
cc = incenter(DT);
```

Plot the original triangulation and reference points.

```
figure
subplot(1,2,1);
triplot(DT);
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;
axis equal;
```



Create a new triangulation which is a deformed version of `DT` .

```
ti = DT.ConnectivityList;
y = [0 0 0 0 16 16 16 16]';
TR = triangulation(ti,x,y);
```

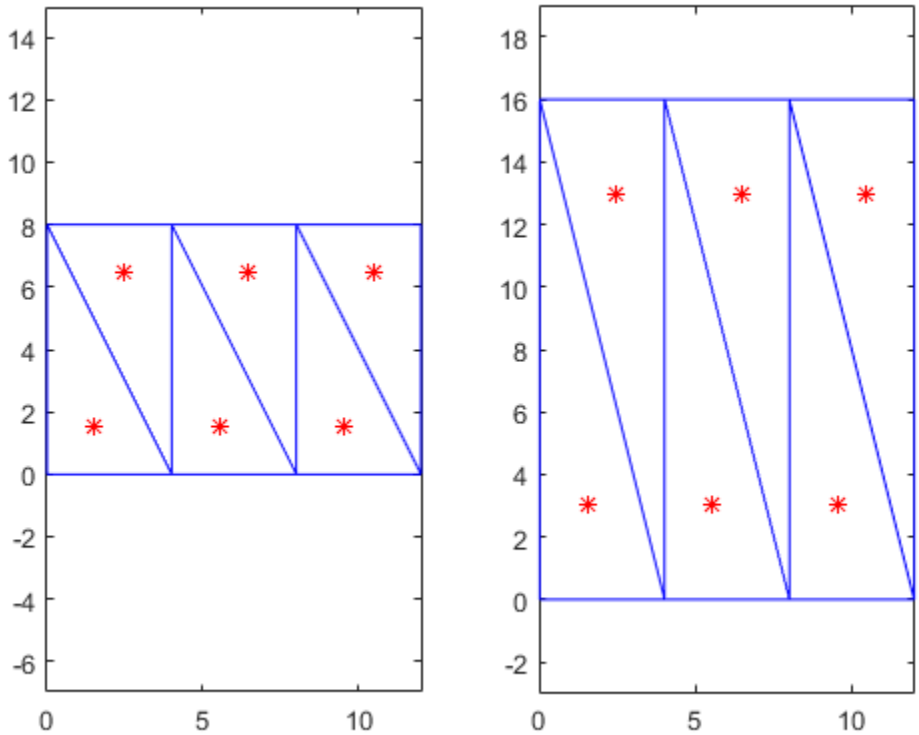
Compute the barycentric coordinates for the incenters of `DT`, and use them to compute the Cartesian coordinates of the analogous points in `TR`.

```
b = cartesianToBarycentric(DT,[1:length(ti)]',cc);
xc = barycentricToCartesian(TR,[1:length(ti)]',b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
```

```
triplot(TR);
hold on;
plot(xc(:,1),xc(:,2),'*r');
hold off;
axis equal;
```



See Also

`cartesianToBarycentric` | `delaunayTriangulation`

cartesianToBarycentric

Class: triangulation

Converts point coordinates from Cartesian to barycentric

Syntax

```
B = cartesianToBarycentric(TR,ti,PC)
```

Description

`B = cartesianToBarycentric(TR,ti,PC)` returns the barycentric coordinates of the points in `PC`. Each row, `PC(j,:)`, contains the Cartesian coordinates of a point you want to convert. `B(j,:)` are the barycentric coordinates of the point, `PC(j,:)`, with respect to triangle or tetrahedron, ti(j).

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`ti`

Triangle or tetrahedron IDs, specified as a column vector.

`PC`

Cartesian coordinates, specified as a matrix. Each row, `PC(j,:)`, contains the Cartesian coordinates of a point with respect to triangle or tetrahedron, `ti(j)`.

Output Arguments

`B`

Barycentric coordinates, returned as a matrix. `B(j,:)` are the barycentric coordinates of `PC(j,:)` with respect to ti(j).

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Cartesian Coordinates Converted to Barycentric Coordinates

Create a triangulation from a set of points, `P`, and triangulation connectivity list, `T`.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];

TR = triangulation(T,P);
```

Specify the first triangle.

```
ti = 1;
```

Get the coordinates of the third vertex in the first triangle.

```
PC = TR.Points(TR.ConnectivityList(1,3),:)
```

```
PC =

    2.5000    8.0000
```

Convert the point to barycentric coordinates.

```
B = cartesianToBarycentric(TR,ti,PC)
```

```
B =

     0     0     1
```

Mapped Incenters of Deformed Triangulation

Create a Delaunay triangulation from a set of points.

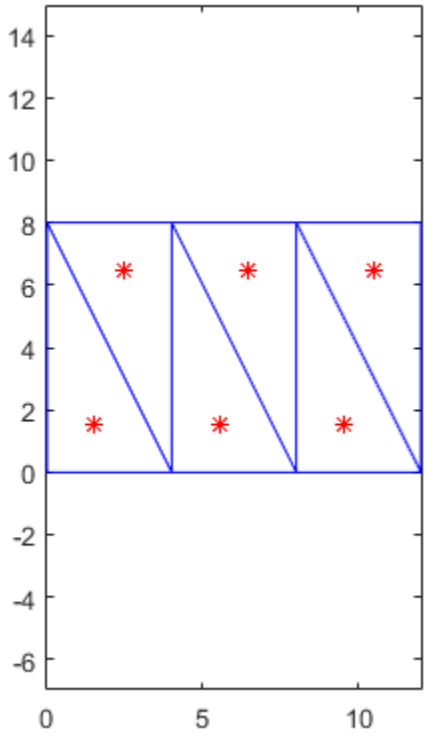
```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
DT = delaunayTriangulation(x,y);
```

Calculate the Cartesian coordinates of the incenters.

```
cc = incenter(DT);
```

Plot the original triangulation and reference points.

```
figure
subplot(1,2,1);
triplot(DT);
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;
axis equal;
```



Create a new triangulation which is a deformed version of `DT` .

```
ti = DT.ConnectivityList;
y = [0 0 0 0 16 16 16 16]';
TR = triangulation(ti,x,y);
```

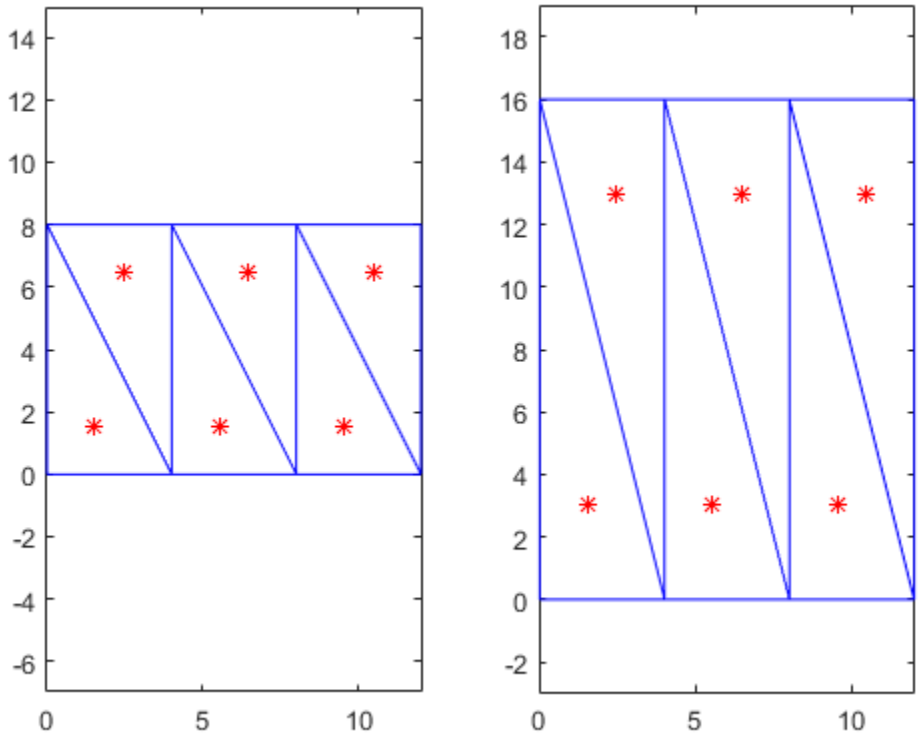
Compute the barycentric coordinates for the incenters of `DT`, and use them to compute the Cartesian coordinates of the analogous points in `TR`.

```
b = cartesianToBarycentric(DT,[1:length(ti)]',cc);
xc = barycentricToCartesian(TR,[1:length(ti)]',b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
```

```
triplot(TR);
hold on;
plot(xc(:,1),xc(:,2),'*r');
hold off;
axis equal;
```



See Also

barycentricToCartesian | delaunayTriangulation

circumcenter

Class: triangulation

Circumcenter of triangle or tetrahedron

Syntax

```
CC = circumcenter(TR,ti)
[CC,r] = circumcenter(TR,ti)
CC = circumcenter(TR)
[CC,r] = circumcenter(TR)
```

Description

`CC = circumcenter(TR,ti)` returns the coordinates of the circumcenter of each triangle or tetrahedron specified in `ti`.

`[CC,r] = circumcenter(TR,ti)` also returns the corresponding radii of the circumscribed circles or spheres.

`CC = circumcenter(TR)` returns the circumcenters of all triangles or tetrahedra in the triangulation.

`[CC,r] = circumcenter(TR)` also returns the corresponding radii of the circumscribed circles or spheres for the entire triangulation.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`ti`

Triangle or tetrahedron IDs, specified as a column vector.

Output Arguments

cc

Circumcenters, returned as a matrix. Each row, `CC(j,:)`, contains the coordinates of the circumcenter of `ti(j)`.

r

Radii of the circumscribed circles or spheres, returned as a vector. The radius at `r(j)` corresponds to the circle or sphere circumscribing `ti(j)`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

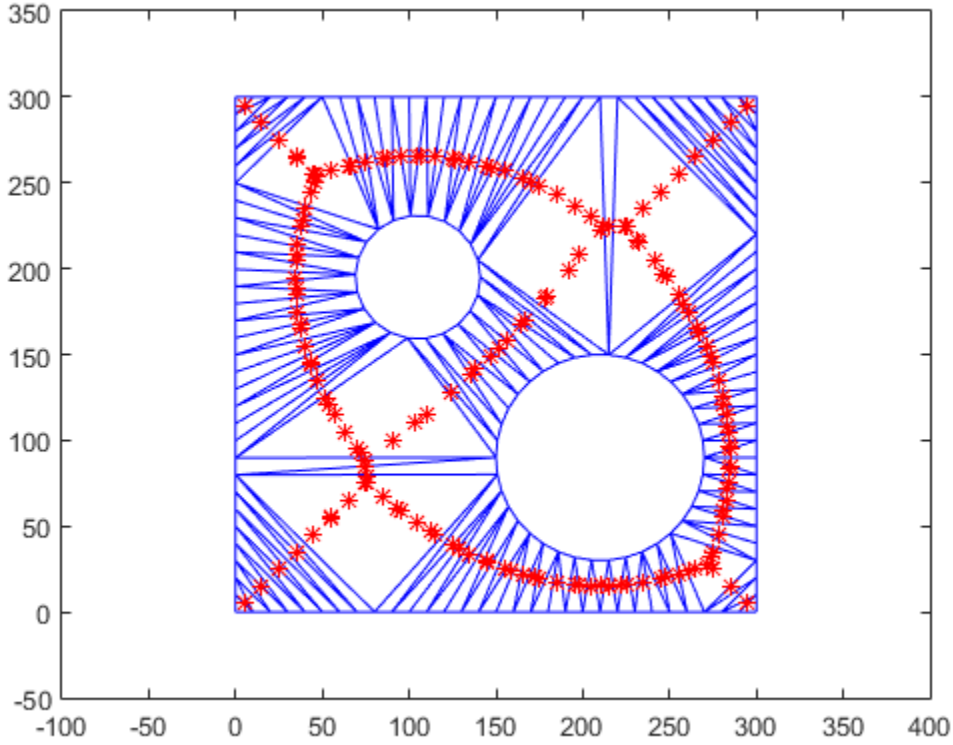
Circumcenters in 2-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
load trimesh2d
TR = triangulation(tri,x,y);
```

Compute the circumcenters.

```
CC = circumcenter(TR);
triplot(TR)
axis([-100 400 -50 350])
hold on
plot(CC(:,1),CC(:,2),'*r')
hold off
```



The circumcenters represent points on the medial axis of the polygon.

Circumcenters in 3-D Delaunay Triangulation

Create the Delaunay triangulation using a random set of points, `P`.

```
P = gallery('uniformdata',10,3,0);
DT = delaunayTriangulation(P);
```

Calculate the circumcenters of the first five tetrahedra in `DT`.

```
CC = circumcenter(DT,[1:5]')
```

```
CC =

    0.9626    0.3892    0.0928
    6.3458    0.2377    3.1814
    0.4820    0.9064    0.5176
   -1.2993    1.8384   -1.2185
   -0.1595    1.0852   -0.2536
```

See Also

delaunayTriangulation | incenter

edgeAttachments

Class: triangulation

Triangles or tetrahedra attached to specified edge

Syntax

```
ti = edgeAttachments(TR,vstart,vend)
ti = edgeAttachments(TR,E)
```

Description

`ti = edgeAttachments(TR,vstart,vend)` returns the triangles or tetrahedra attached to the specified edges. To specify the edges, use the vectors, `vstart` and `vend`. These vectors contain the "Vertex ID" on page 1-6850 at the start and end of each edge.

`ti = edgeAttachments(TR,E)` specifies the starting and ending vertices of each edge in a 2-column matrix, `E`.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`vstart`

IDs of starting vertices, specified as a column vector of vertex IDs. Each ID refers to a vertex at the start of an edge.

`vend`

IDs of ending vertices, specified as a column vector of vertex IDs. Each ID refers to a vertex at the end of an edge.

E

IDs of the edge vertices, specified as a 2-column matrix of vertex IDs. Each row of `E` corresponds to an edge and contains two IDs:

- `E(j,1)` is the ID of the vertex at the start of an edge.
- `E(j,2)` is the ID of the vertex at end of the edge.

Output Arguments

ti

IDs of the triangles or tetrahedra attached to the edges, returned as an `m`-by-`1` cell array. Each cell in `ti` contains the IDs of the attached triangles or tetrahedra. The attachments to the edge located between the vertices, `vstart(j)` and `vend(j)`, or `E(j,:)`, are returned in `ti{j}`. The attachments are returned in a cell array because the number of triangles or tetrahedra associated with each edge can vary.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Edge Attachments in 3-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
load tetmesh
TR = triangulation(tet,X);
```

Define the starting and ending vertices of the edges.

```
vstart = [15; 21];
vend = [936; 716];
```

Find the edge attachments.

```
ti = edgeAttachments(TR,vstart,vend);
```

Examine the attachments to the first edge.

```
ti{1}
```

```
ans =

     927     2060     3438     3423     2583     4690
```

Edge Attachments in 2-D Delaunay Triangulation

Create a Delaunay triangulation.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
DT = delaunayTriangulation(x,y);
```

Find the triangles attached to edge `(1,5)`.

```
ti = edgeAttachments(DT,1,5);
ti{:}
```

```
ans =

     4     1
```

See Also

`delaunayTriangulation` | `edges` | `vertexAttachments`

edges

Class: triangulation

Triangulation edges

Syntax

```
E = edges(TR)
```

Description

`E = edges(TR)` returns the triangulation edges as a matrix of vertex IDs. The vertices referenced in `E` indicate the beginning and end of each edge.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

Output Arguments

E

Edge vertex IDs, returned as a 2-column matrix. Each row of `E` corresponds to an edge and contains two IDs:

- `E(j,1)` is the ID of the vertex at the start of the edge.
- `E(j,2)` is the ID of the vertex at end of the edge.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Edges in a 2-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
load trimesh2d
TR = triangulation(tri,x,y);
```

Find the edges in the triangulation.

```
E = edges(TR);
```

List the first five edges.

```
E(1:5,:)
```

```
ans =

     1     2
     1   118
     1   119
     1   120
     2     3
```

Edges in a 2-D Delaunay Triangulation

Create 2-D Delaunay triangulation from a set of 10 random points.

```
X = gallery('uniformdata',[10, 2],0);
DT = delaunayTriangulation(X);
```

Find the edges in the triangulation.

```
E = edges(DT)
```

```
E =

     1     3
     1     4
     1     5
     1     6
     1     9
     2     4
     2     5
     2     6
     2     7
     2     8
     2    10
     3     4
     3     7
     3    10
     4     6
     4    10
     5     6
     5     8
     5     9
     6     9
     7     8
     7    10
```

See Also

`delaunayTriangulation` | `edgeAttachments`

faceNormal

Class: triangulation

Triangulation face normal

Syntax

```
FN = faceNormal(TR,ti)
FN = faceNormal(TR)
```

Description

`FN = faceNormal(TR,ti)` returns the unit normal vector to each triangle specified by `ti`. The `faceNormal` function supports only 2-D triangulations.

`FN = faceNormal(TR)` returns the unit normal vectors to all triangles.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`ti`

Triangle IDs, specified as a column vector.

Output Arguments

`FN`

Face normals, returned as a matrix. Each row, `FN(j,:)`, is the unit normal vector to triangle `ti(j)`.

If you do not specify `ti`, then `faceNormal` returns the unit normal information for the entire triangulation. In this case, the normal associated with `TR.ConnectivityList(j,:)` is `FN(j,:)`.

Definitions

Triangle ID

A row number of the matrix, `TR.ConnectivityList`. You use this ID to refer a specific triangle.

Examples

Plot Unit Normals to Facets on a Spherical Surface

Create a set of random points on a spherical surface.

```
theta = gallery('uniformdata',[100,1],0)*2*pi;
phi = gallery('uniformdata',[100,1],1)*pi;
x = cos(theta).*sin(phi);
y = sin(theta).*sin(phi);
z = cos(phi);
```

Triangulate the points with `delaunayTriangulation`.

```
DT = delaunayTriangulation(x,y,z);
```

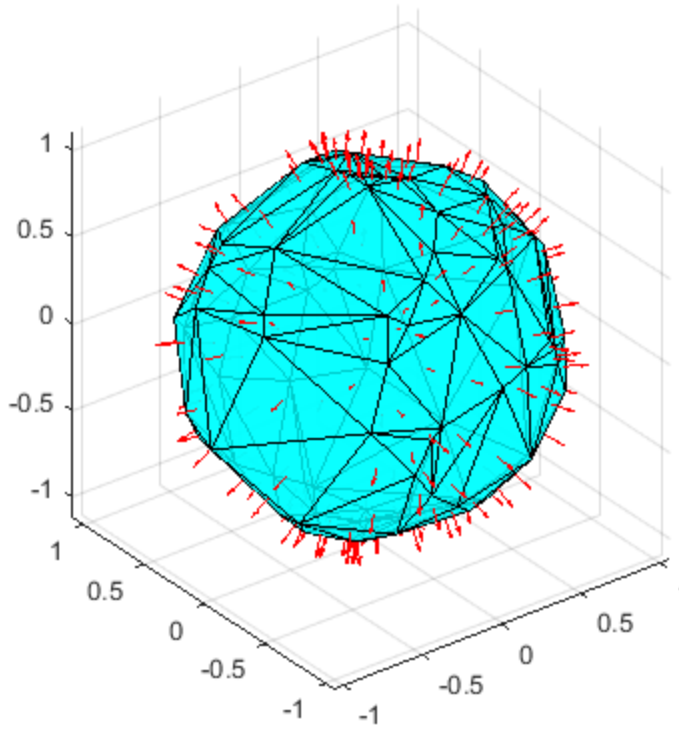
Find the free boundary facets and use them to create a triangulation representation for plotting.

```
[T,Xb] = freeBoundary(DT);
TR = triangulation(T,Xb);
```

Plot the triangulation.

```
figure
trisurf(T,Xb(:,1),Xb(:,2),Xb(:,3), ...
     'FaceColor', 'cyan', 'faceAlpha', 0.8);
axis equal;
hold on;
```

```
% Calulate the incenters and face normals.
P = incenter(TR);
fn = faceNormal(TR);

% Display the normal vectors on the surface.
quiver3(P(:,1),P(:,2),P(:,3), ...
     fn(:,1),fn(:,2),fn(:,3),0.5, 'color','r');
hold off;
```



See Also

delaunayTriangulation | freeBoundary

featureEdges

Class: triangulation

Triangulation sharp edges

Syntax

```
FE = featureEdges(TR,filterangle)
```

Description

`FE = featureEdges(TR,filterangle)` returns the feature edges in a 2-D triangulation.

Use this method to extract the sharp edges in the surface mesh for display purposes. A feature edge is an edge that has any of the following attributes:

- The edge is shared by only one triangle.
- The edge is shared by more than two triangles.
- The edge is shared by a pair of triangles with angular deviation greater than the `filterangle`.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`filterangle`

Filter angle, specified as a scalar value in the range [0,π]. `featureEdges` returns adjacent triangles that have a dihedral angle that deviates from π by an angle greater than `filterangle`.

Output Arguments

FE

Feature edge vertex IDs, returned as a two-column matrix. Each row of `FE` corresponds to a feature edge and contains two IDs:

- `FE(j,1)` is the ID of the vertex at the start of the edge.
- `FE(j,2)` is the ID of the vertex at end of the edge.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Feature Edges Shown on a Surface

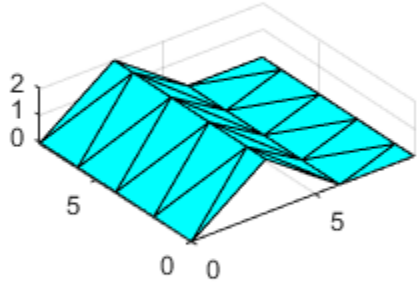
Use `featureEdges` to find the feature edges of a surface and display them on a plot.

Create a surface triangulation.

```
x = [0 0 0 0 0 3 3 3 3 3 6 6 6 6 6 9 9 9 9 9]';
y = [0 2 4 6 8 0 1 3 5 7 8 0 2 4 6 8 0 1 3 5 7 8]';
DT = delaunayTriangulation(x,y);
T = DT(:,:);
```

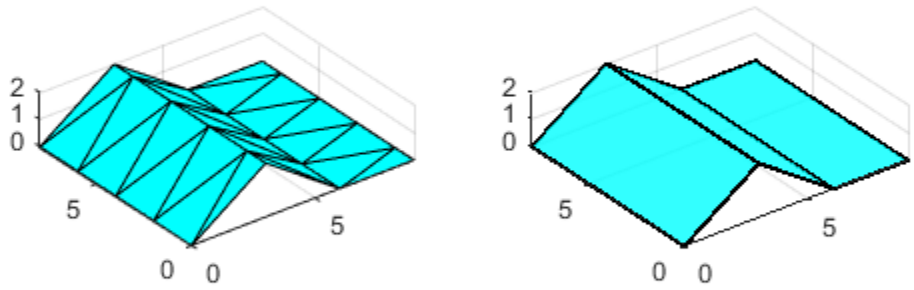
Elevate the 2-D mesh to create a surface.

```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0]';
subplot(1,2,1);
trisurf(T,x,y,z,'FaceColor','cyan');
axis equal;
```



Compute the feature edges using a filter angle of `pi/6`.

```
TR = triangulation(T,x,y,z);
fe = featureEdges(TR,pi/6)';
subplot(1,2,2);
trisurf(TR,'FaceColor','cyan','EdgeColor',...
'none','FaceAlpha',0.8);
axis equal;

% Add the feature edges to the plot.
hold on;
plot3(x(fe), y(fe), z(fe), 'k', 'LineWidth',1.5);
hold off;
```



See Also

`delaunayTriangulation` | `edges`

freeBoundary

Class: triangulation

Triangulation facets referenced by only one triangle or tetrahedron

Syntax

```
FBtri = freeBoundary(TR)
[FBtri,FBpoints] = freeBoundary(TR)
```

Description

`FBtri = freeBoundary(TR)` returns the free boundary facets of `TR.ConnectivityList`. A facet in `TR` is on the free boundary if it is referenced by only one triangle or tetrahedron.

`[FBtri,FBpoints] = freeBoundary(TR)` also returns a matrix containing the vertices of the free boundary facets.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

Output Arguments

`FBtri`

Triangulation connectivity list, returned as a matrix that contains the following information:

- Each row in `FBtri` represents a facet on the free boundary.
- Each element is a “Vertex ID” on page 1-6863.

The vertex IDs in `FBtri` reference a specific matrix, depending on the syntax you choose:

- If you call `freeBoundary` with one output argument, then the elements of `FBtri` are row numbers of `TR.Points`.
- If you call `freeBoundary` with two output arguments, then the elements of `FBtri` are row numbers of `FBpoints`.

`FBpoints`

Free boundary points, returned as a matrix containing the coordinates of the vertices of the free boundary facets. Each row of `FBpoints` contains coordinates of a vertex.

Definitions

Vertex ID

A row number of the matrix, `TR.Points` or FBpoints. Use this ID to refer a specific vertex in the triangulation.

Examples

Surface of 3-D Triangulation

Use `freeBoundary` to extract the facets of a 3-D triangulation that cover the surface of an object.

Load a 3-D triangulation.

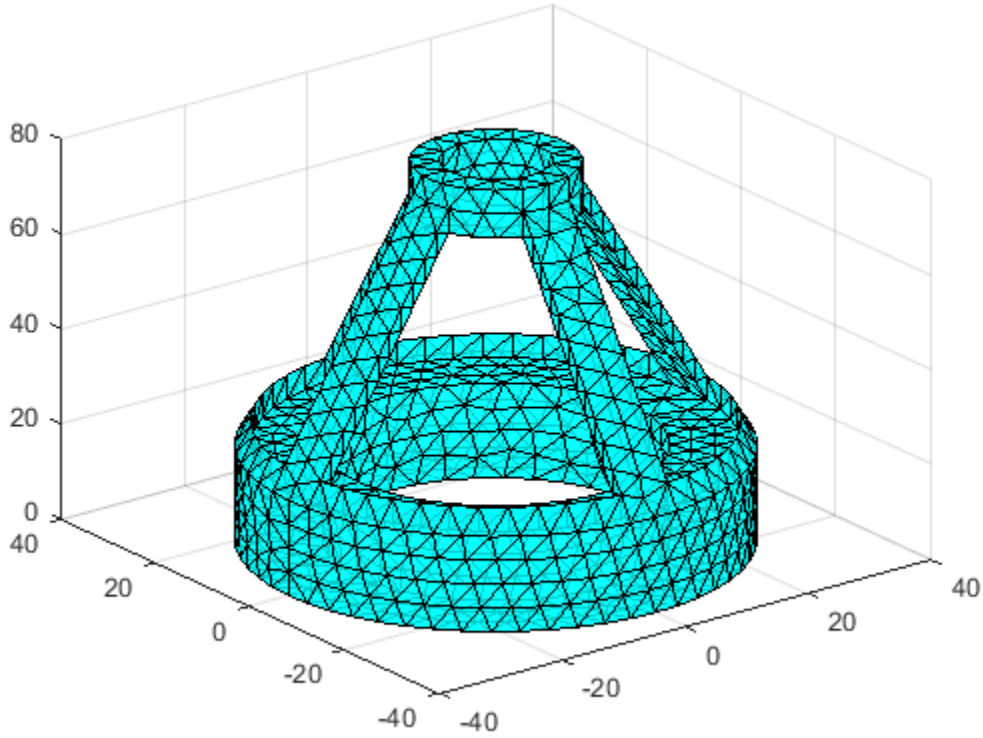
```
load tetmesh;
TR = triangulation(tet,X);
```

Compute the boundary triangulation.

```
[FBtri,FBpoints] = freeBoundary(TR);
```

Plot the boundary triangulation.

```
trisurf(FBtri,FBpoints(:,1),FBpoints(:,2),FBpoints(:,3), ...
       'FaceColor','cyan','FaceAlpha', 0.8);
```



Free Boundary of 2-D Delaunay Triangulation

Use `freeBoundary` when you want to highlight the outer edges of a 2-D Delaunay triangulation.

Create a triangulation from a random set of points.

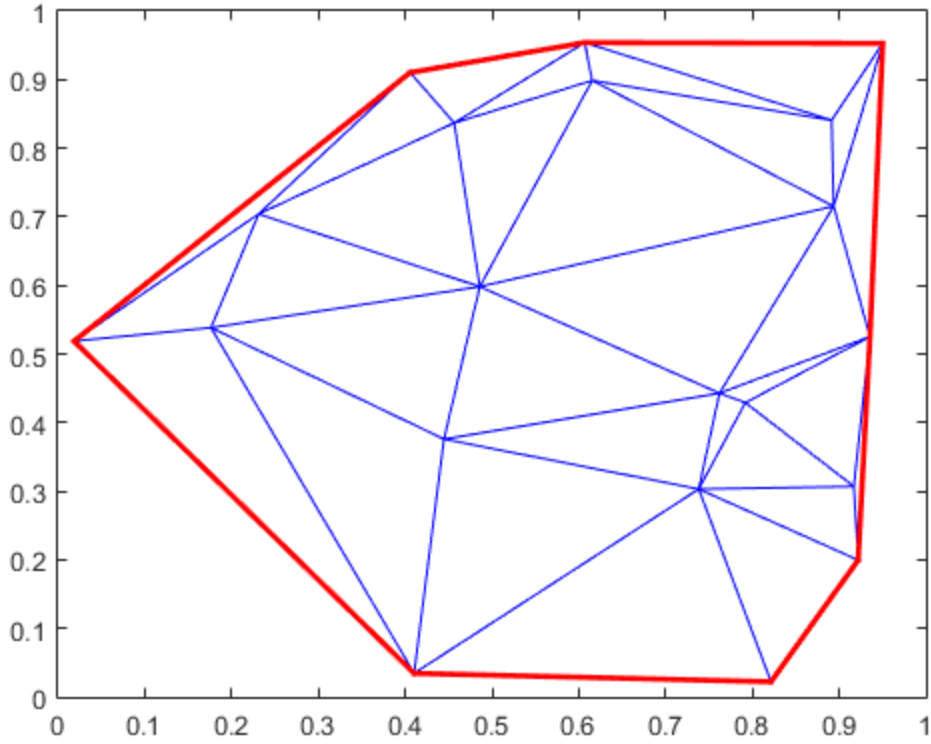
```
x = gallery('uniformdata',[20,1],0);
y = gallery('uniformdata',[20,1],1);
DT = delaunayTriangulation(x,y);
```

Find the free boundary edges.

```
fe = freeBoundary(DT)';
```

Plot the mesh and highlight the free boundary edges in red.

```
triplot(DT);
hold on;
plot(x(fe),y(fe),'-r','LineWidth',2) ;
hold off;
```



See Also

`delaunayTriangulation` | `faceNormal` | `featureEdges`

incenter

Class: triangulation

Incenter of triangle or tetrahedron

Syntax

```
IC = incenter(TR,ti)
[IC,r] = incenter(TR,ti)
```

Description

`IC = incenter(TR,ti)` returns the coordinates of the incenter of each triangle or tetrahedron specified by `ti`.

`[IC,r] = incenter(TR,ti)` also returns the radii of the inscribed circles or spheres.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`ti`

Triangle or tetrahedron IDs, specified as a column vector.

Output Arguments

`IC`

Incenters, returned as a matrix. Each row of `IC` contains the coordinates of an incenter. For example, `IC(j,:)` is the incenter of `ti(j)`.

`r`

Radii of the inscribed circles or spheres, returned as a vector. `r(j)` is the radius of the inscribed circle or sphere whose center is `IC(j,:)`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Find Incenters in 3-D Triangulation

Load a 3-D triangulation.

```
load tetmesh
```

Calculate the incenters of the first five tetrahedra.

```
TR = triangulation(tet,X);
IC = incenter(TR,[1:5]')
```

```
IC =

   -6.1083  -31.0234    8.1439
   -2.1439  -31.0283    5.8742
   -1.9555  -31.9463    7.4112
   -4.3019  -30.8460   10.5169
   -3.1596  -29.3642    6.1851
```

Find Incenters in 2-D Delaunay Triangulation

Create the Delaunay triangulation.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
DT = delaunayTriangulation(x,y);
```

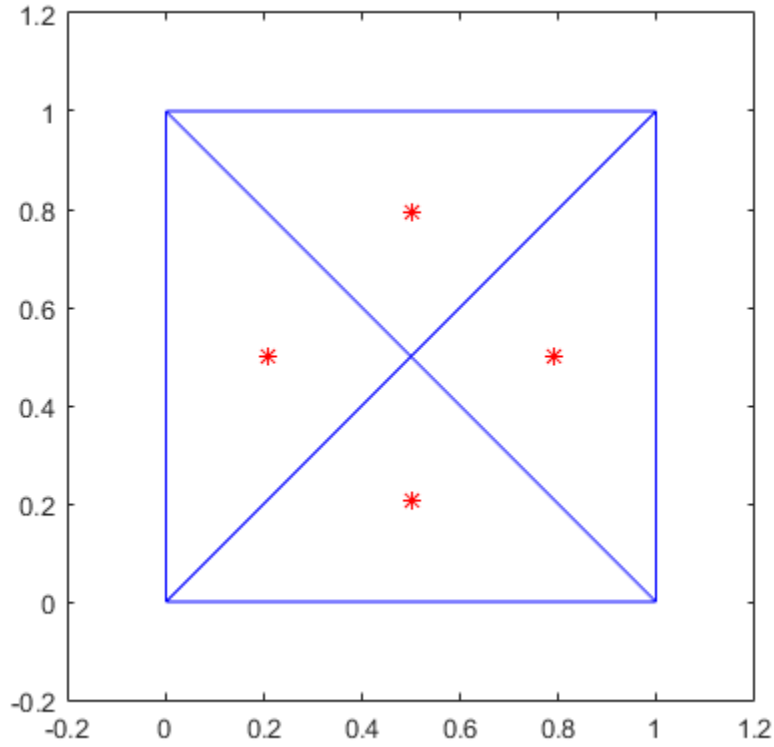
Calculate incenters of the triangles

```
IC = incenter(DT)
```

```
IC =

    0.2071    0.5000
    0.5000    0.7929
    0.7929    0.5000
    0.5000    0.2071
```

Plot the triangles and incenters.

```
figure
triplot(DT)
axis equal
axis([-0.2 1.2 -0.2 1.2])
hold on
plot(IC(:,1),IC(:,2),'*r')
hold off
```



See Also

`circumcenter` | `delaunayTriangulation`

isConnected

Class: triangulation

Test if two vertices are connected by edge

Syntax

```
tf = isConnected(TR,vstart,vend)
tf = isConnected(TR,E)
```

Description

`tf = isConnected(TR,vstart,vend)` returns a `logical` array of `true` or `false` values that indicate whether the specified pairs of vertices are connected by an edge. Element `tf(j)` is `true` when the vertices, `vstart(j)` and `vend(j)`, are connected by an edge.

`tf = isConnected(TR,E)` specifies the edge start and end indices in matrix `E`. Element `tf(j)` is `true` when the vertices, `E(j,1)` and `E(j,2)`, are connected by an edge.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`vstart`

IDs of start vertices, specified as a column vector. Each vertex ID refers to a vertex at the start of an edge.

`vend`

IDs of end vertices, specified as a column vector. Each vertex ID refers to a vertex at the end of an edge.

`E`

IDs of the edge vertices, specified as a two-column matrix. Each row of `E` corresponds to a candidate edge and contains two IDs:

- `E(j,1)` is the ID of the vertex at the start of a candidate edge.
- `E(j,2)` is the ID of the vertex at end of the edge.

Output Arguments

`tf`

Logical values, returned as a column vector. Element `tf(j)` is `true` when either of the following are true:

- The vertices, `vstart(j)` and `vend(j)`, are connected by an edge.
- The vertices, `E(j,1)` and `E(j,2)`, are connected by an edge.

Otherwise, `tf(j)` is false.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Determine Whether Vertices are connected by an Edge in 2-D

Load a 2-D triangulation.

```
load trimesh2d
TR = triangulation(tri,x,y);
```

Determine whether vertices `3` and `117` are connected by an edge.

```
isConnected(TR,3,117)

ans =

     1
```

The vertices are connected by an edge.

Determine whether vertices 3 and 164 are connected by an edge.

```
isConnected(TR,3,164)

ans =

     0
```

The vertices are not connected by an edge.

Determine Whether Vertices are connected by an Edge in 3-D

```
X = gallery('uniformdata',[10,3],0);
DT = delaunayTriangulation(X);
```

Determine whether vertices 2 and 7 are connected by an edge.

```
E = [2 7];
isConnected(DT,E)

ans =

     1
```

The vertices are connected by an edge.

See Also

`delaunayTriangulation` | `edgeAttachments` | `edges`

nearestNeighbor

Class: triangulation

Vertex closest to specified location

Syntax

```
vi = nearestNeighbor(T,QP)
vi = nearestNeighbor(T,qx,qy)
vi = nearestNeighbor(T,qx,qy,qz)
[vi,d] = nearestNeighbor( ___ )
```

Description

`vi = nearestNeighbor(T,QP)` returns the IDs of the vertices closest to the query points in `QP`. Each row in matrix `QP` contains the `x`, `y`, (and possibly `z`) coordinates of a query point.

`vi = nearestNeighbor(T,qx,qy)` specifies the `x` and `y` coordinates of the query points in 2-D as separate column vectors.

`vi = nearestNeighbor(T,qx,qy,qz)` specifies the `x`, `y`, and `z` coordinates of the query points in 3-D as separate column vectors.

`[vi,d] = nearestNeighbor( ___ )` also returns the Euclidean distance between each query point and its nearest neighbor. You can specify both output arguments with any of the previous syntaxes.

Note: `nearestNeighbor` does not support Delaunay triangulations with constrained edges.

Input Arguments

`T` — Triangulation representation

`triangulation` object | `delaunayTriangulation` object

Triangulation representation, specified as a `triangulation` object or `delaunayTriangulation` object. For more information, see `triangulation` or `delaunayTriangulation`.

`QP` — Query points

matrix with 2 columns (2-D) | matrix with 3 columns (3-D)

Query points, specified as a matrix with 2 or 3 columns. `QP` contains the `x`, `y`, (and possibly `z`) coordinates of the query points.

`qx` — Query point x-coordinates

column vector

Query point x-coordinates, specified as a column vector.

`qy` — Query point y-coordinates

column vector

Query point y-coordinates, specified as a column vector.

`qz` — Query point z-coordinates

column vector

Query point z-coordinates, specified as a column vector.

Output Arguments

`vi` — Vertex IDs of nearest neighbor

column vector

Vertex IDs of the nearest neighbor, returned as a column vector. `vi(j)` is the vertex ID of the nearest neighbor to the query point `QP(j,:)`.

`d` — Distance to nearest neighbor

column vector

Distance to the nearest neighbor, returned as a column vector of the same length as `vi`. The value at `d(j)` is the Euclidean distance between the `j`th query point `QP(j,:)` and its nearest neighbor.

Examples

Determine Nearest Neighbors of Query Points

Define the points and connectivity of the triangulation.

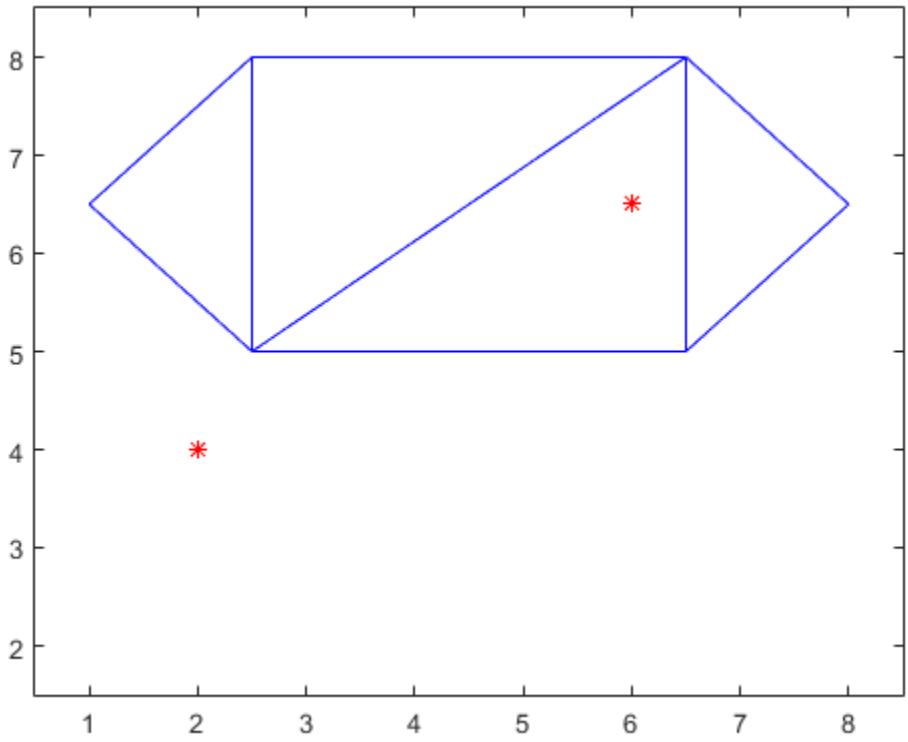
```
P = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
T = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(T,P);
```

Define two query points.

```
QP = [2 4; 6 6.5];
```

Plot the triangulation and the query points.

```
triplot(TR)
hold on
plot(QP(:,1),QP(:,2),'r*')
ylim([1.5 8.5])
xlim([0.5 8.5])
```

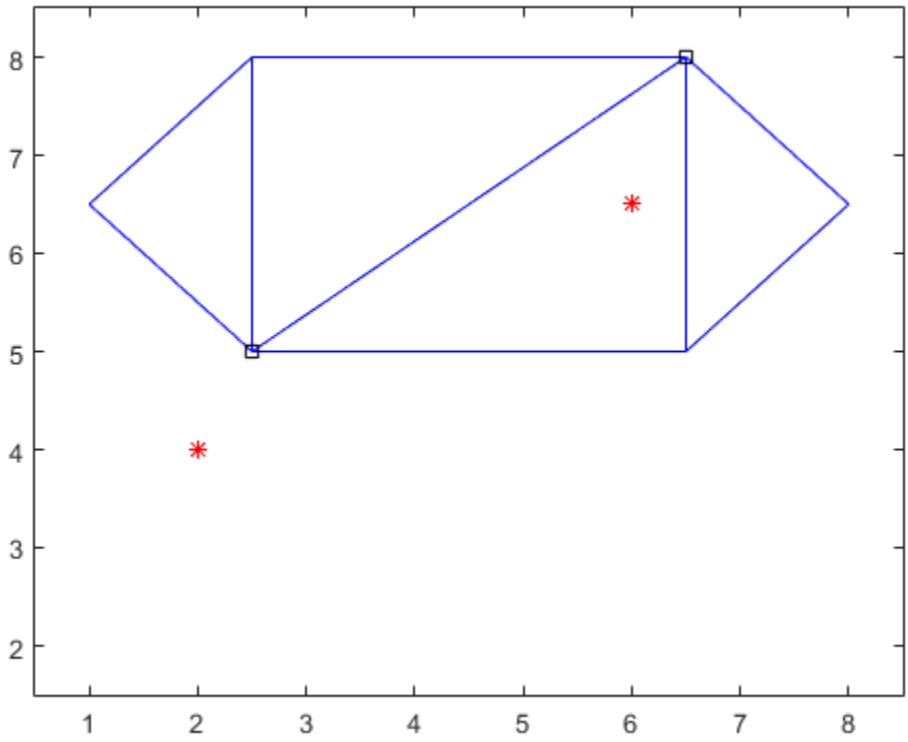


Find the nearest neighbors to the query points, as well as the distances between them.

```
[vi,d] = nearestNeighbor(TR,QP);
```

Highlight the points in the triangulation that are the nearest neighbors to the query points.

```
TP = P(vi,:);
plot(TP(:,1),TP(:,2),'ks')
```



Examine the distance between each query point and its nearest neighbor.

```
d
```

```
d =

    1.1180
    1.5811
```

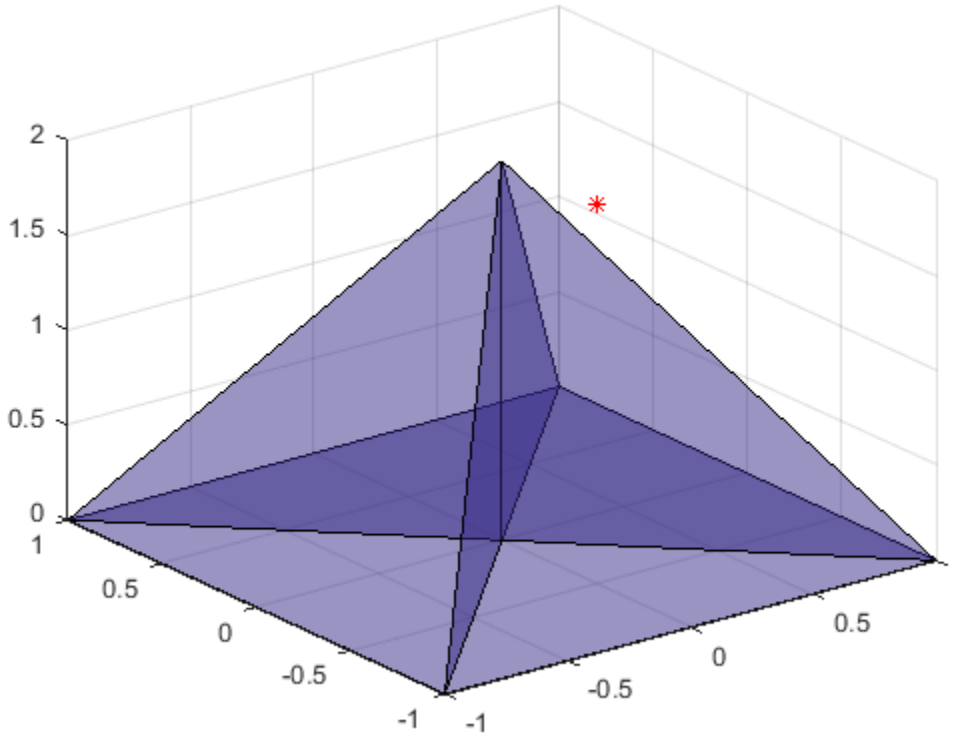
Locate Nearest Neighbors in 3-D Delaunay Triangulation

Create a 3-D Delaunay Triangulation.

```
P = [1 1 0; -1 1 0; -1 -1 0; 1 -1 0; 0 0 2; 0 0 0];
DT = delaunayTriangulation(P);
```

Plot the triangulation and a query point.

```
tri = DT(:,:);
trisurf(tri,P(:,1),P(:,2),P(:,3),'FaceAlpha',0.5)
hold on
QP = [0 -0.5 2];
plot3(QP(1),QP(2),QP(3),'r*')
```



Find the coordinates for the nearest-neighbor of the query point.

```
nnVertexID = nearestNeighbor(DT,QP);
```

```
nnCoordinates = DT.Points(nnVertexID,:)

nnCoordinates =

     0     0     2
```

See Also

`delaunayTriangulation` | `triangulation`

More About

- "Spatial Searching"

neighbors

Class: triangulation

Neighbors to specified triangle or tetrahedron

Syntax

```
N = neighbors(TR,ti)
N = neighbors(TR)
```

Description

`N = neighbors(TR,ti)` returns the neighbors of the triangles or tetrahedra specified in `ti`.

`N = neighbors(TR)` returns the neighbors of all the triangles or tetrahedra.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`ti`

Triangle or tetrahedron IDs, specified as a column vector.

Output Arguments

`TN`

IDs of the neighboring triangles or tetrahedra, returned as a matrix. The elements in `TN(i,:)` are the neighbors associated with `ti(i)`.

By convention, `TN(i,j)` is the neighbor opposite the `j`th vertex of `ti(i)`. If a triangle or tetrahedron has one or more boundary facets, the nonexistent neighbors are represented as `NaN` values in `TN`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Query Neighbors in 2-D Delaunay Triangulation

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[10,1],0);
y = gallery('uniformdata',[10,1],1);
DT = delaunayTriangulation(x,y);
```

Find the neighbors of the first triangle.

```
TN = neighbors(DT,1)
```

```
TN =

   NaN     4     5
```

The first triangle has a boundary edge and two neighbors.

Examine the vertex IDs of the first neighbor, `TN(2)`.

```
DT.ConnectivityList(TN(2),:)
```

```
ans =

     2     4     7
```

Query Neighbors in 3-D Triangulation

Create the Delaunay triangulation.

```
load tetmesh
TR = triangulation(tet,X);
```

Find the neighbors to each triangle in the triangulation.

```
TN = neighbors(TR);
```

Find the neighbors of the fifth tetrahedron.

```
TN(5,:)
```

```
ans =

        2360        1539           2        1851
```

Examine the vertex IDs of the first neighbor, `TN(1)`.

```
TR.ConnectivityList(TN(1),:)
```

```
ans =

        1093         891         893         858
```

See Also

`delaunayTriangulation` | `edgeAttachments`

pointLocation

Class: triangulation

Triangle or tetrahedron containing specified point

Syntax

```
ti = pointLocation(T,QP)
ti = pointLocation(T,qx,qy)
ti = pointLocation(T,qx,qy,qz)
[ti,B] = pointLocation( ___ )
```

Description

`ti = pointLocation(T,QP)` returns the enclosing triangles or tetrahedra for each query point in `QP`. Each row in matrix `QP` contains the `x`, `y`, (and possibly `z`) coordinates of a query point.

`ti = pointLocation(T,qx,qy)` specifies the `x` and `y` coordinates of the query points in 2-D as separate column vectors.

`ti = pointLocation(T,qx,qy,qz)` specifies the `x`, `y`, and `z` coordinates of the query points in 3-D as separate column vectors.

`[ti,B] = pointLocation( ___ )` also returns the barycentric coordinates of each query point with respect to its enclosing triangle or tetrahedron. You can specify both output arguments with any of the previous syntaxes.

Input Arguments

`T` — Triangulation representation

`triangulation` object | `delaunayTriangulation` object

Triangulation representation, specified as a `triangulation` object or `delaunayTriangulation` object. For more information, see `triangulation` or `delaunayTriangulation`.

QP — Query points

matrix with 2 columns (2-D) | matrix with 3 columns (3-D)

Query points, specified as a matrix with 2 or 3 columns. `QP` contains the `x`, `y`, (and possibly `z`) coordinates of the query points.

qx — Query point x-coordinates

column vector

Query point x-coordinates, specified as a column vector.

qy — Query point y-coordinates

column vector

Query point y-coordinates, specified as a column vector.

qz — Query point z-coordinates

column vector

Query point z-coordinates, specified as a column vector.

Output Arguments

ti — IDs of enclosing triangles or tetrahedra

column vector

IDs of the enclosing triangles or tetrahedra, returned as a column vector. The values in `ti` correspond to the row numbers of the triangulation connectivity matrix `T.ConnectivityList`. For a query point `QP(k,:)`, the triangle or tetrahedron enclosing the point is `ti(k)`.

`ti` contains `NaN` values for points that are not located in a triangle or tetrahedron of the triangulation, `T`.

B — Barycentric coordinates

matrix

Barycentric coordinates, returned as a matrix. The values at `B(j,:)` are the barycentric coordinates of the `j`th query point `QP(j,:)` with respect to `ti(j)`.

Examples

Find Enclosing Triangles of Query Points

Define the points and connectivity of the triangulation.

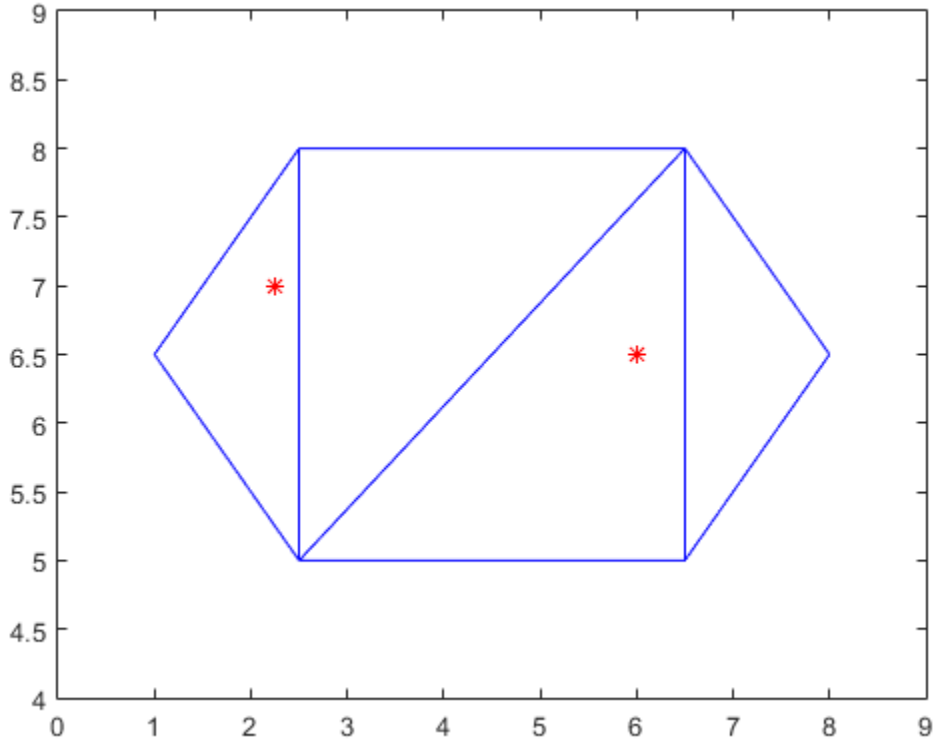
```
P = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
T = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(T,P);
```

Define two query points.

```
QP = [2.25 7; 6 6.5];
```

Plot the triangulation and the query points.

```
triplot(TR)
hold on
plot(QP(:,1),QP(:,2),'r*')
ylim([4 9])
xlim([0 9])
```



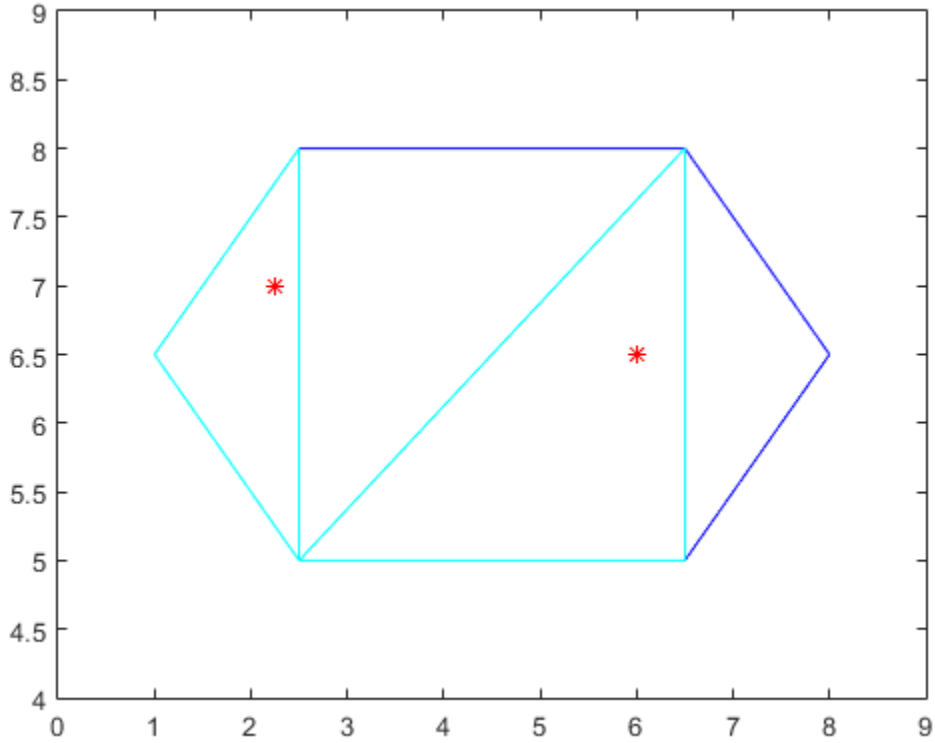
Determine which triangles in the triangulation enclose each query point.

```
ti = pointLocation(TR,QP)
```

```
ti =

     1
     3
```

Highlight the triangles that enclose each of the query points.

```
triplot(TR.ConnectivityList(ti,:),P(:,1),P(:,2),'c')
```



Find Enclosing Tetrahedra and Barycentric Coordinates in 3-D

Create a 3-D Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[20 1],0);
y = gallery('uniformdata',[20 1],1);
z = gallery('uniformdata',[20 1],2);
DT = delaunayTriangulation(x,y,z);
```

Find the tetrahedra that enclose the query points and calculate the barycentric coordinates of the query points.

```
QP = [0.7 0.6 0.3; 0.5 0.5 0.5];
[ti,B] = pointLocation(DT,QP)
```

```
ti =

    57
    56

B =

    0.2796    0.0184    0.5286    0.1734
    0.3687    0.0149    0.5343    0.0821
```

See Also

`delaunayTriangulation` | `triangulation`

More About

- "Spatial Searching"

size

Class: triangulation

Size of triangulation connectivity list

Syntax

```
sz = size(TR)
```

Description

`sz = size(TR)` returns the size the triangulation connectivity list.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

Output Arguments

sz

Size of connectivity list, returned as a two-element row vector. The first element, `sz(1)` is the number of triangles or tetrahedra in the triangulation, and `sz(2)` is the number of vertices per triangle or tetrahedron.

Examples

Size of 2-D Triangulation

Create a triangulation.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];

TR = triangulation(T,P);
```

Get the size of the connectivity list.

```
size(TR)
```

```
ans =

     4     3
```

The triangulation has 4 triangles, and each triangle has 3 vertices.

See Also

`delaunayTriangulation` | `size`

vertexAttachments

Class: triangulation

Triangles or tetrahedra attached to specified vertex

Syntax

```
ti = vertexAttachments(TR,vi)
ti = vertexAttachments(TR)
```

Description

`ti = vertexAttachments(TR,vi)` returns the triangles or tetrahedra attached to the specified vertices, `vi`.

`ti = vertexAttachments(TR)` returns the triangles or tetrahedra attached to every vertex in the triangulation.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`vi`

Query vertices, specified as a column vector of vertex IDs.

Output Arguments

`ti`

IDs of the triangles or tetrahedra attached to the vertices, returned as an `m`-by-`1` cell array. Each cell in `ti` contains the IDs of the attached triangles or tetrahedra. `ti{j}` contains the attachments to the vertex, `vi(j)`. The attachments are returned in a cell

array because the number of triangles or tetrahedra associated with each vertex can vary.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Attachments to Specific Vertex in 2-D Delaunay Triangulation

Locate and plot the attachments to a specific vertex.

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[20,1],0);
y = gallery('uniformdata',[20,1],1);
DT = delaunayTriangulation(x,y);
```

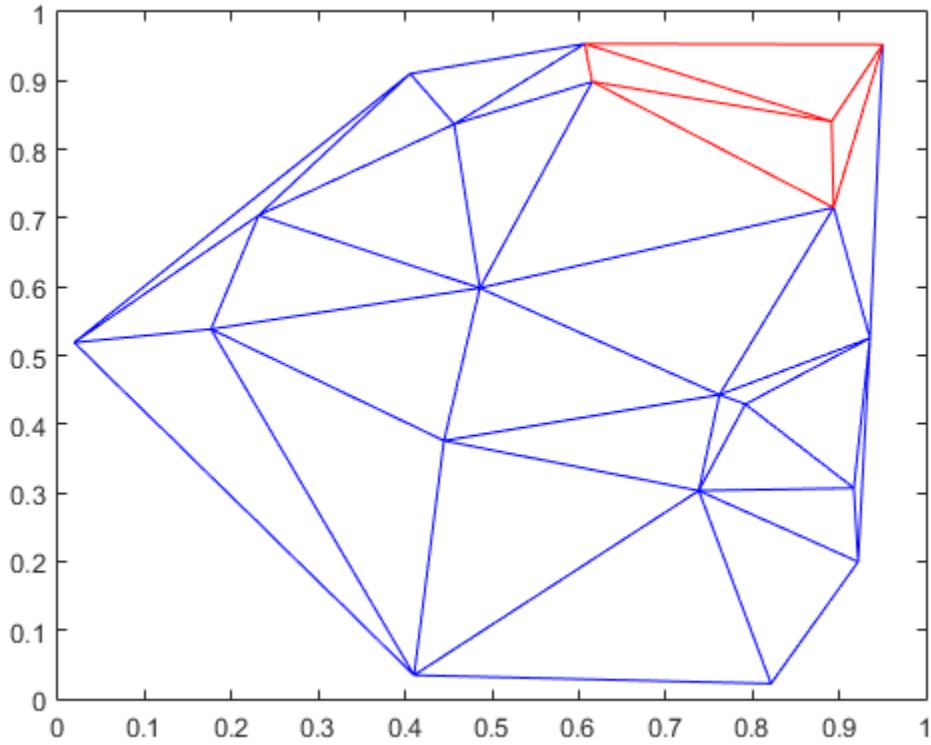
Find the triangles attached to the fifth vertex.

```
ti = vertexAttachments(DT,5);
ti{:}
```

```
ans =

    18     23     21     22
```

Plot the triangulation. Plot the triangles attached to vertex 5 in red.

```
triplot(DT)
hold on

triplot(DT(ti{:},:),x,y,'Color','r')  % vertex 5 (in red)
hold off
```



See Also

delaunayTriangulation | edgeAttachments

vertexNormal

Class: triangulation

Triangulation vertex normal

Syntax

```
VN = vertexNormal(TR,vi)
VN = vertexNormal(TR)
```

Description

`VN = vertexNormal(TR,vi)` returns the unit normal vector to each of the specified vertices in `vi`.

`VN = vertexNormal(TR)` returns the normal information for all vertices in the triangulation.

Input Arguments

`TR`

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

`vi`

IDs of vertices to query, specified as a column vector. Each element in `vi` is a "Vertex ID" on page 1-6895.

Output Arguments

`VN`

Vertex normals, returned a matrix. Each row, `VN(j,:)`, is the unit normal vector at the vertex `vi(j)`. The vector at `VN(j,:)` is the average unit normal of the faces attached to vertex `vi(j)`.

If you do not specify `vi`, then `vertexNormal` returns the unit normal information for all vertices in the triangulation. In this case, the normal associated with `TR.Points(j,:)` is `VN(j,:)`.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Unit Normal Vectors to the Surface of a Cube

Create a 3-D triangulation representing the volume of a cube.

```
[X,Y,Z] = meshgrid(1:4);
X = X(:);
Y = Y(:);
Z = Z(:);
dt = delaunayTriangulation(X,Y,Z);
```

Find the surfaces of the cube and isolate them in a 2-D triangulation.

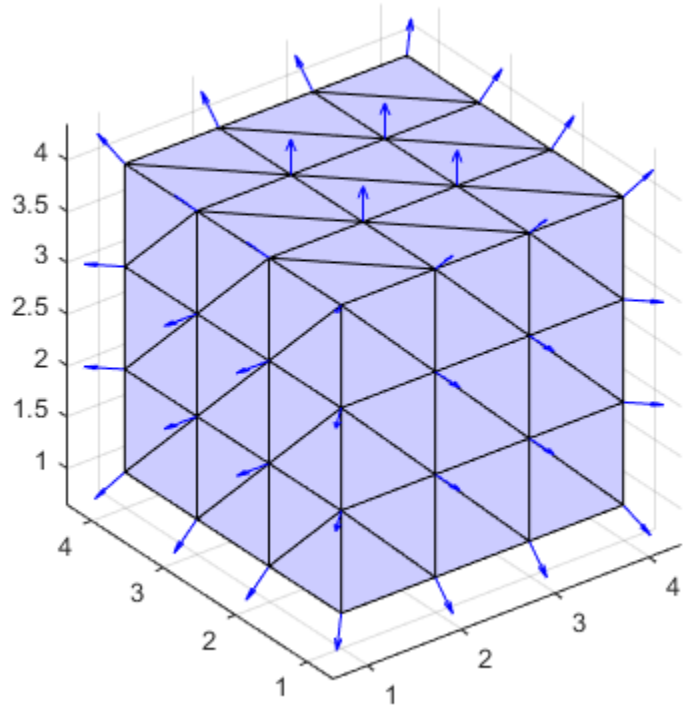
```
[Tfb,Xfb] = freeBoundary(dt);
TR = triangulation(Tfb,Xfb);
```

Find the unit normal vectors to the triangles on the surface of the cube.

```
vn = vertexNormal(TR);
```

Plot the results.

```
trisurf(TR,'FaceColor', [0.8 0.8 1.0]);
axis equal
hold on
quiver3(Xfb(:,1),Xfb(:,2),Xfb(:,3),...
     vn(:,1),vn(:,2),vn(:,3),0.5,'color','b');
hold off
```



See Also

`delaunayTriangulation` | `freeBoundary`

readtable

Create table from file

Syntax

```
T = readtable(filename)
T = readtable(filename,Name,Value)
```

Description

`T = readtable(filename)` creates a table by reading column oriented data from a file.

`readtable` determines the file format from the file name's extension:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

`readtable` creates one variable in `T` for each column in the file and reads variable names from the first row of the file. By default, the variables created are `double` if the entire column is numeric, or cell arrays of strings if any element in a column is not numeric.

`T = readtable(filename,Name,Value)` creates a table from a file with additional options specified by one or more Name,Value pair arguments.

For example, you can specify not to read the first row of the file as variable names.

Examples

Create Table from Text File

Create a file, `myCsvTable.dat`, that contains the following comma-separated column oriented data.

```
LastName,Gender,Age,Height,Weight,Smoker
```

```
Smith,M,38,71,176,1
Johnson,M,43,69,163,0
Williams,F,38,64,131,0
Jones,F,40,67,133,0
Brown,F,49,64,119,0
```

Create a table from the comma-separated text file.

```
T = readtable('myCsvTable.dat')
```

```
T =

     LastName      Gender    Age    Height    Weight    Smoker
    ___________    ______    ___    ______    ______    ______

    'Smith'        'M'       38     71        176       1
    'Johnson'      'M'       43     69        163       0
    'Williams'     'F'       38     64        131       0
    'Jones'        'F'       40     67        133       0
    'Brown'        'F'       49     64        119       0
```

`T` contains one variable for each column in the file and `readtable` the entries in first line of the file as variable names.

Create Table from Text File without Column Headings

Create a file, `mySpaceDelimTable.txt`, that contains the following space delimited, column oriented data.

```
M 45 45 NY true
F 41 32 CA false
M 40 34 MA false
```

Create a table from the space delimited text file that does not contain variable names as column headings.

```
T = readtable('mySpaceDelimTable.txt',...
    'Delimiter',' ','ReadVariableNames',false)
```

```
T =

    Var1    Var2    Var3    Var4     Var5
    ____    ____    ____    ____    _______

    'M'     45      45      'NY'    'true'
```

```
    'F'      41      32      'CA'    'false'
    'M'      40      34      'MA'    'false'
```

`T` contains default variable names.

Create and Format Table from Text File

Create a file, `myCsvTable.dat`, that contains the following comma-separated column oriented data.

```
LastName,Gender,Age,Height,Weight,Smoker
Smith,M,38,71,176,1
Johnson,M,43,69,163,0
Williams,F,38,64,131,0
Jones,F,40,67,133,0
Brown,F,49,64,119,0
```

Create a table from the comma-separated text file. Format the first two variables as strings, the third variable as `uint32`, and the next two variables as double-precision, floating-point numbers. Format the last variable as a string.

```
T = readtable('myCsvTable.dat','Format','%s%s%u%f%f%s')
```

```
T =

     LastName      Gender    Age    Height    Weight    Smoker
    ___________    ______    ___    ______    ______    ______

    'Smith'        'M'       38     71        176       '1'
    'Johnson'      'M'       43     69        163       '0'
    'Williams'     'F'       38     64        131       '0'
    'Jones'        'F'       40     67        133       '0'
    'Brown'        'F'       49     64        119       '0'
```

The conversion specifiers are `%s` for a cell array of strings, `%f` for `double`, and `%u` for `uint32`.

Read Foreign-Language Dates from Text File

Create a sample file named `myfile.txt` that contains comma-separated values. The first column of values contains dates in German and the second and third columns are numeric values.

```
fileID = fopen('myfile.txt','w','n','ISO-8859-15');
```

```
fprintf(fileID,'1 Januar 2014, 20.2, 100.5 \n');
fprintf(fileID,'1 Februar 2014, 21.6, 102.7 \n');
fprintf(fileID,'1 März 2014, 20.7, 99.8 \n');
fclose(fileID);
```

The sample file looks like this:

```
1 Januar 2014, 20.2, 100.5
1 Februar 2014, 21.6, 102.7
1 März 2014, 20.7, 99.8
```

Read the sample file using `readtable`. The conversion specifiers are `%D` for a date and `%f` for floating-point values. Specify the file encoding using the `FileEncoding` name-value pair argument. Specify the format and locale of the dates using the `DateLocale` name-value pair argument.

```
T = readtable('myfile.txt','ReadVariableNames',false,...
    'Format','%{dd MMMM yyyy}D %f %f',...
    'FileEncoding','ISO-8859-15',...
    'DateLocale','de_DE')
```

```
T =

          Var1           Var2    Var3
    ________________     ____    _____

    01 January 2014      20.2    100.5
    01 February 2014     21.6    102.7
    01 March 2014        20.7     99.8
```

Create Table from Spreadsheet Including Row Names

Create a table from a spreadsheet that contains variable names in the first row and row names in the first column.

```
T = readtable('patients.xls','ReadRowNames',true);
```

Display the first five rows and first four variables of the table.

```
T(1:5,1:4)
```

```
ans =

                    Gender      Age             Location                Height
```

```
                ________    ___    ____________________________    ______

    Smith       'Male'      38     'County General Hospital'       71
    Johnson     'Male'      43     'VA Hospital'                   69
    Williams    'Female'    38     'St. Mary's Medical Center'     64
    Jones       'Female'    40     'VA Hospital'                   67
    Brown       'Female'    49     'County General Hospital'       64
```

View the `DimensionNames` property of the table.

```
T.Properties.DimensionNames
```

```
ans =

    'LastName'    'Variable'
```

`'LastName'` is the name in the first column of the first row of the spreadsheet.

Read Specific Range of Data from Spreadsheet

Create a table from the 5-by-3 rectangular region between the two corners `C2` and `E6` on the spreadsheet `patients.xls`. Do not use the first row of this region as variable names.

```
T = readtable('patients.xls',...
    'Range','C2:E6',...
    'ReadVariableNames',false)
```

```
T =

    Var1              Var2                    Var3
    ____    ____________________________    ____

    38      'County General Hospital'       71
    43      'VA Hospital'                   69
    38      'St. Mary's Medical Center'     64
    40      'VA Hospital'                   67
    49      'County General Hospital'       64
```

`T` contains default variable names.

- “Access Data in a Table”

Input Arguments

`filename` — Name of file to read

string

Name of the file to read, specified as a string. If `filename` includes the file extension, then `readtable` determines the file format from the extension. Otherwise, you must specify the `'FileType'` name-value pair arguments to indicate the type of file.

On Windows systems with Microsoft Excel software, `readtable` reads any Excel spreadsheet file format recognized by your version of Excel. If your system does not have Excel for Windows, `readtable` operates in `basic` import mode, and reads only XLS, XLSX, XLSM, XLTX, and XLTM files.

For a delimited text file, `readtable` converts empty fields in the file to either `NaN` (for a numeric variable) or the empty string (for a string-valued variable). All lines in the text file must have the same number of delimiters. `readtable` ignores insignificant white space in the file.

Example: `'myFile.xlsx'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ReadVariableNames',false` indicates that the first row of the file does not correspond to variable names.

Delimited Text Files and Spreadsheet Files

`'FileType'` — Type of file

`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and the string `'text'` or `'spreadsheet'`.

You must specify the `'FileType'` name-value pair argument if the `filename` input argument does not include the file extension or if the extension is other than one of the following.

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

Example: `'FileType','text'`

`'ReadVariableNames'` — Indicator for reading first row as variable names

`true` (default) | `false` | `1` | `0`

Indicator for reading the first row as variable names, specified as the comma-separated pair consisting of `'ReadVariableNames'` and either `true`, `false`, `1`, or `0`.

`true`	The first row of the region to read contains the variable names for the table. This is the default behavior. If both the `'ReadVariableNames'` and `'ReadRowNames'` logical indicators are `true`, then `readtable` saves the name in the first column of the first row of the region to read as the first dimension name in the property, `T.Properties.DimensionNames`.
`false`	The first row of the region to read contains the first row of data in the table. `readtable` creates default variable names of the form `'Var1',...,'VarN'`, where `N` is the number of variables.

`'ReadRowNames'` — Indicator for reading the first column as row names

`false` (default) | `true` | `0` | `1`

Indicator for reading first column as row names, specified as the comma-separated pair consisting of `'ReadRowNames'` and either `false`, `true`, `0`, or `1`.

`false`	The first column of the region to read contains the first variable in the table. It does not contain the row names for the table. This is the default behavior.
`true`	The first column of the region to read contains the row names for the table. If both the `'ReadVariableNames'` and `'ReadRowNames'` logical indicators are `true`, then `readtable` saves the name in the first column of the first row of the region to read as the first dimension name in the property, `T.Properties.DimensionNames`.

'TreatAsEmpty' — Strings to treat as empty value

string | cell array of strings

Strings to treat as empty value, specified as the comma-separated pair consisting of 'TreatAsEmpty' and a string or a cell array of strings. Table elements corresponding to these are set to NaN.

'TreatAsEmpty' only applies to numeric columns in the file, and readtable does not accept numeric literals, such as '-99'.

Example: 'TreatAsEmpty','N/A' sets N/A within numeric columns to NaN.

Example: 'TreatAsEmpty',{'.','NA','N/A'} sets ., NA and N/A within numeric columns to NaN.

Delimited Text Files Only

When reading delimited text files, you can specify any of the name-value pair arguments listed here. Also, if you specify the 'Format' name-value pair argument, you can specify any of the name-value pair arguments accepted by the textscan function.

'Delimiter' — Field delimiter character

string

Field delimiter character, specified as the comma-separated pair consisting of 'Delimiter' and one of the following strings:

','	Comma. This is the default behavior.
'comma'	
' '	Space
'space'	
'\t'	Tab
'tab'	
';'	Semicolon
'semi'	
'|'	Vertical bar
'bar'	

Example: `'Delimiter','space'`

`'HeaderLines'` — Number of lines to skip at beginning of file

`0` (default) | positive integer

Number of lines to skip at beginning of file, specified as the comma-separated pair consisting of `'HeaderLines'` and a positive integer.

Data Types: `single` | `double`

`'Format'` — Format of columns in file

string of one or more conversion specifiers

Format of the columns in the file, specified as the comma-separated pair consisting of `'Format'` and a string of one or more conversion specifiers. The conversion specifiers are the same as those accepted by the `textscan` function.

Specifying the format can significantly improve speed for some large files. If you do not specify a value for `Format`, then `readtable` uses `%q` to interpret nonnumeric columns. The `%q` specifier reads a string and omits double quotation marks (") if appropriate.

By default, the variables created are either `double`, if the entire column is numeric, or cell arrays of strings, if any element in a column is not numeric.

`'FileEncoding'` — Character encoding scheme

`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'FileEncoding'` and one of the following strings.

`'Big5'`	`'ISO-8859-1'`	`'windows-932'`
`'GB2312'`	`'ISO-8859-2'`	`'windows-936'`
`'EUC-KR'`	`'ISO-8859-3'`	`'windows-949'`
`'EUC-JP'`	`'ISO-8859-4'`	`'windows-950'`
`'GBK'`	`'ISO-8859-9'`	`'windows-1250'`
`'KSC_5601'`	`'ISO-8859-13'`	`'windows-1251'`
`'Macintosh'`	`'ISO-8859-15'`	`'windows-1252'`
`'Shift_JIS'`		`'windows-1253'`
`'US-ASCII'`		`'windows-1254'`

`'UTF-8'` `'windows-1257'`

The default encoding is system-dependent.

Data Types: `char`

Spreadsheet Files Only

`'Sheet'` — Worksheet to read

1 (default) | positive integer indicating worksheet index | string containing worksheet name

Worksheet to read, specified as the comma-separated pair consisting of `'Sheet'` and a positive integer indicating the worksheet index or a string containing the worksheet name. The worksheet name string cannot contain a colon (`:`). To determine the names of sheets in a spreadsheet file, use `[status,sheets] = xlsfinfo(filename)`.

Example: `'Sheet',2`

`'Range'` — Rectangular portion of worksheet to read

string

Rectangular portion of the worksheet to read, specified as the comma-separated pair consisting of `'Range'` and a string.

Specify the range using the syntax `'Corner1:Corner2'`, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `'Range'` name-value pair argument is not case sensitive, and uses Excel A1 reference style (see Excel help).

If the spreadsheet contains figures or other nontablular information, use the `'Range'` name-value pair argument to read only the tabular data. By default, `readtable` reads data from a spreadsheet contiguously out to the right-most column that contains data, including any empty columns that precede it.

Example: `'Range','D2:H4'`

`'Basic'` — Indicator for reading in `basic` mode

`true` | `false` | `1` | `0`

Indicator for reading in `basic` mode, specified as the comma-separated pair consisting of `'Basic'` and either `true`, `false`, `1`, or `0`.

`basic` mode is the default for systems without Excel for Windows. In `basic` mode, `readtable`:

- Reads XLS, XLSX, XLSM, XLTX, and XLTM files only.
- Does not support the `'Range'` name-value pair argument when reading XLS files.
- Imports all dates as Excel serial date numbers. Excel serial date numbers use a different reference date than MATLAB date numbers.

Output Arguments

`T` — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

More About

- "Ways to Import Text Files"
- "Ways to Import Spreadsheets"

See Also

`table` | `textscan` | `writetable`

Introduced in R2013b

real

Real part of complex number

Syntax

```
X = real(Z)
```

Description

`X = real(Z)` returns the real part of the elements of the complex array `Z`.

Examples

`real(2+3*i)` is 2.

See Also

`abs` | `angle` | `conj` | `i` | `j` | `imag`

Introduced before R2006a

reallog

Natural logarithm for nonnegative real arrays

Syntax

```
Y = reallog(X)
```

Description

`Y = reallog(X)` returns the natural logarithm of each element in array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

```
M = magic(4)

M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

reallog(M)

ans =
    2.7726    0.6931    1.0986    2.5649
    1.6094    2.3979    2.3026    2.0794
    2.1972    1.9459    1.7918    2.4849
    1.3863    2.6391    2.7081         0
```

See Also

`log` | `realpow` | `realsqrt`

Introduced before R2006a

realmax

Largest positive floating-point number

Syntax

```
n = realmax
```

Description

`n = realmax` returns the largest finite floating-point number in IEEE double precision.

`realmax('double')` is the same as `realmax` with no arguments.

`realmax('single')` returns the largest finite floating-point number in IEEE single precision.

Examples

Find the value of the constant `realmax`:

```
ndouble = realmax
nsingle = realmax('single')
```

```
ndouble =

  1.7977e+308


nsingle =

  3.4028e+38
```

See Also

`eps` | `realmin` | `intmax`

Introduced before R2006a

realmin

Smallest positive normalized floating-point number

Syntax

```
n = realmin
realmin('double')
realmin('single')
```

Description

`n = realmin` returns the smallest positive normalized floating-point number in IEEE double precision.

`realmin('double')` is the same as `realmin` with no arguments.

`realmin('single')` returns the smallest positive normalized floating-point number in IEEE single precision.

Examples

Find the value of the constant `realmin`:

```
ndouble = realmin
nsingle = realmin('single')
```

```
ndouble =

  2.2251e-308


nsingle =

  1.1755e-38
```

See Also

`eps` | `realmax` | `intmin`

Introduced before R2006a

realpow

Array power for real-only output

Syntax

```
Z = realpow(X,Y)
```

Description

`Z = realpow(X,Y)` raises each element of array `X` to the power of its corresponding element in array `Y`. Arrays `X` and `Y` must be the same size. The range of `realpow` is the set of all real numbers, i.e., all elements of the output array `Z` must be real.

Examples

```
X = -2*ones(3,3)

X =
     -2     -2     -2
     -2     -2     -2
     -2     -2     -2

Y = pascal(3)

ans =
     1     1     1
     1     2     3
     1     3     6

realpow(X,Y)

ans =
    -2    -2    -2
    -2     4    -8
    -2    -8    64
```

See Also

`mpower` | `power` | `reallog` | `realsqrt`

Introduced before R2006a

realsqrt

Square root for nonnegative real arrays

Syntax

```
Y = realsqrt(X)
```

Description

`Y = realsqrt(X)` returns the square root of each element of array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

```
M = magic(4)

M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

realsqrt(M)

ans =
    4.0000    1.4142    1.7321    3.6056
    2.2361    3.3166    3.1623    2.8284
    3.0000    2.6458    2.4495    3.4641
    2.0000    3.7417    3.8730    1.0000
```

See Also

`reallog` | `sqrt` | `sqrtm` | `realpow`

Introduced before R2006a

record

Record data and event information to file

Syntax

```
record(obj)
record(obj,'switch')
```

Description

`record(obj)` toggles the recording state for the serial port object, `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Examples

This example creates the serial port object `s` on a Windows platform. It connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
s.RecordName = 'MySerialFile.txt';
record(s,'on')
fprintf(s,'*IDN?')
out = fscanf(s);
record(s,'off')
fclose(s)
```

More About

Tips

Before you can record information to disk, `obj` must be connected to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An

error is returned if you attempt to record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to Debugging: Recording Information to Disk.

See Also

`fclose` | `fopen` | `RecordDetail` | `RecordMode` | `RecordName` | `RecordStatus` | `Status`

Introduced before R2006a

record

Record audio to `audiorecorder` object

Syntax

```
record(recorderObj)
record(recorderObj, length)
```

Description

`record(recorderObj)` records audio from an input device, such as a microphone connected to your system. *`recorderObj`* is an `audiorecorder` object that defines the sample rate, bit depth, and other properties of the recording.

`record(recorderObj, length)` records for the number of seconds specified by *`length`*.

Examples

Record 5 seconds of your speech with a microphone:

```
myVoice = audiorecorder;

% Define callbacks to show when
% recording starts and completes.
myVoice.StartFcn = 'disp(''Start speaking.'')';
myVoice.StopFcn = 'disp(''End of recording.'')';

record(myVoice, 5);
```

To listen to the recording, call the `play` method:

```
play(myVoice);
```

See Also

`audiorecorder` | `getaudiodata` | `recordblocking`

How To

- “Record Audio”
- “Record or Play Audio within a Function”

recordblocking

Record audio to `audiorecorder` object, holding control until recording completes

Syntax

```
recordblocking(recorderObj, length)
```

Description

`recordblocking(recorderObj, length)` records audio from an input device, such as a microphone connected to your system, for the number of seconds specified by *`length`*. The `recordblocking` method does not return control until recording completes. *`recorderObj`* is an `audiorecorder` object that defines the sample rate, bit depth, and other properties of the recording.

Examples

Record 5 seconds of your speech with a microphone, and play it back:

```
myVoice = audiorecorder;

disp('Start speaking.');
recordblocking(myVoice, 5);
disp('End of recording. Playing back ...');

play(myVoice);
```

See Also

`audiorecorder` | `record` | `getaudiodata`

How To

- "Record Audio"

rectangle

Create 2-D rectangle object

Syntax

```
rectangle
rectangle('Position',[x,y,w,h])
rectangle('Curvature',[x,y])
rectangle('PropertyName',propertyvalue,...)
h = rectangle(...)
```

Properties

For a list of properties, see Rectangle Properties.

Description

`rectangle` draws a rectangle with `Position` `[0,0,1,1]` and `Curvature` `[0,0]` (i.e., no curvature).

`rectangle('Position',[x,y,w,h])` draws the rectangle from the point `x`,`y` and having a width of `w` and a height of `h`. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the x and y axes. You can do this with the command `axis equal` or `daspect([1,1,1])`.

`rectangle('Curvature',[x,y])` specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse.

If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`rectangle('PropertyName',propertyvalue,...)` draws the rectangle using the values for the property name/property value pairs specified and default values for all other properties. For a description of the properties, see Rectangle Properties.

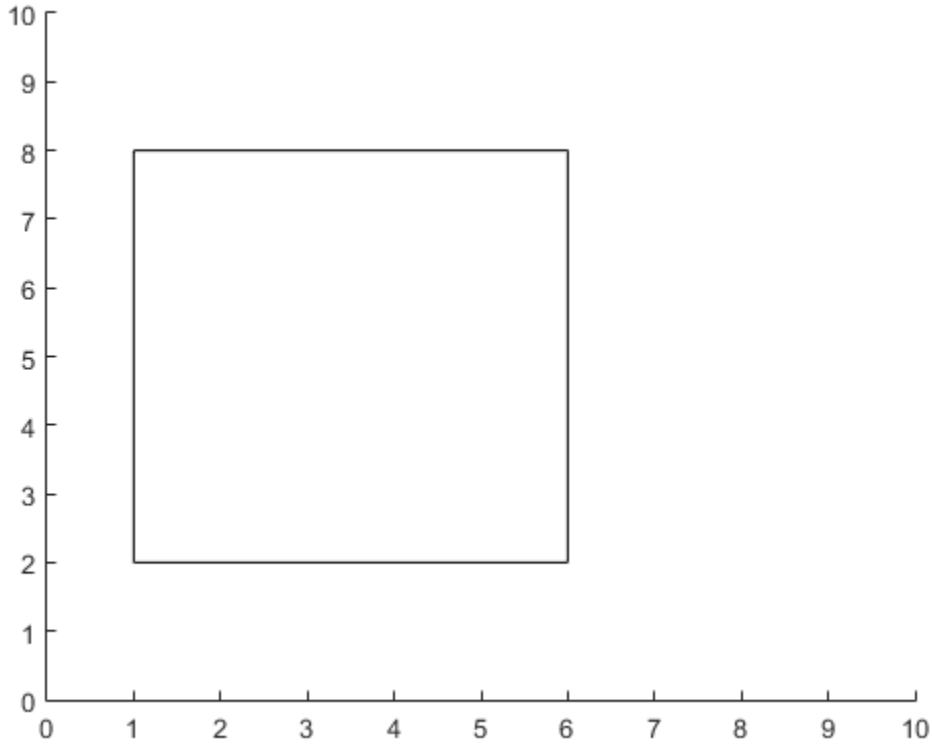
`h = rectangle(...)` returns the handle of the rectangle object created.

Examples

Draw Rectangle

Draw a rectangle that has a lower left corner at the point (1,2) and an upper right corner at (6,8). The width of the rectangle is 5 units and the height is 6 units. Additionally, change the axis limits.

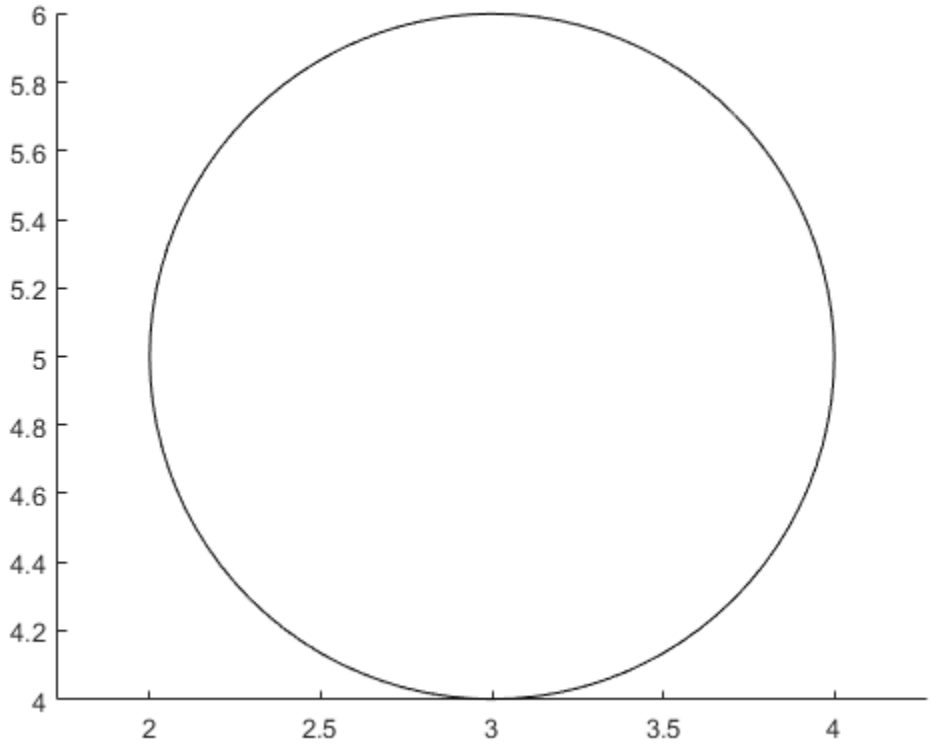
```
rectangle('Position',[1 2 5 6])
axis([0 10 0 10])
```



Draw Circle

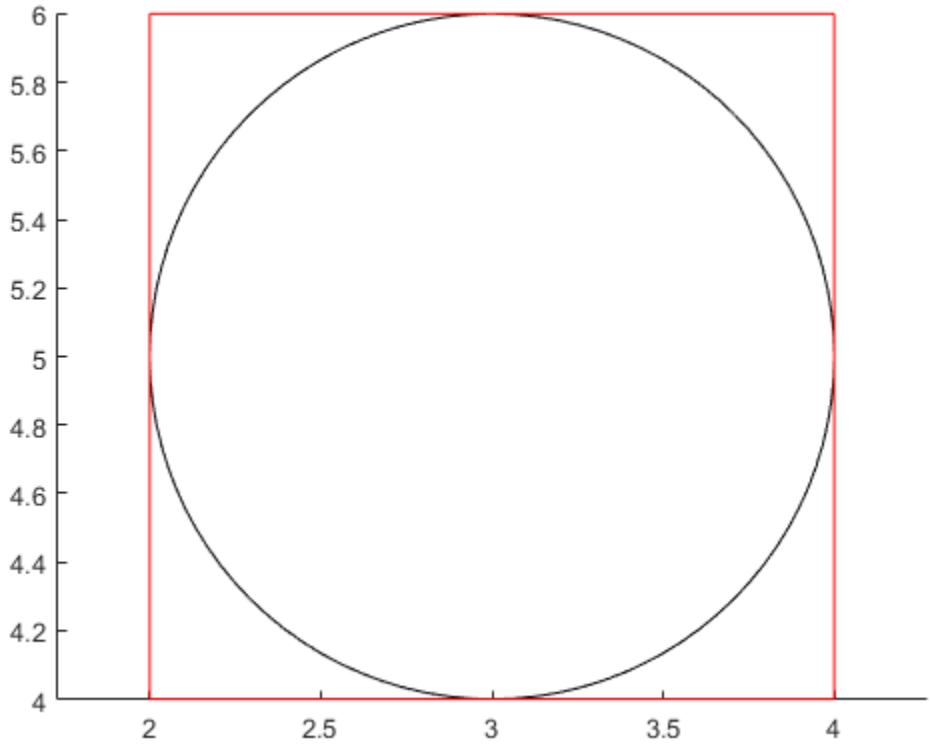
Draw a circle using the `rectangle` function and setting the `Curvature` property to `[1 1]`. Set the `Position` property to define the smallest rectangle that contains the circle. Draw the circle so that it fills the rectangular area between the points (2,4) and (4,6).

```
pos = [2 4 2 2];
rectangle('Position',pos,'Curvature',[1 1])
axis equal
```



Draw a red rectangle using the same position values to show how the circle fills the rectangular area.

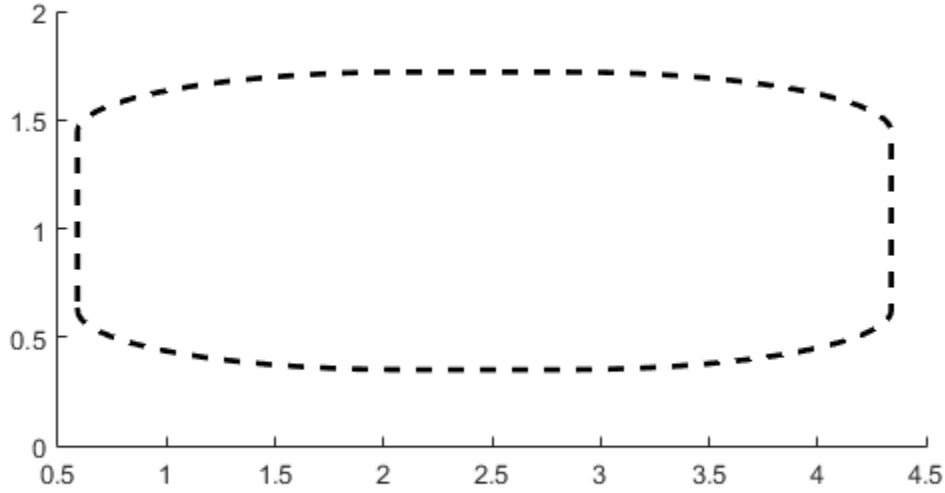
```
rectangle('Position',pos,'EdgeColor','r')
```



Rectangle with Different Curvatures

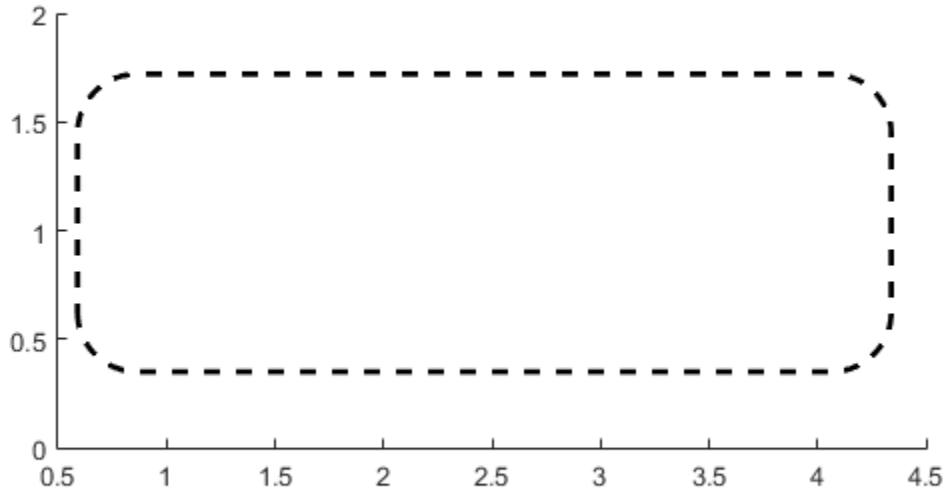
Create a rectangle with a curvature of `[0.8,0.4]`, which produces a rectangle with different horizontal and vertical curvatures. Set the data aspect ratio to `[1,1,1]` so that MATLAB® displays the rectangle in the specified proportions.

```
figure
rectangle('Position',[0.59,0.35,3.75,1.37],...
          'Curvature',[0.8,0.4],...
          'LineWidth',2,...
          'LineStyle','--')
daspect([1,1,1])
```



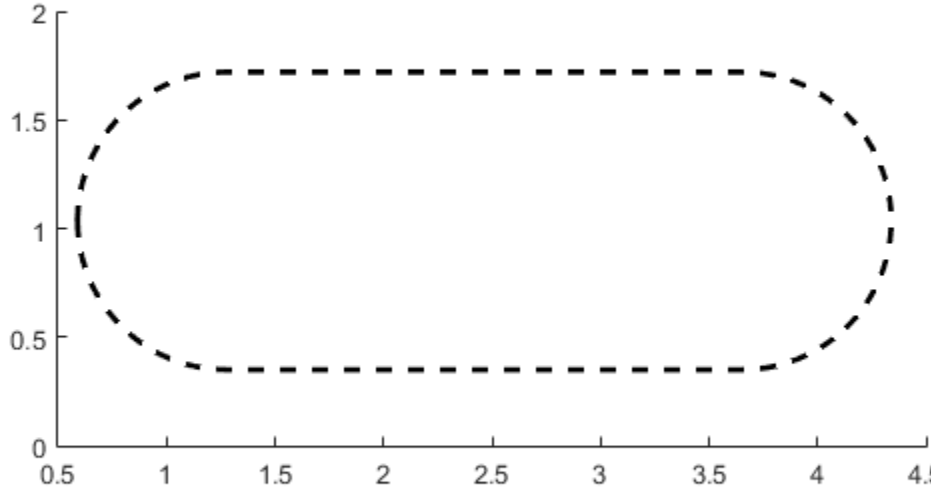
Create a rectangle with a curvature of `0.4`.

```
figure
rectangle('Position',[0.59,0.35,3.75,1.37],...
          'Curvature',0.4,...
          'LineWidth',2,...
          'LineStyle','--')
daspect([1,1,1])
```



Create a rectangle with a curvature of 1, which produces a rectangle with the shortest side completely round.

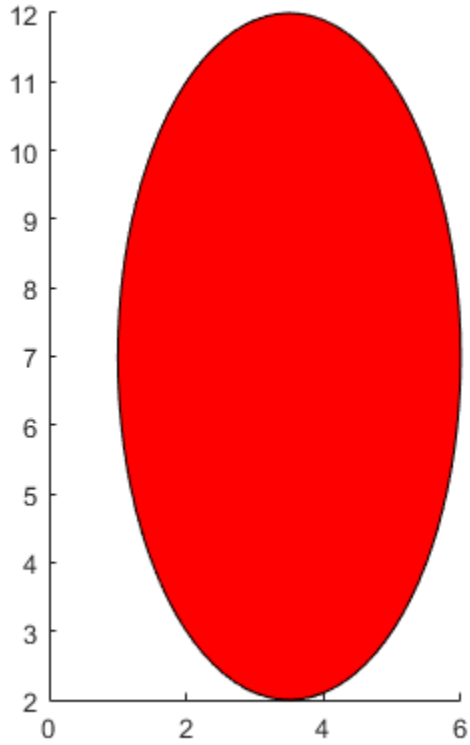
```
figure
rectangle('Position',[0.59,0.35,3.75,1.37],...
          'Curvature',1,...
          'LineWidth',2,...
          'LineStyle','--')
daspect([1,1,1])
```



Create Red Ellipse

Create an ellipse and color the face red.

```
figure
rectangle('Position',[1,2,5,10],...
    'Curvature',[1,1],...
    'FaceColor','r')
daspect([1,1,1])
```



More About

Tips

Rectangles are children of axes and are defined in coordinates of the axes data.

See Also

Functions

`annotation` | `line` | `patch`

Properties
Rectangle Properties

Introduced before R2006a

Rectangle Properties

Control rectangle appearance and behavior

Rectangle properties control the appearance and behavior of a rectangle object. By changing property values, you can modify certain aspects of the rectangle.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = rectangle;
w = h.LineWidth;
h.LineWidth = 3;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Curvature` — Amount of horizontal and vertical curvature

`[0 0]` (default) | two-element vector | scalar

Amount of horizontal and vertical curvature, specified as a two element vector of the form `[x y]` or a scalar value. Use this property to vary the shape of the rectangle from rectangular to ellipsoidal. The horizontal curvature is the fraction of the width that is curved along the top and bottom edges. The vertical curvature is the fraction of the height that is curved along the left and right edges.

- To use different horizontal and vertical curvatures, specify a two-element vector of the form `[x y]`. The `x` element determines the horizontal curvature and the `y` element determines the vertical curvature. Specify `x` and `y` as values between 0 (no curvature) and 1 (maximum curvature). For example, a value of `[0 0]` creates a rectangle with square edges and value of `[1 1]` creates an ellipse.
- To use the same curvature for the horizontal and vertical edges, specify a scalar value in the range `[0,1]`. The shorter dimension determines the length of the curvature.

Example: `[0.5 0.6]`

Example: `0.75`

`EdgeColor` — Outline color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Outline color, specified as an RGB triplet, a color string, or `'none'`. The `'none'` option makes the edge invisible. The default RGB triplet value of `[0 0 0]` corresponds to black.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`FaceColor` — Fill color

`'none'` (default) | RGB triplet | color string

Fill color, specified as `'none'`, an RGB triplet, or a color string. The `'none'` option makes the fill invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

LineStyle — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
'-'	Solid line	————
'--'	Dashed line	- - - - -
':'	Dotted line	.....
'-.'	Dash-dotted line	—·—·—·—
'none'	No line	No line

LineWidth — Line width

0.5 (default) | positive value

Line width, specified as a positive value in point units.

Example: 0.75

AlignVertexCenters — Sharp vertical and horizontal lines

'off' (default) | 'on'

Sharp vertical and horizontal lines, specified as 'off' or 'on'.

If the associated figure has a GraphicsSmoothing property set to 'on' and a Renderer property set to 'opengl', then the figure applies a smoothing technique to

plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Location and Size

`Position` — Size and location of rectangle

`[0 0 1 1]` (default) | four-element vector

Size and location of the rectangle, specified as a four-element vector of the form `[x y width height]`. Specify the values in data units. The `x` and `y` elements define the coordinate for the lower-left corner of the rectangle. The `width` and `height` elements define the dimensions of the rectangle.

Example: `[0.5 0.5 0.3 0.4]`

Visibility

`Visible` — Visibility of rectangle

`'on'` (default) | `'off'`

Visibility of rectangle, specified as one of these values:

- `'on'` — Display the rectangle.
- `'off'` — Hide the rectangle without deleting it. You still can access the properties of an invisible rectangle object.

`Clipping` — Clipping of rectangle to axes limits

`'on'` (default) | `'off'`

Clipping of rectangle to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the rectangle that are outside the axes limits.
- `'off'` — Display the entire rectangle, even if parts of it appear outside the axes limits. Parts of the rectangle might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the rectangle that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the

printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'rectangle'`

Type of graphics object, returned as `'rectangle'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Tag to associate with rectangle

`''` (default) | string

Tag to associate with the rectangle, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with rectangle

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the rectangle object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent of rectangle

axes object | group object | transform object

Parent of rectangle, specified as an axes, group, or transform object.

`Children` — Children of rectangle

empty `GraphicsPlaceholder` array

The rectangle has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of rectangle object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The rectangle object handle is always visible.
- `'off'` — The rectangle object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The rectangle object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the rectangle at the command-line, but allows callback functions to access it.

If the rectangle object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the rectangle. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The rectangle object — You can access properties of the rectangle object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the rectangle. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the rectangle when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the rectangle.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the rectangle that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The HitTest property determines if the rectangle responds to the click or if an ancestor does.
- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'` and you can click a part of the rectangle that has no color. The HitTest property determines if the rectangle responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the rectangle passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the rectangle. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the rectangle that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the rectangle object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the rectangle is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the rectangle tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the rectangle. Setting the `CreateFcn` property on an existing rectangle has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during rectangle creation.

MATLAB executes the callback after creating the rectangle and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The rectangle object — You can access properties of the rectangle object from within the callback function. You also can access the rectangle object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the rectangle. MATLAB executes the callback before destroying the rectangle so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The rectangle object — You can access properties of the rectangle object from within the callback function. You also can access the rectangle object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of rectangle

`'off'` (default) | `'on'`

Deletion status of rectangle, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the rectangle begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the rectangle no longer exists.

Check the value of the BeingDeleted property to verify that the rectangle is not about to be deleted before querying or modifying it.

See Also

`rectangle`

More About

- "Access Property Values"
- "Graphics Object Properties"

rectint

Rectangle intersection area

Syntax

```
area = rectint(A,B)
```

Description

`area = rectint(A,B)` returns the area of intersection of the rectangles specified by position vectors `A` and `B`.

If `A` and `B` each specify one rectangle, the output `area` is a scalar.

`A` and `B` can also be matrices, where each row is a position vector. `area` is then a matrix giving the intersection of all rectangles specified by `A` with all the rectangles specified by `B`. That is, if `A` is `n`-by-`4` and `B` is `m`-by-`4`, then `area` is an `n`-by-`m` matrix where `area(i,j)` is the intersection area of the rectangles specified by the `i`th row of `A` and the `j`th row of `B`.

Note A position vector is a four-element vector `[x,y,width,height]`, where the point defined by `x` and `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the x and y axes respectively.

See Also

`polyarea`

Introduced before R2006a

recycle

Set option to move deleted files to recycle folder

Syntax

```
status = recycle
oldState = recycle(state)
```

Description

`status = recycle` returns the current state for recycling files you remove using the `delete` function. When `status` is `off`, the `delete` function permanently removes the files. When `status` is `on`, deleted files move to a different location. For details, see the Tips section.

`oldState = recycle(state)` sets the recycle option for MATLAB to the specified state, either `on` or `off`. The `previousStat` value is the recycle state before running the statement.

Examples

View Current Recycling State

Start from a state where file recycling is off. Verify the current recycle state.

```
state = recycle
```

```
state =

off
```

Turn File Recycling On

Turn file recycling on. Then, delete an existing file and move it to the recycle bin or temporary folder.

```
recycle('on');
```

```
delete('myfile.txt')
```

Input Arguments

`state` — State of recycle option

`'on' | 'off'`

State of the recycle option, specified as `'on'` or `'off'`.

More About

Tips

- The location for storing recycled files varies by platform, as follows:
  - Microsoft Windows systems — Recycle bin.
  - Apple Macintosh systems — Trash.
  - Linux systems — Subfolder with the prefix `MATLAB_Files_` in the system temporary folder, as returned by the `tempdir` function.
- The general preference for **Deleting files** sets the state of the `recycle` function at startup. When you change the preference, it changes the state of `recycle`. However, when you change the state of `recycle`, it does not change the preference.
- "General Preferences"

See Also

`delete` | `dir` | `ls` | `rmdir`

Introduced before R2006a

reducepatch

Reduce number of patch faces

Syntax

```
reducepatch(p,r)
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
nfv = reducepatch(p)
nfv = reducepatch(fv)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)
```

Description

`reducepatch(p,r)` reduces the number of faces of the patch identified by handle `p`, while attempting to preserve the overall shape of the original object. The MATLAB software interprets the reduction factor `r` in one of two ways depending on its value:

- If `r` is less than 1, `r` is interpreted as a fraction of the original number of faces. For example, if you specify `r` as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If `r` is greater than or equal to 1, then `r` is the target number of faces. For example, if you specify `r` as 400, then the number of faces is reduced until there are 400 faces remaining.

`nfv = reducepatch(p,r)` returns the reduced set of faces and vertices but does not set the `Faces` and `Vertices` properties of patch `p`. The struct `nfv` contains the faces and vertices after reduction.

`nfv = reducepatch(fv,r)` performs the reduction on the faces and vertices in the struct `fv`.

`nfv = reducepatch(p)` and `nfv = reducepatch(fv)` uses a reduction value of `0.5`.

`reducepatch(...,'fast')` assumes the vertices are unique and does not compute shared vertices.

`reducepatch(...,'verbose')` prints progress messages to the command window as the computation progresses.

`nfv = reducepatch(f,v,r)` performs the reduction on the faces in `f` and the vertices in `v`.

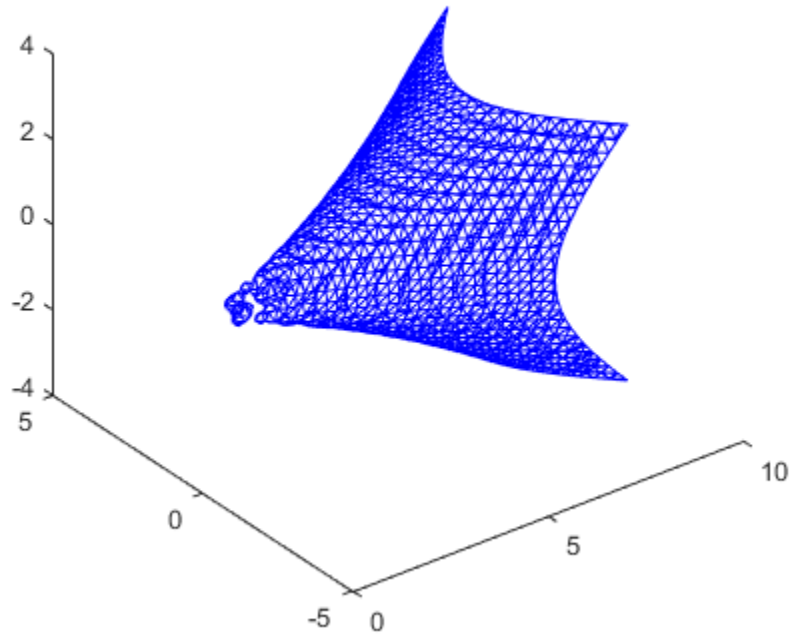
`[nf,nv] = reducepatch(...)` returns the faces and vertices in the arrays `nf` and `nv`.

Examples

Reduce Number of Patch Faces

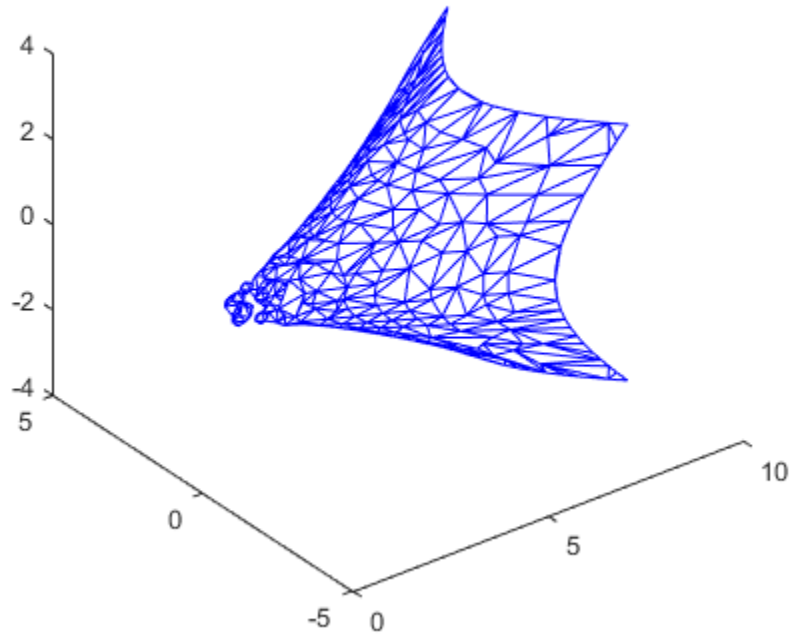
This example illustrates the effect of reducing the number of faces to only 15% of the original value.

```
figure
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
p.FaceColor = 'w';
p.EdgeColor = 'b';
daspect([1,1,1])
view(3)
```



Reduce the number of faces.

```
reducepatch(p,0.15)
```



More About

Tips

If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

The number of output triangles may not be exactly the number specified with the reduction factor argument (`r`), particularly if the faces of the original patch are not triangles.

- Vector Field Displayed with Cone Plots

See Also

`isosurface` | `isocaps` | `isonormals` | `smooth3` | `subvolume` | `reducevolume`

Introduced before R2006a

reducevolume

Reduce number of elements in volume data set

Syntax

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])
[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])
nv = reducevolume(...)
```

Description

`[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])` reduces the number of elements in the volume by retaining every Rx^{th} element in the *x* direction, every Ry^{th} element in the *y* direction, and every Rz^{th} element in the *z* direction. If a scalar `R` is used to indicate the amount or reduction instead of a three-element vector, the MATLAB software assumes the reduction to be `[R R R]`.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The reduced volume is returned in `nv`, and the coordinates of the reduced volume are returned in `nx`, `ny`, and `nz`.

`[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p] = size(V)`.

`nv = reducevolume(...)` returns only the reduced volume.

Examples

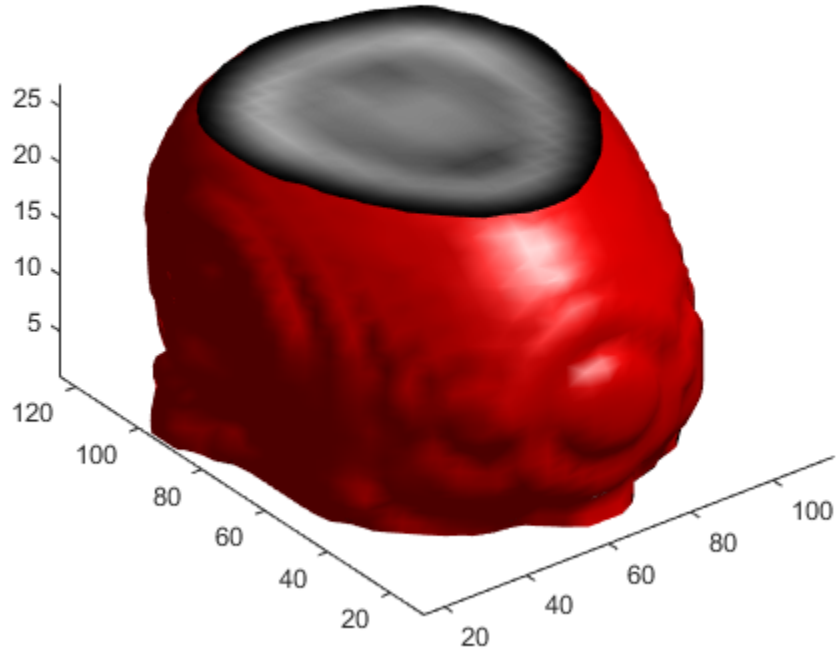
Reduce Volume Data Set

This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every fourth element in the x and y directions and every element in the z direction.

- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`) isocaps).
- The view of the object is set (`view`, `axis`, `daspect`).
- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).
- Adding a light to the right of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = reducevolume(D,[4,4,1]);
D = smooth3(D);
p1 = patch(isosurface(x,y,z,D,5),...
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1)
p2 = patch(isocaps(x,y,z,D,5),...
    'FaceColor','interp','EdgeColor','none');
view(3)
axis tight
daspect([1,1,.4])
colormap(gray(100))
camlight
lighting gouraud
```



See Also

`isosurface` | `isocaps` | `isonormals` | `smooth3` | `subvolume` | `reducepatch`

Introduced before R2006a

refresh

Redraw current figure

Syntax

```
refresh
refresh(h)
```

Description

`refresh` erases and redraws the current figure.

`refresh(h)` redraws the figure identified by `h`.

Introduced before R2006a

refreshdata

Refresh data in graph when data source is specified

Syntax

```
refreshdata
refreshdata(figure_handle)
refreshdata(object_handles)
refreshdata(object_handles,'workspace')
```

Description

`refreshdata` evaluates any data source properties (`XDataSource`, `YDataSource`, or `ZDataSource`) on all objects in graphs in the current figure. If the specified data source has changed, the MATLAB software updates the graph to reflect this change.

Note: The variable assigned to the data source property must be in the base workspace or you must specify the *`workspace`* option as `'caller'`.

`refreshdata(figure_handle)` refreshes the data of the objects in the specified figure.

`refreshdata(object_handles)` refreshes the data of the objects specified in `object_handles` or the children of those objects. Therefore, `object_handles` can contain figure, axes, or plot object handles.

`refreshdata(object_handles,'workspace')` enables you to specify whether the data source properties are evaluated in the base workspace or the workspace of the function in which `refreshdata` was called. *`workspace`* is a string that can be

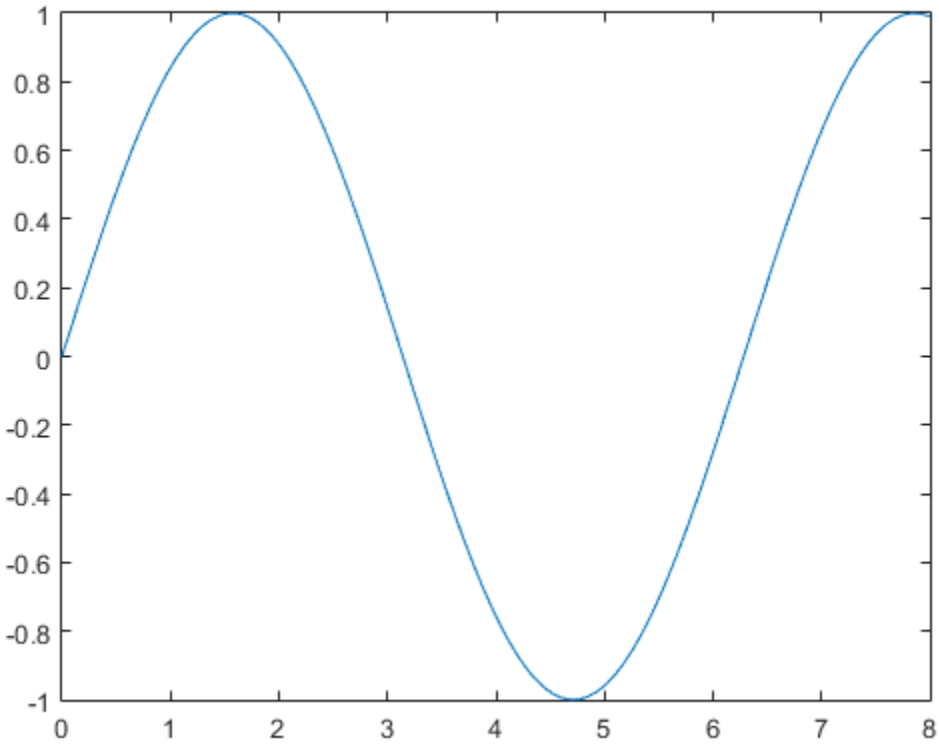
- `base` — Evaluate the data source properties in the base workspace.
- `caller` — Evaluate the data source properties in the workspace of the function that called refreshdata.

Examples

Refresh Graph with Updated Data

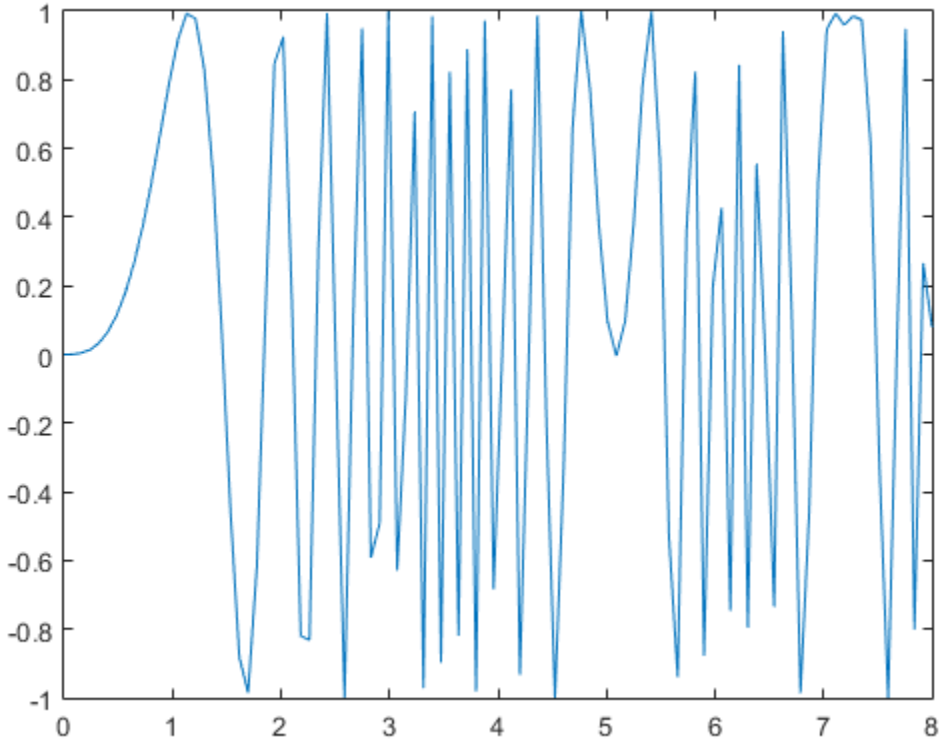
Plot a sine wave and return the chart line handle, `h`.

```
x = linspace(0,8);
y = sin(x);
figure
h = plot(x,y);
```



Identify the data sources for the plot by setting the `XDataSource` and `YDataSource` properties of the line to `x` and `y`, respectively. Then, modify `y`. Call `refreshdata` so that the graph updates with the changes to `y`.

```
h.XDataSource = 'x';
h.YDataSource = 'y';

y = sin(x.^3);
refreshdata
```



More About

Tips

The Linked Plots feature (see documentation for `linkdata`) sets up data sources for graphs and synchronizes them with the workspace variables they display. When you use

this feature, you do not also need to call `refreshdata`, as it is essentially automatically triggered every time a data source changes.

If you are not using the Linked Plots feature, you need to set the `XDataSource`, `YDataSource`, and/or `ZDataSource` properties of a graph in order to use `refreshdata`. You can do that programmatically, as shown in the examples below, or use the Property Editor, one of the plotting tools. In the Property Editor, select the graph (e.g., a lineseries object) and type in (or select from the drop-down choices) the name(s) of the workspace variable(s) from which you want the plot to refresh, in the fields labelled **X Data Source**, **Y Data Source**, and/or **Z Data Source**. The call to `refreshdata` causes the graph to update.

See Also

`linkprop` | `linkaxes`

Introduced before R2006a

regexp

Match regular expression (case sensitive)

Syntax

```
startIndex = regexp(str,expression)
[startIndex,endIndex] = regexp(str,expression)

out = regexp(str,expression,outkey)
[out1,...,outN] = regexp(str,expression,outkey1,...,outkeyN)

___ = regexp( ___ ,option1,...,optionM)
```

Description

`startIndex = regexp(str,expression)` returns the starting index of each substring of `str` that matches the character patterns specified by the regular expression. If there are no matches, `startIndex` is an empty array.

`[startIndex,endIndex] = regexp(str,expression)` returns the starting and ending indices of all matches.

`out = regexp(str,expression,outkey)` returns the output specified by outkey. For example, if `outkey` is `'match'`, then `regexp` returns the substrings that match the expression rather than their starting indices.

`[out1,...,outN] = regexp(str,expression,outkey1,...,outkeyN)` returns the outputs specified by multiple output keywords, in the specified order. For example, if you specify `'match'`,`'tokens'`, then `regexp` returns substrings that match the entire expression and tokens that match parts of the expression.

`___ = regexp( ___ ,option1,...,optionM)` modifies the search using the specified option flags. For example, specify `'ignorecase'` to perform a case-insensitive match. You can include any of the inputs and request any of the outputs from previous syntaxes.

Examples

Find Patterns in Single Strings

Find words that start with `c`, end with `t`, and contain one or more vowels between them.

```
str = 'bat cat can car coat court CUT ct CAT-scan';
expression = 'c[aeiou]+t';
startIndex = regexp(str,expression)
```

```
startIndex =

     5    17
```

The regular expression `'c[aeiou]+t'` specifies this pattern:

- `c` must be the first character.
- `c` must be followed by one of the characters inside the brackets, `[aeiou]`.
- The bracketed pattern must occur one or more times, as indicated by the `+` operator.
- `t` must be the last character, with no characters between the bracketed pattern and the `t`.

Values in `startIndex` indicate the index of the first character of each word that matches the regular expression. The matching word `cat` starts at index 5, and `coat` starts at index 17. The words `CUT` and `CAT` do not match because they are uppercase.

Find Patterns in Multiple Strings

Find the location of capital letters and spaces within strings in a cell array.

```
str = {'Madrid, Spain','Romeo and Juliet','MATLAB is great'};
capExpr = '[A-Z]';
spaceExpr = '\s';

capStartIndex = regexp(str,capExpr);
spaceStartIndex = regexp(str,spaceExpr);
```

`capStartIndex` and `spaceStartIndex` are cell arrays because the input `str` is a cell array.

View the indices for the capital letters.

```
celldisp(capStartIndex)
```

```
capStartIndex{1} =

     1     9


capStartIndex{2} =

     1    11


capStartIndex{3} =

     1     2     3     4     5     6
```

View the indices for the spaces.

```
celldisp(spaceStartIndex)
```

```
spaceStartIndex{1} =

     8


spaceStartIndex{2} =

     6    10


spaceStartIndex{3} =

     7    10
```

Return Substrings Using match Keyword

Capture words within a string that contain the letter `x`.

```
str = 'EXTRA! The regexp function helps you relax.';
expression = '\w*x\w*';
matchStr = regexp(str,expression,'match')
```

```
matchStr =

    'regexp'    'relax'
```

The regular expression `'\w*x\w*'` specifies that the string:

- Begins with any number of alphanumeric or underscore characters, `\w*`.
- Contains the lowercase letter `x`.
- Ends with any number of alphanumeric or underscore characters after the `x`, including none, as indicated by `\w*`.

Split String at Delimiter Using split Keyword

Split a string into several substrings, where each substring is delimited by a `^` character.

```
str = ['Split ^this string into ^several pieces'];
expression = '\^';
splitStr = regexp(str,expression,'split')
```

```
splitStr =

    'Split '    'this string into '    'several pieces'
```

Because the caret symbol has special meaning in regular expressions, precede it with the escape character, a backslash (\). To split a string at other delimiters, such as a semicolon, you do not need to include the backslash.

Return Both Matching and Nonmatching Substrings

Capture parts of a string that match a regular expression using the `'match'` keyword, and the remaining parts that do not match using the `'split'` keyword.

```
str = 'She sells sea shells by the seashore.';
expression = '[Ss]h.';
[match,noMatch] = regexp(str,expression,'match','split')
```

```
match =

    'She'    'she'    'sho'


noMatch =

    ''    ' sells sea '    'lls by the sea'    're.'
```

The regular expression `'[Ss]h.'` specifies that:

- `S` or `s` is the first character.
- `h` is the second character.
- The third character can be anything, including a space, as indicated by the dot (`.`).

When the first (or last) character in a string matches a regular expression, the first (or last) return value from the `'split'` keyword is an empty string.

Optionally, reassemble the original string from the substrings.

```
combinedStr = strjoin(noMatch,match)
```

```
combinedStr =

She sells sea shells by the seashore.
```

Capture Substrings of Matches Using Ordinal Tokens

Find the names of HTML tags by defining a token within a regular expression. Tokens are indicated with parentheses, `()`.

```
str = '<title>My Title</title><p>Here is some text.</p>';
expression = '<(\w+).*>.*</\1>';
[tokens,matches] = regexp(str,expression,'tokens','match');
```

The regular expression `<(\w+).*>.*</\1>` specifies this pattern:

- `<(\w+)` finds an opening angle bracket followed by one or more alphanumeric or underscore characters. Enclosing `\w+` in parentheses captures the name of the HTML tag in a token.
- `.*>` finds any number of additional characters, such as HTML attributes, and a closing angle bracket.
- `</\1>` finds the end tag corresponding to the first token (indicated by `\1`). The end tag has the form `<\tagname>`.

View the tokens and matching substrings.

```
celldisp(tokens)
```

```
tokens{1}{1} =
   title

tokens{2}{1} =
   p
```

```
celldisp(matches)
```

```
matches{1} =
   <title>My Title</title>

matches{2} =
   <p>Here is some text.</p>
```

Capture Substrings of Matches Using Named Tokens

Parse dates that can appear with either the day or the month first, in these forms: `mm/dd/yyyy` or `dd-mm-yyyy`. Use named tokens to identify each part of the date.

```
str = '01/11/2000  20-02-2020  03/30/2000  16-04-2020';
```

```
expression = ['(?<month>\d+)/(?<day>\d+)/(?<year>\d+)|'...
              '(?<day>\d+)-(?<month>\d+)-(?<year>\d+)'];
tokenNames = regexp(str,expression,'names');
```

The regular expression specifies this pattern:

- `(?<name>\d+)` finds one or more numeric digits and assigns the result to the token indicated by name.
- `|` is the logical `or` operator, which indicates that there are two possible patterns for dates. In the first pattern, slashes (`/`) separate the tokens. In the second pattern, hyphens (`-`) separate the tokens.

View the named tokens.

```
for k = 1:length(tokenNames)
   disp(tokenNames(k))
end
```

```
    month: '01'
      day: '11'
     year: '2000'

    month: '02'
      day: '20'
     year: '2020'

    month: '03'
      day: '30'
     year: '2000'

    month: '04'
      day: '16'
     year: '2020'
```

Perform Case-Insensitive Matches

Find both uppercase and lowercase instances of a word.

By default, `regexp` performs case-sensitive matching.

```
str = 'A string with UPPERCASE and lowercase text.';
expression = '\w*case';
matchStr = regexp(str,expression,'match')
```

```
matchStr =

    'lowercase'
```

The regular expression specifies that the string:

- Begins with any number of alphanumeric or underscore characters, `\w*`.
- Ends with the literal text `case`.

The `regexpi` function uses the same syntax as `regexp`, but performs case-insensitive matching.

```
matchWithRegexpi = regexpi(str,expression,'match')
```

```
matchWithRegexpi =

    'UPPERCASE'    'lowercase'
```

Alternatively, disable case-sensitive matching for `regexp` using the `'ignorecase'` option.

```
matchWithIgnorecase = regexp(str,expression,'match','ignorecase')
```

```
matchWithIgnorecase =

    'UPPERCASE'    'lowercase'
```

For multiple expressions, disable case-sensitive matching for selected expressions using the `(?i)` search flag.

```
expression = {'(?-i)\w*case';...
              '(?i)\w*case'};
matchStr = regexp(str,expression,'match');
celldisp(matchStr)
```

```
matchStr{1}{1} =

lowercase
```

```
matchStr{2}{1} =

UPPERCASE


matchStr{2}{2} =

lowercase
```

Parse Strings with Newline Characters

Create a string that contains a newline, `\n`, and parse the string using a regular expression.

```
str = sprintf('abc\n de');
expression = '.*';
matchStr = regexp(str,expression,'match')
```

```
matchStr =

    [1x7 char]
```

By default, the dot (`.`) matches every character, including the newline, and returns a single match that is equivalent to the original string.

Exclude newline characters from the match using the `'dotexceptnewline'` option. This returns separate matches for each line of text.

```
matchStrNoNewline = regexp(str,expression,'match','dotexceptnewline')
```

```
matchStrNoNewline =

    'abc'    ' de'
```

Find the first or last character of each line using the `^` or `$` metacharacters and the `'lineanchors'` option.

```
expression = '.$';
lastInLine = regexp(str,expression,'match','lineanchors')
```

```
lastInLine =

    'c'    'e'
```

Input Arguments

`str` — Input text

string | cell array of strings

Input text, specified as a string or a cell array of strings. Each string can be of any length and contain any characters.

If str and expression are both cell arrays, they must have the same dimensions.

Data Types: `char` | `cell`

`expression` — Regular expression

string | cell array of strings

Regular expression, specified as a string or a cell array of strings. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in str.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
`[`$c_1c_2c_3$`]`	Any character contained within the brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[rp.]ain'` matches `'rain'` or `'pain'` or ‘`.ain`’.

Metacharacter	Description	Example
`[^c1c2c3]`	Any character not contained within the brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[^*rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'` and `'*ain'`. For example, it matches `'gain'`, `'lain'`, or `'vain'`.
`[c1-c2]`	Any character in the range of `c1` through `c2`	`'[A-G]'` matches a single character in the range of `A` through `G`.
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, `\w` is equivalent to `[a-zA-Z_0-9]`	`'\w*'` identifies a word.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, `\W` is equivalent to `[^a-zA-Z_0-9]`	`'\W*'` identifies a substring that is not a word.
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`	`'\w*n\s'` matches words that end with the letter `n`, followed by a white-space character.
`\S`	Any non-white-space character; equivalent to `[^ \f\n\r\t\v]`	`'\d\S'` matches a numeric digit followed by any non-white-space character.
`\d`	Any numeric digit; equivalent to `[0-9]`	`'\d*'` matches any number of consecutive digits.
`\D`	Any nondigit character; equivalent to `[^0-9]`	`'\w*\D\>'` matches words that do not end with a numeric digit.
`\oN` or `\o{N}`	Character of octal value `N`	`'\o{40}'` matches the space character, defined by octal `40`.
`\xN` or `\x{N}`	Character of hexadecimal value `N`	`'\x2C'` matches the comma character, defined by hex `2C`.

Character Representation

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace

Operator	Description
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\char`	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Quantifiers

Quantifiers specify the number of times a string pattern must occur in the matching string.

Quantifier	Matches the expression when it occurs...	Example
`expr*`	0 or more times consecutively.	`'\w*'` matches a word of any length.
`expr?`	0 times or 1 time.	`'\w*(\.m)?'` matches words that optionally end with the extension `.m`.
`expr+`	1 or more times consecutively.	`'<img src="\w+\.gif">'` matches an `<img>` HTML tag when the file name contains one or more characters.
`expr{m,n}`	At least `m` times, but no more than `n` times consecutively. `{0,1}` is equivalent to `?`.	`'\S{4,8}'` matches between four and eight non-white-space characters.
`expr{m,}`	At least `m` times consecutively. `{0,}` and `{1,}` are equivalent to `*` and `+`, respectively.	`'<a href="\w{1,}\.html">'` matches an `<a>` HTML tag when the file name contains one or more characters.
`expr{n}`	Exactly `n` times consecutively. Equivalent to `{n,n}`.	`'\d{4}'` matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
expr*q*	Greedy expression: match as many characters as possible.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*>'` matches all characters between `<tr` and `/td>`: `'<tr><td><p>text</p></td>'`
expr*q*?	Lazy expression: match as few characters as necessary.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*?>'` ends each match at the first occurrence of the closing bracket (`>`): `'<tr>'  '<td>'  '</td>'`
expr*q*+	Possessive expression: match as much as possible, but do not rescan any portions of the string.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*+>'` does not return any matches, because the closing bracket is captured using `.*`, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
`(expr)`	Group elements of the expression and capture tokens.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`(?:expr)`	Group, but do not capture tokens.	`'(?:[aeiou][^aeiou]){2}'` matches two consecutive patterns of a vowel followed by a nonvowel, such as `'anon'`. Without grouping, `'[aeiou][^aeiou]{2}'` matches a vowel followed by two nonvowels.
`(?>expr)`	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	`'A(?>.*)Z'` does not match `'AtoZ'`, although `'A(?:.*)Z'` does. Using the

Grouping Operator	Description	Example
		atomic group, `Z` is captured using `.*` and is not rescanned.
`(expr1|expr2)`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored. You can include `?:` or `?>` after the opening parenthesis to suppress tokens or group atomically.	`'(let|tel)\w+'` matches words in a string that start with `let` or `tel`.

Anchors

Anchors in the expression match the beginning or end of the string or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input string.	`'^M\w*'` matches a word starting with `M` at the beginning of the string.
`expr$`	End of the input string.	`'\w*m$'` matches words ending with `m` at the end of the string.
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for string patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches strings that are followed by `ing`, such as `'Fly'` and `'fall'` in the input string `'Flying, not falling.'`
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.
`(?<=test)expr`	Look behind for characters that match `test`.	`'(?<=re)\w*'` matches strings that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input string `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which string, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

Conditional Operator	Description	Example
`expr1|expr2`	Match expression `expr1` or expression `expr2`.	`'(let|tel)\w+'` matches words in a string that start with `let` or `tel`.

Conditional Operator	Description	Example
	If there is a match with `expr1`, then `expr2` is ignored.	
`(?(cond)expr)`	If condition `cond` is `true`, then match `expr`.	`'(?(?@ispc)[A-Z]:\\)'` matches a drive name, such as `C:\`, when run on a Windows system.
`(?(cond)expr1|expr2)`	If condition `cond` is `true`, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the string (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
`(expr)`	Capture in a token the characters that match the enclosed expression.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`\N`	Match the `N`th token.	`'<(\w+).*>.*</\1>'` captures tokens for HTML tags, such as `'title'` from the string `'<title>Some text</title>'`.
`(?(N)expr1|expr2)`	If the `N`th token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Named Token Operator	Description	Example
`(?<name>expr)`	Capture in a named token the characters that match the enclosed expression.	`'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)'` creates named tokens for the month, day, and year in an

Named Token Operator	Description	Example
		input date string of the form `mm-dd-yy`.
`\k<name>`	Match the token referred to by `name`.	`'<(?<tag>\w+).*>.*</\k<tag>>'` captures tokens for HTML tags, such as `'title'` from the string `'<title>Some text</title>'`.
`(?(name)expr1|expr2)`	If the named token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Note: If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting string in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the string. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.

Operator	Description	Example
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement strings.

Replacement String Operator	Description
`$&` or `$0`	Portion of the input string that is currently a match
`$``	Portion of the input string that precedes the current match
`$'`	Portion of the input string that follows the current match (use `$''` to represent the string `$'`)
`$N`	`N`th token
`$<name>`	Named token
`${cmd}`	String returned when MATLAB executes the command, `cmd`

Comments

Characters	Description	Example
`(?#comment)`	Insert a comment in the regular expression. The comment text is ignored when matching the input string.	`'(?# Initial digit)\<\d\w+'` includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an option input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).

Flag	Description
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern string with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of a string (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\ '` and `'\#'` to match space and # characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types: `char | cell`

`outkey` — Keyword that indicates which outputs to return

`'start' | 'end' | 'tokenExtents' | 'match' | 'tokens' | 'names' | 'split'`

Keyword that indicates which outputs to return, specified as one of the following strings.

Output Keyword	Returns
`'start'`	Starting indices of all matches, startIndex
`'end'`	Ending indices of all matches, endIndex
`'tokenExtents'`	Starting and ending indices of all tokens
`'match'`	Text of each substring that matches the pattern in expression
`'tokens'`	Text of each captured token in str
`'names'`	Name and text of each named token

Output Keyword	Returns
`'split'`	Text of nonmatching substrings of str

Data Types: `char`

`option` — Search option

`'once'` | `'warnings'` | `'ignorecase'` | `'emptymatch'` | `'dotexceptnewline'` | `'lineanchors'` | ...

Search option, specified as a string. Options come in pairs: one option that corresponds to the default behavior, and one option that allows you to override the default. Specify only one option from a pair. Options can appear in any order.

Default	Override	Description
`'all'`	`'once'`	Match the expression as many times as possible (default), or only once.
`'nowarnings'`	`'warnings'`	Suppress warnings (default), or display them.
`'matchcase'`	`'ignorecase'`	Match letter case (default), or ignore case.
`'noemptymatch'`	`'emptymatch'`	Ignore zero length matches (default), or include them.
`'dotall'`	`'dotexceptnewline'`	Match dot with any character (default), or all except newline (`\n`).
`'stringanchors'`	`'lineanchors'`	Apply `^` and `$` metacharacters to the beginning and end of a string (default), or to the beginning and end of a line.
`'literalspacing'`	`'freespacing'`	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and # characters.

Data Types: `char`

Output Arguments

`startIndex` — Starting index of each match

row vector | cell array of row vectors

Starting indices of each match, returned as a row vector or cell array, as follows:

- If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`endIndex` — Ending index of each match

row vector | cell array of row vectors

Ending index of each match, returned as a row vector or cell array, as follows:

- If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`out` — Information about matches

numeric array | cell array | structure array

Information about matches, returned as a numeric, cell, or structure array. The information in the output depends upon the value you specify for outkey, as follows.

Output Keyword	Output Description	Output Type and Dimensions
`'start'`	Starting indices of matches	For both `'start'` and `'end'`: • If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array). • If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.
`'end'`	Ending indices of matches	

Output Keyword	Output Description	Output Type and Dimensions
`'tokenExtents'`	Starting and ending indices of all tokens	By default, when returning all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array, where `n` is the number of matches. Each cell contains an `m`-by-2 numeric array of indices, where `m` is the number of tokens in the match. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array, where each inner cell contains an `m`-by-2 numeric array. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either an `m`-by-2 numeric array or a cell array with the same dimensions as `str` and/or `expression`. If a token is expected at a particular index `N`, but is not found, then MATLAB returns extents for that token of `[N,N-1]`.

Output Keyword	Output Description	Output Type and Dimensions
`'match'`	Text of each substring that matches the pattern in expression	By default, when returning all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array of strings, where `n` is the number of matches. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array of strings. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either a string or a cell array of strings with the same dimensions as `str` and `expression`.

Output Keyword	Output Description	Output Type and Dimensions
`'tokens'`	Text of each captured token in str	By default, when returning all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array, where `n` is the number of matches. Each cell contains a 1-by-`m` cell array of strings, where `m` is the number of tokens in the match. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array, where each inner cell contains a 1-by-`m` cell array of strings. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is a 1-by-`m` cell array of strings or a cell array that has the same dimensions as `str` and/or `expression`. If a token is expected at a particular index, but is not found, then MATLAB returns an empty string for the token, `''`.
`'names'`	Name and text of each named token	For all matches: • If str and expression are both strings, the output is a 1-by-`n` structure array, where `n` is the number of matches. The structure field names correspond to the token names. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` structure array. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

Output Keyword	Output Description	Output Type and Dimensions
`'split'`	Text of nonmatching substrings of str	For all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array of strings, where `n` is the number of nonmatching strings. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array of strings. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

More About

Tokens

Tokens are portions of the matched text that correspond to portions of the regular expression. To create tokens, enclose part of the regular expression in parentheses.

For example, this expression finds a date of the form `dd-mmm-yyyy`, including tokens for the day, month, and year.

```
str = 'Here is a date: 01-Apr-2020';
expression = '(\d+)-(\w+)-(\d+)';

mydate = regexp(str,expression,'tokens');
mydate{:}
```

```
ans =
    '01'    'April'    '2020'
```

You can associate names with tokens so that they are more easily identifiable:

```
str = 'Here is a date: 01-Apr-2020';
expression = '(?<day>\d+)-(?<month>\w+)-(?<year>\d+)';

mydate = regexp(str,expression,'names')
```

```
mydate =
```

```
      day: '01'
    month: 'Apr'
     year: '2020'
```

For more information, see “Tokens in Regular Expressions”.

Tips

- Use `strfind` to find an exact character match within a string. Use `regexp` to look for a pattern of characters.

Algorithms

MATLAB parses each input string from left to right, attempting to match the text in the string with the first element of the regular expression. During this process, MATLAB skips over any text that does not match.

When MATLAB finds the first match, it continues parsing the string to match the second piece of the expression, and so on.

- “Regular Expressions”
- “Lookahead Assertions in Regular Expressions”
- “Dynamic Regular Expressions”

See Also

`regexpi` | `regexprep` | `regexptranslate` | `strfind` | `strjoin` | `strrep` | `strsplit`

Introduced before R2006a

regexpi

Match regular expression (case insensitive)

Syntax

```
startIndex = regexpi(str,expression)
[startIndex,endIndex] = regexpi(str,expression)

out = regexpi(str,expression,outkey)
[out1,...,outN] = regexpi(str,expression,outkey1,...,outkeyN)

___ = regexpi( ___ ,option1,...,optionM)
```

Description

`startIndex = regexpi(str,expression)` returns the starting index of each substring of `str` that matches the character patterns specified by the regular expression, without regard to letter case. If there are no matches, `startIndex` is an empty array.

`[startIndex,endIndex] = regexpi(str,expression)` returns the starting and ending indices of all matches.

`out = regexpi(str,expression,outkey)` returns the output specified by outkey. For example, if `outkey` is `'match'`, then `regexpi` returns the substrings that match the expression rather than their starting indices.

`[out1,...,outN] = regexpi(str,expression,outkey1,...,outkeyN)` returns the outputs specified by multiple output keywords, in the specified order. For example, if you specify `'match'`,`'tokens'`, then `regexpi` returns substrings that match the entire expression and tokens that match parts of the expression.

`___ = regexpi( ___ ,option1,...,optionM)` modifies the search using the specified option flags. For example, specify `'matchcase'` to perform a case-sensitive match. You can include any of the inputs and request any of the outputs from previous syntaxes.

Examples

Pattern Matching

Find words that start with `c`, end with `t`, and contain one or more vowels between them.

```
str = 'bat cat can car COAT court cut ct CAT-scan';
expression = 'c[aeiou]+t';
startIndex = regexpi(str,expression)
```

```
startIndex =
     5    17    28    35
```

Values in `startIndex` indicate the index of the first character of each word that matches the regular expression.

The regular expression `'c[aeiou]+t'` specifies this pattern:

- `c` must be the first character.
- `c` must be followed by one of the characters inside the brackets, `[aeiou]`.
- The bracketed pattern must occur one or more times, as indicated by the `+` operator.
- `t` must be the last character, with no characters between the bracketed pattern and the `t`.

Case-Sensitive Match

Match letter case in all or part of an expression.

By default, `regexpi` performs case-insensitive matching.

```
str = 'A string with UPPERCASE and lowercase text.';
expression = '\w*case';
matchStr = regexpi(str,expression,'match')
```

```
matchStr =
    'UPPERCASE'    'lowercase'
```

Use the `regexp` function with the same syntax as `regexpi` to perform case-sensitive matching.

```
matchWithRegexp = regexp(str,expression,'match')
```

```
matchWithRegexp =
    'lowercase'
```

To disable case-sensitive matching for `regexp`, use the `'ignorecase'` option.

```
matchWithIgnorecase = regexp(str,expression,'match','ignorecase')
```

```
matchWithIgnorecase =
    'UPPERCASE'    'lowercase'
```

For multiple expressions, enable and disable case-insensitive matching for selected expressions using the `(?i)` and `(?-i)` search flags.

```
expression = {'(?-i)\w*case';...
              '(?i)\w*case'};
matchStr = regexp(str,expression,'match');
celldisp(matchStr)
```

```
matchStr{1}{1} =
   lowercase

matchStr{2}{1} =
   UPPERCASE

matchStr{2}{2} =
   lowercase
```

Input Arguments

`str` — Input text

string | cell array of strings

Input text, specified as a string or a cell array of strings. Each string can be of any length and contain any characters.

If str and expression are both cell arrays, they must have the same dimensions.

Data Types: `char` | `cell`

`expression` — Regular expression

string | cell array of strings

Regular expression, specified as a string or a cell array of strings. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in str.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
`[`$c_1c_2c_3$`]`	Any character contained within the brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[rp.]ain'` matches `'rain'` or `'pain'` or ‘`.ain`’.
`[^`$c_1c_2c_3$`]`	Any character not contained within the brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[^*rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'` and ‘`*ain`’. For example, it matches `'gain'`, `'lain'`, or `'vain'`.
`[`c_1`-`c_2`]`	Any character in the range of c_1 through c_2	`'[A-G]'` matches a single character in the range of `A` through `G`.
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, `\w` is equivalent to `[a-zA-Z_0-9]`	`'\w*'` identifies a word.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, `\W` is equivalent to `[^a-zA-Z_0-9]`	`'\W*'` identifies a substring that is not a word.
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`	`'\w*n\s'` matches words that end with the letter `n`, followed by a white-space character.
`\S`	Any non-white-space character; equivalent to `[^ \f\n\r\t\v]`	`'\d\S'` matches a numeric digit followed by any non-white-space character.
`\d`	Any numeric digit; equivalent to `[0-9]`	`'\d*'` matches any number of consecutive digits.

Metacharacter	Description	Example
`\D`	Any nondigit character; equivalent to `[^0-9]`	`'\w*\D\>'` matches words that do not end with a numeric digit.
`\oN` or `\o{N}`	Character of octal value `N`	`'\o{40}'` matches the space character, defined by octal `40`.
`\xN` or `\x{N}`	Character of hexadecimal value `N`	`'\x2C'` matches the comma character, defined by hex `2C`.

Character Representation

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\char`	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Quantifiers

Quantifiers specify the number of times a string pattern must occur in the matching string.

Quantifier	Matches the expression when it occurs...	Example
`expr*`	0 or more times consecutively.	`'\w*'` matches a word of any length.
`expr?`	0 times or 1 time.	`'\w*(\.m)?'` matches words that optionally end with the extension `.m`.
`expr+`	1 or more times consecutively.	`'<img src="\w+\.gif">'` matches an `<img>` HTML tag when the file name contains one or more characters.

Quantifier	Matches the expression when it occurs...	Example
`expr{m,n}`	At least `m` times, but no more than `n` times consecutively. `{0,1}` is equivalent to `?`.	`'\S{4,8}'` matches between four and eight non-white-space characters.
`expr{m,}`	At least `m` times consecutively. `{0,}` and `{1,}` are equivalent to `*` and `+`, respectively.	`'<a href="\w{1,}\.html">'` matches an `<a>` HTML tag when the file name contains one or more characters.
`expr{n}`	Exactly `n` times consecutively. Equivalent to `{n,n}`.	`'\d{4}'` matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
`expr`*q*	Greedy expression: match as many characters as possible.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*>'` matches all characters between `<tr` and `/td>`: `'<tr><td><p>text</p></td>'`
`expr`*q*`?`	Lazy expression: match as few characters as necessary.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*?>'` ends each match at the first occurrence of the closing bracket (`>`): `'<tr>'  '<td>'  '</td>'`
`expr`*q*`+`	Possessive expression: match as much as possible, but do not rescan any portions of the string.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*+>'` does not return any matches, because the closing bracket is captured using `.*`, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
`(expr)`	Group elements of the expression and capture tokens.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`(?:expr)`	Group, but do not capture tokens.	`'(?:[aeiou][^aeiou]){2}'` matches two consecutive patterns of a vowel followed by a nonvowel, such as `'anon'`. Without grouping, `'[aeiou][^aeiou]{2}'`matches a vowel followed by two nonvowels.
`(?>expr)`	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	`'A(?>.*)Z'` does not match `'AtoZ'`, although `'A(?:.*)Z'` does. Using the atomic group, `Z` is captured using `.*` and is not rescanned.
`(expr1|expr2)`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored. You can include `?:` or `?>` after the opening parenthesis to suppress tokens or group atomically.	`'(let|tel)\w+'` matches words in a string that start with `let` or `tel`.

Anchors

Anchors in the expression match the beginning or end of the string or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input string.	`'^M\w*'` matches a word starting with `M` at the beginning of the string.
`expr$`	End of the input string.	`'\w*m$'` matches words ending with `m` at the end of the string.
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.

Anchor	Matches the...	Example
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for string patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches strings that are followed by `ing`, such as `'Fly'` and `'fall'` in the input string `'Flying, not falling.'`
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.
`(?<=test)expr`	Look behind for characters that match `test`.	`'(?<=re)\w*'` matches strings that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input string `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which string, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

Conditional Operator	Description	Example
`expr1|expr2`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored.	`'(let|tel)\w+'` matches words in a string that start with `let` or `tel`.
`(?(cond)expr)`	If condition `cond` is `true`, then match `expr`.	`'(?(?@ispc)[A-Z]:\\)'` matches a drive name, such as `C:\`, when run on a Windows system.
`(?(cond)expr1|expr2)`	If condition `cond` is `true`, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the string (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
`(expr)`	Capture in a token the characters that match the enclosed expression.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`\N`	Match the `N`th token.	`'<(\w+).*>.*</\1>'` captures tokens for HTML tags, such as `'title'` from the string `'<title>Some text</title>'`.

Ordinal Token Operator	Description	Example
`(?(N)expr1|expr2)`	If the Nth token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his) \w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Named Token Operator	Description	Example
`(?<name>expr)`	Capture in a named token the characters that match the enclosed expression.	`'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)'` creates named tokens for the month, day, and year in an input date string of the form `mm-dd-yy`.
`\k<name>`	Match the token referred to by `name`.	`'<(?<tag>\w+).*>.*</\k<tag>>'` captures tokens for HTML tags, such as `'title'` from the string `'<title>Some text</title>'`.
`(?(name)expr1|expr2)`	If the named token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Note: If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting string in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the string. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement strings.

Replacement String Operator	Description
`$&` or `$0`	Portion of the input string that is currently a match
`$``	Portion of the input string that precedes the current match
`$'`	Portion of the input string that follows the current match (use `$''` to represent the string `$'`)
`$N`	Nth token
`$<name>`	Named token
`${cmd}`	String returned when MATLAB executes the command, `cmd`

Comments

Characters	Description	Example
`(?#comment)`	Insert a comment in the regular expression. The comment text is ignored when matching the input string.	`'(?# Initial digit)\<\d\w+'` includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an option input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern string with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of a string (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\ '` and `'\#'` to match space and # characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types: `char` | `cell`

`outkey` — Keyword that indicates which outputs to return

`'start'` | `'end'` | `'tokenExtents'` | `'match'` | `'tokens'` | `'names'` | `'split'`

Keyword that indicates which outputs to return, specified as one of the following strings.

Output Keyword	Returns
`'start'`	Starting indices of all matches, startIndex
`'end'`	Ending indices of all matches, endIndex
`'tokenExtents'`	Starting and ending indices of all tokens
`'match'`	Text of each substring that matches the pattern in expression
`'tokens'`	Text of each captured token in str
`'names'`	Name and text of each named token
`'split'`	Text of nonmatching substrings of str

Data Types: `char`

`option` — Search option

`'once' | 'warnings' | 'matchcase' | 'emptymatch' | 'dotexceptnewline' | 'lineanchors' | ...`

Search option, specified as a string. Options come in pairs: one option that corresponds to the default behavior, and one option that allows you to override the default. Specify only one option from a pair. Options can appear in any order.

Default	Override	Description
`'all'`	`'once'`	Match the expression as many times as possible (default), or only once.
`'nowarnings'`	`'warnings'`	Suppress warnings (default), or display them.
`'ignorecase'`	`'matchcase'`	Ignore letter case (default), or match case.
`'noemptymatch'`	`'emptymatch'`	Ignore zero length matches (default), or include them.
`'dotall'`	`'dotexceptnewline'`	Match dot with any character (default), or all except newline (`\n`).
`'stringanchors'`	`'lineanchors'`	Apply `^` and `$` metacharacters to the beginning and end of a string (default), or to the beginning and end of a line.

Default	Override	Description
`'literalspacing'`	`'freespacing'`	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and # characters.

Data Types: `char`

Output Arguments

`startIndex` — Starting index of each match

row vector | cell array of row vectors

Starting indices of each match, returned as a row vector or cell array, as follows:

- If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`endIndex` — Ending index of each match

row vector | cell array of row vectors

Ending index of each match, returned as a row vector or cell array, as follows:

- If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`out` — Information about matches

numeric array | cell array | structure array

Information about matches, returned as a numeric, cell, or structure array. The information in the output depends upon the value you specify for outkey, as follows.

Output Keyword	Output Description	Output Type and Dimensions
`'start'`	Starting indices of matches	For both `'start'` and `'end'`: • If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array). • If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.
`'end'`	Ending indices of matches	
`'tokenExtents'`	Starting and ending indices of all tokens	By default, when returning all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array, where `n` is the number of matches. Each cell contains an `m`-by-2 numeric array of indices, where `m` is the number of tokens in the match. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array, where each inner cell contains an `m`-by-2 numeric array. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either an `m`-by-2 numeric array or a cell array with the same dimensions as `str` and/or `expression`.

Output Keyword	Output Description	Output Type and Dimensions
		If a token is expected at a particular index `N`, but is not found, then MATLAB returns extents for that token of `[N,N-1]`.
`'match'`	Text of each substring that matches the pattern in expression	By default, when returning all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array of strings, where `n` is the number of matches. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array of strings. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either a string or a cell array of strings with the same dimensions as `str` and `expression`.

Output Keyword	Output Description	Output Type and Dimensions
`'tokens'`	Text of each captured token in str	By default, when returning all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array, where `n` is the number of matches. Each cell contains a 1-by-`m` cell array of strings, where `m` is the number of tokens in the match. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array, where each inner cell contains a 1-by-`m` cell array of strings. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is a 1-by-`m` cell array of strings or a cell array that has the same dimensions as `str` and/or `expression`. If a token is expected at a particular index, but is not found, then MATLAB returns an empty string for the token, `''`.
`'names'`	Name and text of each named token	For all matches: • If str and expression are both strings, the output is a 1-by-`n` structure array, where `n` is the number of matches. The structure field names correspond to the token names. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` structure array. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

Output Keyword	Output Description	Output Type and Dimensions
`'split'`	Text of nonmatching substrings of str	For all matches: • If str and expression are both strings, the output is a 1-by-`n`-cell array of strings, where `n` is the number of nonmatching strings. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array of strings. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

More About

Tokens

Tokens are portions of the matched text that correspond to portions of the regular expression. To create tokens, enclose part of the regular expression in parentheses.

For example, this expression finds a date of the form `dd-mmm-yyyy`, including tokens for the day, month, and year.

```
str = 'Here is a date: 01-Apr-2020';
expression = '(\d+)-(\w+)-(\d+)';

mydate = regexp(str,expression,'tokens');
mydate{:}
```

```
ans =
    '01'    'April'    '2020'
```

You can associate names with tokens so that they are more easily identifiable:

```
str = 'Here is a date: 01-Apr-2020';
expression = '(?<day>\d+)-(?<month>\w+)-(?<year>\d+)';

mydate = regexp(str,expression,'names')
```

```
mydate =
```

```
      day: '01'
    month: 'Apr'
     year: '2020'
```

For more information, see “Tokens in Regular Expressions”.

- “Lookahead Assertions in Regular Expressions”
- “Dynamic Regular Expressions”

See Also

`regexp` | `regexprep` | `regexptranslate` | `strfind` | `strjoin` | `strrep` | `strsplit`

Introduced before R2006a

regexprep

Replace string using regular expression

Syntax

```
newStr = regexprep(str,expression,replace)
newStr = regexprep(str,expression,replace,option1,...optionM)
```

Description

`newStr = regexprep(str,expression,replace)` replaces the text in str that matches expression with the text described by replace. The `regexprep` function returns the updated text in newStr.

- If str is a string, then newStr is also a string, even when expression or replace is a cell array of strings. When `expression` is a cell array, `regexprep` applies the first expression to the string, and then applies each subsequent expression to the preceding result.
- If str is a cell array, then newStr is a cell array with the same dimensions as str. For each element of str, the `regexprep` function applies each expression in sequence.
- If there are no matches to expression, then newStr is equivalent to str.

`newStr = regexprep(str,expression,replace,option1,...optionM)` modifies the search using the specified options. For example, specify `'ignorecase'` to perform a case-insensitive match.

Examples

Update a Single String

Replace words that begin with `M`, end with `y`, and have at least one character between them.

```
str = 'My flowers may bloom in May';
expression = 'M(\w+)y';
```

```
replace = 'April';

newStr = regexprep(str,expression,replace)

newStr =

My flowers may bloom in April
```

Include Tokens in Replacement Text

Replace variations of the phrase `'walk up'` by capturing the letters that follow `'walk'` in a token.

```
str = 'I walk up, they walked up, we are walking up.';
expression = 'walk(\w*) up';
replace = 'ascend$1';

newStr = regexprep(str,expression,replace)

newStr =

I ascend, they ascended, we are ascending.
```

Include Dynamic Expression in Replacement Text

Replace lowercase letters at the beginning of sentences with their uppercase equivalents using the `upper` function.

```
str = 'here are two sentences. neither is capitalized.';
expression = '(^|\.)\s*.';
replace = '${upper($0)}';

newStr = regexprep(str,expression,replace)

newStr =

Here are two sentences. Neither is capitalized.
```

The regular expression matches single characters (`.`) that follow the beginning of the string (`^`) or a period (`\.`) and any whitespace (`\s*`). The `replace` expression calls the `upper` function for the currently matching character (`$0`).

Update Multiple Strings

Replace each occurrence of a double letter in a set of strings with the symbols `'--'`.

```
str = {                                    ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};

expression = '(.)\1';
replace = '--';
newStr = regexprep(str,expression,replace)

newStr =

    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

Preserve Case in Original String

Ignore letter case in the regular expression when finding matches, but mimic the letter case of the original string when updating.

```
str = 'My flowers may bloom in May';
expression = 'M(\w+)y';
replace = 'April';

newStr = regexprep(str,expression,replace,'preservecase')

newStr =

My flowers april bloom in April
```

Replace Zero-Length Matches

Insert text at the beginning of a string using the `'^'` operator, which returns a zero-length match, and the `'emptymatch'` keyword.

```
str = 'abc';
expression = '^';
replace = '__';

newStr = regexprep(str,expression,replace,'emptymatch')

newStr =
```

```
__abc
```

Input Arguments

`str` — Text to update

string | cell array of strings

Text to update, specified as a string or a cell array of strings.

Data Types: `char` | `cell`

`expression` — Regular expression

string | cell array of strings

Regular expression, specified as a string or a cell array of strings. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in str.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
`[`$c_1c_2c_3$`]`	Any character contained within the brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[rp.]ain'` matches `'rain'` or `'pain'` or ‘`.ain`’.
`[^`$c_1c_2c_3$`]`	Any character not contained within the brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[^*rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'` and ‘`*ain`’. For example, it matches `'gain'`, `'lain'`, or `'vain'`.

Metacharacter	Description	Example
`[c1-c2]`	Any character in the range of `c1` through `c2`	`'[A-G]'` matches a single character in the range of `A` through `G`.
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, `\w` is equivalent to `[a-zA-Z_0-9]`	`'\w*'` identifies a word.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, `\W` is equivalent to `[^a-zA-Z_0-9]`	`'\W*'` identifies a substring that is not a word.
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`	`'\w*n\s'` matches words that end with the letter `n`, followed by a white-space character.
`\S`	Any non-white-space character; equivalent to `[^ \f\n\r\t\v]`	`'\d\S'` matches a numeric digit followed by any non-white-space character.
`\d`	Any numeric digit; equivalent to `[0-9]`	`'\d*'` matches any number of consecutive digits.
`\D`	Any nondigit character; equivalent to `[^0-9]`	`'\w*\D\>'` matches words that do not end with a numeric digit.
`\oN` or `\o{N}`	Character of octal value `N`	`'\o{40}'` matches the space character, defined by octal `40`.
`\xN` or `\x{N}`	Character of hexadecimal value `N`	`'\x2C'` matches the comma character, defined by hex `2C`.

Character Representation

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab

Operator	Description
`\v`	Vertical tab
`\char`	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Quantifiers

Quantifiers specify the number of times a string pattern must occur in the matching string.

Quantifier	Matches the expression when it occurs...	Example
`expr*`	0 or more times consecutively.	`'\w*'` matches a word of any length.
`expr?`	0 times or 1 time.	`'\w*(\.m)?'` matches words that optionally end with the extension `.m`.
`expr+`	1 or more times consecutively.	`'<img src="\w+\.gif">'` matches an `<img>` HTML tag when the file name contains one or more characters.
`expr{m,n}`	At least `m` times, but no more than `n` times consecutively. `{0,1}` is equivalent to `?`.	`'\S{4,8}'` matches between four and eight non-white-space characters.
`expr{m,}`	At least `m` times consecutively. `{0,}` and `{1,}` are equivalent to `*` and `+`, respectively.	`'<a href="\w{1,}\.html">'` matches an `<a>` HTML tag when the file name contains one or more characters.
`expr{n}`	Exactly `n` times consecutively. Equivalent to `{n,n}`.	`'\d{4}'` matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
`expr`*q*	Greedy expression: match as many characters as possible.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*>'` matches all characters between `<tr` and `/td>`:

Mode	Description	Example
		`'<tr><td><p>text</p></td>'`
`expr`*q*`?`	Lazy expression: match as few characters as necessary.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*?>'` ends each match at the first occurrence of the closing bracket (`>`): `'<tr>'  '<td>'  '</td>'`
`expr`*q*`+`	Possessive expression: match as much as possible, but do not rescan any portions of the string.	Given the string `'<tr><td><p>text</p></td>'`, the expression `'</?t.*+>'` does not return any matches, because the closing bracket is captured using `.*`, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
`(expr)`	Group elements of the expression and capture tokens.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`(?:expr)`	Group, but do not capture tokens.	`'(?:[aeiou][^aeiou]){2}'` matches two consecutive patterns of a vowel followed by a nonvowel, such as `'anon'`. Without grouping, `'[aeiou][^aeiou]{2}'` matches a vowel followed by two nonvowels.
`(?>expr)`	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	`'A(?>.*)Z'` does not match `'AtoZ'`, although `'A(?:.*)Z'` does. Using the atomic group, `Z` is captured using `.*` and is not rescanned.
`(expr1|expr2)`	Match expression `expr1` or expression `expr2`.	`'(let|tel)\w+'` matches words in a string that start with `let` or `tel`.

Grouping Operator	Description	Example
	If there is a match with `expr1`, then `expr2` is ignored. You can include `?:` or `?>` after the opening parenthesis to suppress tokens or group atomically.	

Anchors

Anchors in the expression match the beginning or end of the string or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input string.	`'^M\w*'` matches a word starting with `M` at the beginning of the string.
`expr$`	End of the input string.	`'\w*m$'` matches words ending with `m` at the end of the string.
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for string patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches strings that are followed by `ing`, such as `'Fly'` and `'fall'` in the input string `'Flying, not falling.'`

Lookaround Assertion	Description	Example
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.
`(?<=test)expr`	Look behind for characters that match `test`.	`'(?<=re)\w*'` matches strings that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input string `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which string, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

Conditional Operator	Description	Example
`expr1|expr2`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored.	`'(let|tel)\w+'` matches words in a string that start with `let` or `tel`.

Conditional Operator	Description	Example
`(?(cond)expr)`	If condition `cond` is `true`, then match `expr`.	`'(?(?@ispc)[A-Z]:\\)'` matches a drive name, such as `C:\`, when run on a Windows system.
`(?(cond)expr1|expr2)`	If condition `cond` is `true`, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the string (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
`(expr)`	Capture in a token the characters that match the enclosed expression.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`\N`	Match the `N`th token.	`'<(\w+).*>.*</\1>'` captures tokens for HTML tags, such as `'title'` from the string `'<title>Some text</title>'`.
`(?(N)expr1|expr2)`	If the `N`th token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Named Token Operator	Description	Example
`(?<name>expr)`	Capture in a named token the characters that match the enclosed expression.	`'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)'` creates named tokens for the month, day, and year in an

Named Token Operator	Description	Example
		input date string of the form `mm-dd-yy`.
`\k<name>`	Match the token referred to by `name`.	`'<(?<tag>\w+).*>.*</\k<tag>>'` captures tokens for HTML tags, such as `'title'` from the string `'<title>Some text</title>'`.
`(?(name)expr1|expr2)`	If the named token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*'` matches strings that include `her` when the string begins with `Mrs`, or that include `his` when the string begins with `Mr`.

Note: If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting string in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the string. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching

Operator	Description	Example
		`'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement strings.

Replacement String Operator	Description
`$&` or `$0`	Portion of the input string that is currently a match
`$``	Portion of the input string that precedes the current match
`$'`	Portion of the input string that follows the current match (use `$''` to represent the string `$'`)
`$N`	`N`th token
`$<name>`	Named token
`${cmd}`	String returned when MATLAB executes the command, `cmd`

Comments

Characters	Description	Example
`(?#comment)`	Insert a comment in the regular expression. The comment text is ignored when matching the input string.	`'(?# Initial digit)\<\d\w+'` includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an option input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern string with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of a string (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\ '` and `'\#'` to match space and # characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types: `char` | `cell`

`replace` — Replacement text

string | cell array of strings

Replacement text, specified as a string or a cell array of strings, as follows:

- If `replace` is a single string and `expression` is a cell array of strings, then `regexprep` uses the same replacement text for each expression.
- If `replace` is a cell array of `N` strings and `expression` is a single string, then `regexprep` attempts `N` matches and replacements.
- If both `replace` and `expression` are cell arrays of strings, then they must contain the same number of elements. `regexprep` pairs each `replace` element with its matching element in `expression`.

The replacement text can include regular characters, special characters (such as tabs or new lines), or string operators, as shown in the following tables.

Replacement String Operator	Description
`$&` or `$0`	Portion of the input string that is currently a match
`$``	Portion of the input string that precedes the current match
`$'`	Portion of the input string that follows the current match (use `$''` to represent the string `$'`)
`$N`	Nth token
`$<name>`	Named token
`${cmd}`	String returned when MATLAB executes the command, `cmd`

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\`*`char`*	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Data Types: `char` | `cell`

`option` — Search or replacement option

`'once'` | `N` | `'warnings'` | `'ignorecase'` | `'preservecase'` | `'emptymatch'` | `'dotexceptnewline'` | `'lineanchors'` | ...

Search or replacement option, specified as a string or an integer value, as shown in the following table.

Options come in sets: one option that corresponds to the default behavior, and one or two options that allow you to override the default. Specify only one option from a set. Options can appear in any order.

Default	Override	Description
`'all'`	`'once'`	Match and replace the expression as many times as possible (default), or only once.
	`N`	Replace only the `N`th occurrence of the match, where `N` is an integer value.
`'nowarnings'`	`'warnings'`	Suppress warnings (default), or display them.
`'matchcase'`	`'ignorecase'`	Match letter case (default), or ignore case while matching and replacing.
	`'preservecase'`	Ignore case while matching, but preserve the case of corresponding characters in the original string while replacing.
`'noemptymatch'`	`'emptymatch'`	Ignore zero length matches (default), or include them.
`'dotall'`	`'dotexceptnewline'`	Match dot with any character (default), or all except newline (`\n`).
`'stringanchors'`	`'lineanchors'`	Apply `^` and `$` metacharacters to the beginning and end of a string (default), or to the beginning and end of a line.
`'literalspacing'`	`'freespacing'`	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and # characters.

Data Types: `char`

Output Arguments

`newStr` — Updated text

string | cell array of strings

Updated text, returned as a string or a cell array of strings. The data type of newStr is the same as the data type of str.

More About

- “Lookahead Assertions in Regular Expressions”
- “Tokens in Regular Expressions”
- “Dynamic Regular Expressions”

See Also

`regexp` | `strcmp` | `strfind` | `strrep`

Introduced before R2006a

regexptranslate

Translate string into regular expression

Syntax

```
s2 = regexptranslate(type, s1)
```

Description

`s2 = regexptranslate(type, s1)` translates string `s1` into a regular expression string `s2` that you can then use as input into one of the MATLAB regular expression functions such as `regexp`. The `type` input can be either one of the following strings that define the type of translation to be performed. See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for more information.

Type	Description
`'escape'`	Translate all special characters (e.g., '$', '.', '?', '[') in string `s1` so that they are treated as literal characters when used in the `regexp` and `regexprep` functions. The translation inserts an escape character ('\') before each special character in `s1`. Return the new string in `s2`.
`'wildcard'`	Translate all wildcard and '.' characters in string `s1` so that they are treated as literal wildcards and periods when used in the `regexp` and `regexprep` functions. The translation replaces all instances of '*' with '.*', all instances of '?' with '.', and all instances of '.' with '\.'. Return the new string in `s2`.

Examples

Example 1 — Using the 'escape' Option

Because `regexp` interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\' and 'n' in this string:

```
str = 'The sequence \n generates a new line';
```

```
pat = '\n';

regexp(str, pat)

ans =

     []
```

To have `regexp` interpret the expression `expr` as the characters '\' and 'n', first translate the expression using `regexptranslate`:

```
pat2 = regexptranslate('escape', pat)

pat2 =

\\n

regexp(str, pat2)

ans =

    14
```

Example 2 — Using 'escape' In a Replacement String

Replace the word '`walk`' with '`ascend`' in this string, treating the characters '`$1`' as a token designator:

```
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';

regexprep(str, pat, 'ascend$1')

ans =

I ascend, they ascended, we are ascending.
```

Make another replacement on the same string, this time treating the '`$1`' as literal characters:

```
regexprep(str, pat, regexptranslate('escape', 'ascend$1'))

ans =
```

```
I ascend$1, they ascend$1, we are ascend$1.
```

Example 3 — Using the 'wildcard' Option

Given the following string of filenames, pick out just the MAT-files. Use `regexptranslate` to interpret the `'*'` wildcard as `'\w+'` instead of as a regular expression quantifier:

```
files = ['test1.mat, myfile.mat, newfile.txt, ' ...
         'jan30.mat, table3.xls'];
regexp(files, regexptranslate('wildcard', '*.mat'), 'match')

ans =

    'test1.mat, myfile.mat, newfile.txt, jan30.mat'
```

To see the translation, you can type

```
regexptranslate('wildcard','*.mat')

ans =

.*\.mat
```

More About

- "Regular Expressions"

See Also

`regexp` | `regexpi` | `regexprep`

Introduced in R2006a

registerevent

Associate event handler for COM object event at run time

Syntax

```
registerevent(h,eventhandler)
```

Description

`registerevent(h,eventhandler)` registers event handler routines with their corresponding events. The `eventhandler` argument can be either a string that specifies the name of the event handler function, or a function handle that maps to that function. Strings used in the `eventhandler` argument are not case-sensitive.

COM functions are available on Microsoft Windows systems only.

Examples

Show events in the MATLAB sample control:

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f);
events(h)
```

```
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
     Variant x, Variant y)
   Event_Args = void Event_Args(int16 typeshort,
     int32 typelong, double typedouble, string typestring,
     bool typebool)
```

MATLAB displays all events associated with the instance of the control.

Register all events with the same event handler routine, `sampev`:

```
registerevent(h,'sampev');
```

```
eventlisteners(h)

ans =
    'Click'         'sampev'
    'DblClick'      'sampev'
    'MouseDown'     'sampev'
    'Event_Args'    'sampev'
```

Register individual events:

```
% Unregister existing events
unregisterallevents(h);
% Register specific events
registerevent(h,{'click' 'myclick'; ...
    'dblclick' 'my2click'});
eventlisteners(h)

ans =
    'click'       'myclick'
    'dblclick'    'my2click'
```

Register events using a function handle (`@sampev`) instead of the function name:

```
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200]);
registerevent(h,@sampev);
```

More About

- "Writing Event Handlers"

See Also

`events (COM)` | `eventlisteners` | `unregisterevent` | `unregisterallevents` | `isevent`

Introduced before R2006a

rehash

Refresh function and file system path caches

Syntax

```
rehash
rehash path
rehash toolbox
rehash pathreset
rehash toolboxreset
rehash toolboxcache
```

Description

`rehash` with no arguments updates the MATLAB list of known files and classes for directories on the search path that are not in *`matlabroot`*`/toolbox`. It compares the timestamps for loaded functions against their timestamps on disk. It clears loaded functions if the files on disk are newer. All of this normally happens each time MATLAB displays the Command Window prompt. Use `rehash` with no arguments only when you run a program file that updates another program file, and the calling file needs to reuse the updated version of the second file before the calling file has finished running.

`rehash path` performs the same updates as `rehash`, but uses a different technique for detecting the files and directories that require updates. Run `rehash path` only if you receive a warning during MATLAB startup notifying you that MATLAB could not tell if a directory has changed, and you encounter problems with MATLAB not using the most current versions of your program files.

`rehash toolbox` performs the same updates as rehash path, except it updates the list of known files and classes for *all* directories on the search path, including those in *`matlabroot`*`/toolbox`. Run `rehash toolbox` when you change, add, or remove files in *`matlabroot`*`/toolbox` during a session. Typically, you should not make changes to files and directories in *`matlabroot`*`/toolbox`.

`rehash pathreset` performs the same updates as `rehash path`, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash toolboxreset` performs the same updates as `rehash toolbox`, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash toolboxcache` performs the same updates as `rehash toolbox`, and also updates the cache file. This is the equivalent of clicking the **Update Toolbox Path Cache** button in the General Preferences dialog box.

More About

- "Toolbox Path Caching in MATLAB"
- "What Is the MATLAB Search Path?"

See Also

`addpath` | `path` | `rmpath` | `clear` | `matlabroot`

Introduced before R2006a

release

Release COM interface

Syntax

```
release(h)
```

Description

`release(h)` releases the interface and all resources used by the interface. Other interfaces on that object might still be active.

You must release the handle when you are done with the interface. A released interface is no longer valid. MATLAB generates an error if you try to use an object that represents that interface.

To release the interface and delete the control itself, use the `delete` function.

COM functions are available on Microsoft Windows systems only.

More About

- Releasing Interfaces

See Also

`delete (COM)` | `actxcontrol` | `actxserver`

Introduced before R2006a

rem

Remainder after division

Syntax

```
R = rem(X,Y)
```

Description

`R = rem(X,Y)` returns the remainder after division of `X` by `Y`. In general, if `Y` does not equal `0`, `R = rem(X,Y)` returns `X - n.*Y`, where `n = fix(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. Inputs `X` and `Y` must have the same dimensions unless one of them is a scalar double. If one of the inputs has an integer data type, then the other input must be of the same integer data type or be a scalar double.

The following are true by convention:

- `rem(X,0)` is `NaN`.
- `rem(X,X)` for `X~=0` is `0`.
- `rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

Examples

Remainder of Two Scalars

Compute the remainder after dividing 5 into 23.

```
X = 23;
Y = 5;
R = rem(X,Y)
```

```
R =
```

```
     3
```

Remainder of a Vector

Create a vector, then use `rem` to find the remainder after dividing a scalar into each element of the vector.

```
X = 1:5;
Y = 3;
R = rem(X,Y)

R =

     1     2     0     1     2
```

When you specify one or more of the inputs as an array, the `rem` function acts on each array element independently.

Remainder of Two Arrays

Create two 3-by-3 matrices, then use `rem` to find the remainder after dividing `Y` into `X`.

```
X = [1 2 3;4 5 6;7 8 9];
Y = [9 8 7;6 5 4;3 2 1];
R = rem(X,Y)

R =

     1     2     3
     4     0     2
     1     0     0
```

Inputs `X` and `Y` must have the same dimensions unless one is a scalar double.

Forced Rounding in rem

If `Y` is not an integer and `X./Y` is within roundoff error of an integer, then `rem` rounds to that integer for its calculation. The size of the roundoff error is very small.

```
X = 2;
Y = 2 - eps(2)

Y =

    2.0000
```

It looks like `Y` is trivially equal to 2, but in fact there is an infinitesimal difference.

```
2 - Y
```

```
ans =

  4.4409e-16
```

This difference is forced to zero by `rem` if it is small enough.

```
R = rem(X,Y)
```

```
R =

     0
```

Make the difference a little larger and the forced rounding disappears.

```
Y = 2 - eps(4);
R = rem(X,Y)
```

```
R =

   8.8818e-16
```

Difference Between rem and mod

Define `X` and `Y` with different signs.

```
X = 5;
Y = -2;
```

Compute the remainder after division with `rem`, then compute the modulus after division with `mod`.

```
R = rem(X,Y)
```

```
R =

     1
```

```
M = mod(X,Y)
```

```
M =

    -1
```

`rem(X,Y)` and `mod(X,Y)` are equal if `X` and `Y` have the same sign, but differ by `Y` if `X` and `Y` have different signs. Notice that `rem` retains the sign of `X`, while `mod` retains the sign of `Y`.

Input Arguments

`X` — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a scalar, vector, matrix, or multidimensional array. Must be a real-valued number of any numerical type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

`Y` — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a scalar, vector, matrix, or multidimensional array. Must be a real-valued number of any numerical type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

See Also

`mod`

Introduced before R2006a

remove

Class: containers.Map
Package: containers

Remove key-value pairs from `containers.Map` object

Syntax

```
remove(mapObj,keySet)
```

Description

`remove(mapObj,keySet)` erases all specified keys, and the values associated with them, from `mapObj`. Input `keySet` can be a scalar key or a cell array of keys.

Input Arguments

`mapObj`

Object of class `containers.Map`.

`keySet`

Scalar value, string, or cell array that specifies keys in `mapObj` to delete.

Examples

Remove Key-Value Pairs from a Map

Create a map and view the keys and the `Count` property:

```
myKeys = {'a','b','c','d'};
myValues = [1,2,3,4];
mapObj = containers.Map(myKeys,myValues);
```

```
mapKeys = keys(mapObj)
mapCount = mapObj.Count
```

The initial map contains four key-value pairs:

```
mapKeys =
    'a'    'b'    'c'    'd'

mapCount =
                    4
```

Remove the pairs corresponding to keys `b` and `d`:

```
keySet = {'b','d'};
remove(mapObj,keySet)

mapKeys = keys(mapObj)
mapCount = mapObj.Count
```

The modified map contains two key-value pairs:

```
mapKeys =
    'a'    'c'

mapCount =
                    2
```

See Also

`keys` | `values` | `containers.Map` | `isKey`

removecats

Remove categories from categorical array

Syntax

```
B = removecats(A)
B = removecats(A,oldcats)
```

Description

`B = removecats(A)` removes unused categories from the categorical array, `A`. The output categorical array, `B`, has the same size and values as `A`. However, `B` possibly has fewer categories.

`B = removecats(A,oldcats)` removes the categories specified by oldcats. The function `removecats` removes categories, but does not remove any elements of the array. Therefore, elements of `B`, whose values correspond to `oldcats`, are undefined.

Examples

Remove All Unused Categories

Create a categorical array representing political parties of four people.

```
A = categorical({'republican' 'democrat' 'democrat' 'republican'},...
    {'democrat' 'republican' 'independent'})
```

```
A =

     republican      democrat      democrat      republican
```

`A` is a 1-by-4 categorical array.

Summarize the categorical array, `A`.

```
summary(A)
```

```
     democrat      republican      independent
            2               2                0
```

A has three categories. `democrat` appears twice in the array, `republican` appears twice in the array, and `independent` is unused.

Remove the unused category, `independent`.

```
B = removecats(A)
```

```
B =

     republican      democrat      democrat      republican
```

B has the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans =

    'democrat'
    'republican'
```

B has fewer categories than A.

Remove Categories with Corresponding Values Used in A

Create a categorical array, A, containing modes of transportation.

```
A = categorical({'plane' 'car'; 'train' 'car'; 'plane' 'car'})
```

```
A =

     plane      car
     train      car
     plane      car
```

A is a 3-by-2 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans =
```

```
    'car'
    'plane'
    'train'
```

A has three categories, `car`, `plane`, and `train`.

Remove the category, `train`.

```
B = removecats(A,'train')
```

```
B =

     plane            car
     <undefined>      car
     plane            car
```

The element that was from the category `train` is now undefined.

Display the categories of B.

```
categories(B)
```

```
ans =

    'car'
    'plane'
```

B has one fewer category than A.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

oldcats — Categories to remove

string | cell array of strings

Categories to remove, specified as a string or cell array of strings. The default is all the unused categories from A.

More About

Tips

- `~ismember(categories(A),unique(A))` returns logical `true` (1) for any unused category of A.

See Also

`addcats` | `categories` | `iscategory` | `mergecats` | `renamecats` | `reordercats` | `setcats` | `summary`

removets

Remove `timeseries` objects from `tscollection` object

Syntax

```
tsc = removets(tsc,Name)
```

Description

`tsc = removets(tsc,Name)` removes one or more `timeseries` objects with the name specified in `Name` from the `tscollection` object tsc. `Name` can either be a string or a cell array of strings.

Examples

The following example shows how to remove a time series from a `tscollection`.

1. Create two `timeseries` objects, `ts1` and `ts2`.

   ```
   ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
   ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
   ```

2. Create a `tscollection` object `tsc`, which includes `ts1` and `ts2`.

   ```
   tsc=tscollection({ts1 ts2});
   ```

3. To view the members of `tsc`, type the following at the MATLAB prompt:

   ```
   tsc
   ```

   The response is

   ```
   Time Series Collection Object: unnamed

   Time vector characteristics

         Start time            1 seconds
         End time              5 seconds
   ```

```
Member Time Series Objects:

      acceleration
      speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of `ts1` and `ts2`, respectively.

4 Remove `ts2` from `tsc`.

```
tsc=removets(tsc,'speed');
```

5 To view the current members of `tsc`, type the following at the MATLAB prompt:

```
tsc
```

The response is

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
      acceleration
```

The remaining member of `tsc` is `acceleration`. The `timeseries` `speed` has been removed.

See Also

`addts` | `tscollection`

Introduced before R2006a

renamecats

Rename categories in categorical array

Syntax

```
B = renamecats(A,newnames)
B = renamecats(A,oldnames,newnames)
```

Description

`B = renamecats(A,newnames)` renames all the categories in the categorical array, `A`. Elements of `B` use the new category names.

`B = renamecats(A,oldnames,newnames)` renames only the categories specified by `oldnames`.

Examples

Rename All Categories

Create a categorical array containing states from New England.

```
A = categorical({'MA';'ME';'CT';'VT';'ME';'NH';'VT';'MA';'NH';'CT';'RI'})
```

```
A =

     MA
     ME
     CT
     VT
     ME
     NH
     VT
     MA
     NH
     CT
     RI
```

A is an 11-by-1 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans =

    'CT'
    'MA'
    'ME'
    'NH'
    'RI'
    'VT'
```

A has six categories.

Rename the categories to use the full state name instead of the abbreviation.

```
B = renamecats(A,{'Connecticut','Massachusetts',...
    'Maine','New Hampshire','Rhode Island' 'Vermont'})
```

```
B =

     Massachusetts
     Maine
     Connecticut
     Vermont
     Maine
     New Hampshire
     Vermont
     Massachusetts
     New Hampshire
     Connecticut
     Rhode Island
```

Elements of B use the new category names.

Display the categories of B.

```
categories(B)
```

```
ans =

    'Connecticut'
```

```
    'Massachusetts'
    'Maine'
    'New Hampshire'
    'Rhode Island'
    'Vermont'
```

Rename One Category

Create a categorical array containing colors.

```
A = categorical({'red' 'blue'; 'purple' 'white'; 'green' 'red'})
```

```
A =

     red          blue
     purple       white
     green        red
```

`A` is a 3-by-2 categorical array.

Display the categories of `A`.

```
categories(A)
```

```
ans =

    'blue'
    'green'
    'purple'
    'red'
    'white'
```

`A` has five categories that are listed in alphabetical order.

Change the category name from `purple` to `violet`.

```
B = renamecats(A,'purple','violet')
```

```
B =

     red          blue
     violet       white
     green        red
```

The element `B(2,1)` is `violet` instead of `purple`.

Display the categories of B.

```
categories(B)
```

```
ans =

    'blue'
    'green'
    'violet'
    'red'
    'white'
```

violet replaces purple and the categories are no longer in alphabetical order. Note that the category has not changed its position.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

newnames — New category names for B

string | cell array of strings

New category names for B, specified as a string or a cell array of strings. The new category names must be unique, and must not duplicate any existing names.

oldnames — Old category names from A

string | cell array of strings

Old category names from A, specified as a string or a cell array of strings.

More About

Tips

- Renaming categories does not change their position in `categories(B)`. Use `reordercats` to change the category ordering. For example, you can use `B = reordercats(B,sort(categories(B)))` to put the categories in alphabetical order.

See Also

addcats | categories | iscategory | mergecats | removecats | reordercats | setcats

reordercats

Reorder categories in categorical array

Syntax

```
B = reordercats(A)
B = reordercats(A,neworder)
```

Description

`B = reordercats(A)` reorders the categories in the categorical array, `A`, to be in alphanumeric order.

The order of the categories is used by functions such as `summary` and `hist`. If the categorical array is ordinal, the order of the categories defines their mathematical ordering. The first category specified is the smallest and the last category is the largest.

`B = reordercats(A,neworder)` puts the categories in the order specified by neworder.

Examples

Alphabetize Categories of Nonordinal Categorical Array

Create two categorical arrays, `X` and `Y`.

```
X = categorical({'frog';'cat';'cat';'ant';'frog'})

Y = categorical({'deer';'bear';'eagle';'deer'})
```

```
X =

     frog
     cat
     cat
     ant
     frog
```

```
Y =

     deer
     bear
     eagle
     deer
```

X is a 5-by-1 categorical array. The categories of X are the sorted unique values from the array: `{'ant';'cat';'frog'}`.

Y is a 4-by-1 categorical array. The categories of Y are the sorted unique values from the array: `{'bear';'deer';'eagle'}`.

Concatenate X and Y into a single categorical array, A.

```
A = [X;Y]
```

```
A =

     frog
     cat
     cat
     ant
     frog
     deer
     bear
     eagle
     deer
```

`vertcat` appends the values from Y to the values from X.

List the categories of the categorical array, A.

```
acats = categories(A)
```

```
acats =

    'ant'
    'cat'
    'frog'
    'bear'
    'deer'
    'eagle'
```

`vertcat` appends the categories of `Y` to the categories from `X`. The categories of `A` are *not* in alphabetical order.

Reorder the categories of `A` into alphabetical order.

```
B = reordercats(A)
```

```
B =

     frog
     cat
     cat
     ant
     frog
     deer
     bear
     eagle
     deer
```

The output categorical array, `B`, has the same elements in the same order as the input categorical array, `A`.

List the categories of the categorical array, `B`.

```
bcats = categories(B)
```

```
bcats =

    'ant'
    'bear'
    'cat'
    'deer'
    'eagle'
    'frog'
```

The categories of `B` are in alphabetical order.

Reorder Categories in Nonordinal Categorical Array

Create a categorical array containing the color of various items.

```
A = categorical({'red';'green';'blue';'red';'green';'red';'blue';'blue'})
```

```
A =

     red
```

```
     green
     blue
     red
     green
     red
     blue
     blue
```

A is an 8-by-1 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans =

    'blue'
    'green'
    'red'
```

The categories of A are in alphabetical order and have no mathematical meaning.

Reorder the categories to match the order commonly used for colors.

```
B = reordercats(A,{'red','green','blue'})
```

```
B =

     red
     green
     blue
     red
     green
     red
     blue
     blue
```

B contains the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans =
```

```
    'red'
    'green'
    'blue'
```

B is not ordinal and the order of the categories has no mathematical meaning. Although the categories appear in the order of the color spectrum, relational operations, such as greater than and less than, have no meaning.

Reorder Categories in Ordinal Categorical Array

Create an ordinal categorical array, A, containing modes of transportation. Order the categories based on the average price of travel.

```
A = categorical({'plane';'car'; 'train';'car';'plane';'car'},...
    {'car','train','plane'},'Ordinal',true)
```

```
A =

     plane
     car
     train
     car
     plane
     car
```

A is a 6-by-1 ordinal categorical array.

Display the categories of A.

```
categories(A)
```

```
ans =

    'car'
    'train'
    'plane'
```

Since A is ordinal, `car < train < plane`.

Reorder the categories to reflect a decrease in the cost of train travel.

```
B = reordercats(A,{'train','car','plane'})
```

```
B =
```

```
     plane
     car
     train
     car
     plane
     car
```

B contains the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans =

    'train'
    'car'
    'plane'
```

The mathematical ordering of the categories is now `train < car < plane`. The results from relational operations, `min`, and `max` reflect the new category ordering.

Reorder Categories with Numeric Vector

Create a categorical array, A, containing modes of transportation.

```
A = categorical({'plane';'car';'train';'car';'car';'plane';'car'})
```

```
A =

     plane
     car
     train
     car
     car
     plane
     car
```

Display the categories of A.

```
categories(A)
```

```
ans =

    'car'
```

```
    'plane'
    'train'
```

Reorder categories from least to most frequent occurrence in A.

```
B = countcats(A);
[C,neworder] = sort(B);
neworder

neworder =

     3
     2
     1

D = reordercats(A,neworder);
categories(D)

ans =

    'train'
    'plane'
    'car'
```

Because `countcats` counts the occurrences of each category, `neworder` describes how to reorder the categories—not the elements—of A.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array. If A is an ordinal categorical array, a reordering of the categories changes the mathematical meaning. Consequently, the relational operators, such as greater than and less than, might return different results.

neworder — New category order for B

cell array of strings | numeric vector

New category order for B, specified as a cell array of strings or a numeric vector. `neworder` must be a permutation of `categories(A)`.

More About

Tips

- To convert the categorical array, `B`, to an ordinal categorical array, use `B = categorical(B,'Ordinal',true)`. You can specify the order of the categories with `B = categorical(B,`*`valueset`*`,'Ordinal',true)`, where the order of the values in *`valueset`* defines the category order.

See Also

`addcats` | `categories` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `setcats`

rename

Class: FTP

Rename file on FTP server

Syntax

```
rename(ftpobj,oldname,newname)
```

Description

`rename(ftpobj,oldname,newname)` changes the name of the file `oldname` to `newname` in the current folder on an FTP server.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`oldname`

String enclosed in single quotation marks that specifies the original name of the file.

`newname`

String enclosed in single quotation marks that specifies a new name for the file.

Examples

Suppose that hypothetical FTP server `ftp.testsite.com` contains a file named `testfile.m`. Rename the file to `final.m`:

```
test = ftp('ftp.testsite.com');
rename(test, 'testfile.m', 'final.m');
```

```
close(test);
```

See Also

`dir` | `ftp` | `mput` | `delete` | `mget`

Introduced before R2006a

repelem

Repeat copies of array elements

Syntax

```
u = repelem(v,n)
B = repelem(A,r1,...,rN)
```

Description

`u = repelem(v,n)`, where `v` is a scalar or vector, returns a vector of repeated elements of `v`.

- If `n` is a scalar, then each element of `v` is repeated `n` times. The length of `u` is `length(v)*n`.
- If `n` is a vector, then it must be the same length as `v`. Each element of `n` specifies the number of times to repeat the corresponding element of `v`.

This syntax is not supported for `table` input.

`B = repelem(A,r1,...,rN)` returns an array with each element of `A` repeated according to `r1,...,rN`. Each `r1,...,rN` must either be a scalar or a vector with the same length as `A` in the corresponding dimension. For example, if `A` is a matrix, `repelem(A,2,3)` returns a matrix containing a `2`-by-`3` block of each element of `A`.

Examples

Repeat Vector Elements

Create a vector and repeat each of its elements three times into a new vector.

```
v = [1 2 3 4];
u = repelem(v,3)
```

```
u =
```

```
     1     1     1     2     2     2     3     3     3     4     4     4
```

Repeat the first two elements of `v` twice and the last two elements three times.

```
u = repelem(v,[2 2 3 3])
```

```
u =

     1     1     2     2     3     3     3     4     4     4
```

Repeat Matrix Elements

Create a matrix and repeat each element into a 3-by-2 block of a new matrix.

```
A = [1 2; 3 4]
```

```
A =

     1     2
     3     4
```

```
B = repelem(A,3,2)
```

```
B =

     1     1     2     2
     1     1     2     2
     1     1     2     2
     3     3     4     4
     3     3     4     4
     3     3     4     4
```

Repeat Matrix Columns

Create a matrix and copy its columns into a new array, repeating the first column twice and second column three times.

```
A = [1 2; 3 4]
```

```
B = repelem(A,1,[2 3])


A =

     1     2
     3     4


B =

     1     1     2     2     2
     3     3     4     4     4
```

Input Arguments

v — Input element

scalar | vector

Input element, specified as a scalar or a vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `datetime` | `duration`
Complex Number Support: Yes

n — Number of times to repeat each element

scalar | vector

Number of times to repeat each element, specified as a scalar or a vector. If `n` is a scalar, then all elements of `v` are repeated `n` times. If `n` is a vector, then each element of `n` specifies the number of times to repeat the corresponding element of `v`. In either case, `n` must be integer-valued.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

A — Input array

matrix | multidimensional array

Input array, specified as a matrix or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `datetime` | `duration`
Complex Number Support: Yes

`r1,...,rN` — Repetition factors for each dimension (as separate arguments)

scalars | vectors

Repetition factors for each dimension, specified as separate arguments of integer-valued scalars or vectors. If `A` is a table, each repetition factor must be a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`u` — Output vector

vector

Output vector. If `v` is a row vector or scalar, `u` is a row vector. If `v` is a column vector, `u` is also a column vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `datetime` | `duration`

`B` — Output array

matrix | multidimensional array

Output array, returned as a matrix or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `datetime` | `duration`

See Also

`kron` | `repmat`

Introduced in R2015a

repmat

Repeat copies of array

Syntax

```
B = repmat(A,n)
B = repmat(A,r1,...,rN)
B = repmat(A,r)
```

Description

`B = repmat(A,n)` returns an array containing `n` copies of `A` in the row and column dimensions. The size of `B` is `size(A)*n` when `A` is a matrix.

`B = repmat(A,r1,...,rN)` specifies a list of scalars, `r1,..,rN`, that describes how copies of `A` are arranged in each dimension. When `A` has `N` dimensions, the size of `B` is `size(A).*[r1...rN]`. For example, `repmat([1 2; 3 4],2,3)` returns a 4-by-6 matrix.

`B = repmat(A,r)` specifies the repetition scheme with row vector `r`. For example, `repmat(A,[2 3])` returns the same result as `repmat(A,2,3)`.

Examples

Square Block Format

Repeat copies of a matrix into a 2-by-2 block arrangement.

```
A = diag([100 200 300])
```

```
A =

   100     0     0
     0   200     0
     0     0   300
```

```
B = repmat(A,2)

B =

   100     0     0   100     0     0
     0   200     0     0   200     0
     0     0   300     0     0   300
   100     0     0   100     0     0
     0   200     0     0   200     0
     0     0   300     0     0   300
```

Rectangular Block Format

Repeat copies of a matrix into a 2-by-3 block arrangement.

```
A = diag([100 200 300])

A =

   100     0     0
     0   200     0
     0     0   300

B = repmat(A,2,3)

B =

   100     0     0   100     0     0   100     0     0
     0   200     0     0   200     0     0   200     0
     0     0   300     0     0   300     0     0   300
   100     0     0   100     0     0   100     0     0
     0   200     0     0   200     0     0   200     0
     0     0   300     0     0   300     0     0   300
```

3-D Block Array

Repeat copies of a matrix into a 2-by-3-by-2 block arrangement.

```
A = [1 2; 3 4]
```

```
A =

     1     2
     3     4


B = repmat(A,[2 3 2])


B(:,:,1) =

     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4


B(:,:,2) =

     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4
```

Vertical Stack of Row Vectors

Vertically stack a row vector four times.

```
A = 1:4;
B = repmat(A,4,1)


B =

     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
```

Horizontal Stack of Column Vectors

Horizontally stack a column vector four times.

```
A = (1:3)';
B = repmat(A,1,4)
```

```
B =

     1     1     1     1
     2     2     2     2
     3     3     3     3
```

Tabular Block Format

Create a table with variables `Age` and `Height`.

```
A = table([39; 26],[70; 63],'VariableNames',{'Age' 'Height'})
```

```
A =

    Age    Height
    ___    ______

    39     70
    26     63
```

Repeat copies of the table into a 2-by-3 block format.

```
B = repmat(A,2,3)
```

```
B =

    Age    Height    Age_1    Height_1    Age_2    Height_2
    ___    ______    _____    ________    _____    ________

    39     70        39       70          39       70
    26     63        26       63          26       63
    39     70        39       70          39       70
    26     63        26       63          26       63
```

`repmat` repeats the entries of the table and appends a number to the new variable names.

Input Arguments

`A` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`
Complex Number Support: Yes

`n` — Number of times to repeat input array in row and column dimensions

integer value

Number of times to repeat the input array in the row and column dimensions, specified as an integer value. If `n` is `0` or negative, the result is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`r1,...,rN` — Repetition factors for each dimension (as separate arguments)

integer values

Repetition factors for each dimension, specified as separate arguments of integer values. If any repetition factor is `0` or negative, the result is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`r` — Vector of repetition factors for each dimension (as a row vector)

integer values

Vector of repetition factors for each dimension, specified as a row vector of integer values. If any value in `r` is `0` or negative, the result is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Tips

- To build block arrays by forming the tensor product of the input with an array of ones, use `kron`. For example, to stack the row vector `A = 1:3` four times vertically, you can use `B = kron(A,ones(4,1))`.
- To create block arrays and perform a binary operation in a single pass, use `bsxfun`. In some cases, `bsxfun` provides a simpler and more memory efficient solution. For example, to add the vectors `A = 1:5` and `B = (1:10)'` to produce a 10-by-5 array, use `bsxfun(@plus,A,B)` instead of `repmat(A,10,1) + repmat(B,1,5)`.
- When `A` is a scalar of a certain type, you can use other functions to get the same result as `repmat`.

repmat Syntax	Equivalent Alternative
`repmat(NaN,m,n)`	`NaN(m,n)`
`repmat(single(inf),m,n)`	`inf(m,n,'single')`
`repmat(int8(0),m,n)`	`zeros(m,n,'int8')`
`repmat(uint32(1),m,n)`	`ones(m,n,'uint32')`
`repmat(eps,m,n)`	`eps(ones(m,n))`

See Also

`bsxfun` | `kron` | `meshgrid` | `ndgrid` | `repelem` | `reshape`

Introduced before R2006a

resample (tscollection)

Select or interpolate data in `tscollection` using new time vector

Syntax

```
tsc = resample(tsc,Time)
tsc = resample(tsc,Time,interp_method)
tsc = resample(tsc,Time,interp_method,code)
```

Description

`tsc = resample(tsc,Time)` resamples the `tscollection` object tsc on the new Time vector. When `tsc` uses date strings and Time is numeric, Time is treated as numerical specified relative to the `tsc.TimeInfo.StartDate` property and in the same units that tsc uses. The `resample` method uses the default interpolation method for each time series member.

`tsc = resample(tsc,Time,interp_method)` resamples the `tscollection` object tsc using the interpolation method given by the string interp_method. Valid interpolation methods include `'linear'` and `'zoh'` (zero-order hold).

`tsc = resample(tsc,Time,interp_method,code)` resamples the `tscollection` object tsc using the interpolation method given by the string interp_method. The integer code is a user-defined quality code for resampling, applied to all samples.

Examples

The following example shows how to resample a `tscollection` that consists of two `timeseries` members.

1. Create two `timeseries` objects.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
```

2. Create a `tscollection` `tsc`.

```
tsc=tscollection({ts1 ts2});
```

The time vector of the collection `tsc` is [1:5], which is the same as for `ts1` and `ts2` (individually).

3 Get the interpolation method for `acceleration` by typing

```
tsc.acceleration
```

MATLAB responds with

```
Time Series Object: acceleration

Time vector characteristics

      Length               5
      Start time           1 seconds
      End time             5 seconds

Data characteristics

      Interpolation method  linear
      Size                  [1  1  5]
      Data type             double
```

4 Set the interpolation method for `speed` to zero-order hold by typing

```
setinterpmethod(tsc.speed,'zoh')
```

MATLAB responds with

```
Time Series Object: acceleration

Time vector characteristics

      Length               5
      Start time           1 seconds
      End time             5 seconds

Data characteristics

      Interpolation method  zoh
      Size                  [1  1  5]
```

```
        Data type              double
```

5 Resample the time-series collection `tsc` by individually resampling each time-series member of the collection and using its interpolation method.

```
res_tsc=resample(tsc,[1 1.5 3.5 4.5 4.9])
```

See Also

`getinterpmethod` | `setinterpmethod` | `tscollection`

Introduced before R2006a

reset

Reset graphics object properties to their defaults

Syntax

```
reset(h)
```

Description

`reset(h)` resets all properties on the object identified by `h` to their default values. Properties that do not have default values are not affected.

If `h` is a figure, the MATLAB software does not reset `Position`, `Units`, `WindowStyle`, or `PaperUnits`. If `h` is an `axes`, MATLAB does not reset `Position` and `Units`.

Examples

`reset(gca)` resets the properties of the current axes.

`reset(gcf)` resets the properties of the current figure.

See Also

`cla` | `clf` | `gca` | `gcf` | `hold`

Introduced before R2006a

reset (RandStream)

Reset random number stream

Class

`RandStream`

Syntax

```
reset(s)
reset(s,seed)
```

Description

`reset(s)` resets the generator for the random stream, `s`, to the internal state corresponding to its seed. This is similar to clearing `s` and recreating it using `RandStream(Type,...)`, except that `reset` does not set the stream's `NormalTransform`, `Antithetic`, and `FullPrecision` properties to their original values.

`reset(s,seed)` resets the generator for the random stream, `s`, to the internal state corresponding to `seed` (the seed value), and it updates the `seed` property of `s`. The value of `seed` must be an integer between 0 and $2^{32} - 1$. Resetting a stream's seed can invalidate independence with other streams.

Note: Resetting a stream should be used primarily for reproducing results.

Examples

Example 1

Reset a random number stream to its initial state. This does not create a random number stream, it simply resets the stream:

```
stream = RandStream('twister','Seed',0)

stream =

mt19937ar random stream
             Seed: 0
  NormalTransform: Ziggurat

reset(stream);
stream.Seed

ans =

           0
```

Example 2

Reset a random number stream using a specific seed:

```
stream = RandStream('twister','Seed',0)

stream =

mt19937ar random stream
             Seed: 0
  NormalTransform: Ziggurat

reset(stream,1);
stream.Seed

ans =

           1
```

See Also

RandStream | RandStream.getGlobalStream

reshape

Reshape array

Syntax

```
B = reshape(A,sz)
B = reshape(A,sz1,...,szN)
```

Description

`B = reshape(A,sz)` reshapes `A` using the size vector, `sz`, to define `size(B)`. For example, `reshape(A,[2,3])` reshapes `A` into a 2-by-3 matrix. `sz` must contain at least 2 elements, and `prod(sz)` must be the same as `numel(A)`.

`B = reshape(A,sz1,...,szN)` reshapes `A` into a `sz1`-by-`...`-by-`szN` array where `sz1,...,szN` indicates the size of each dimension. You can specify a single dimension size of `[]` to have the dimension size automatically calculated, such that the number of elements in `B` matches the number of elements in `A`. For example, if `A` is a 10-by-10 matrix, then `reshape(A,2,2,[])` reshapes the 100 elements of `A` into a 2-by-2-by-25 array.

Examples

Reshape Vector into Matrix

Reshape a 1-by-10 vector into a 5-by-2 matrix.

```
A = 1:10;
B = reshape(A,[5,2])
```

```
B =

     1     6
     2     7
     3     8
```

```
     4     9
     5    10
```

Reshape Matrix to Have Specified Number of Columns

Reshape a 6-by-6 magic square matrix into a matrix that has only 3 columns. Specify `[]` for the first dimension size to let `reshape` automatically calculate the appropriate number of rows.

```
A = magic(6);
B = reshape(A,[],3)

B =

    35     6    19
     3     7    23
    31     2    27
     8    33    10
    30    34    14
     4    29    18
     1    26    24
    32    21    25
     9    22    20
    28    17    15
     5    12    16
    36    13    11
```

The result is a 12-by-3 matrix, which maintains the same number of elements (36) as the original 6-by-6 matrix. The elements in `B` also maintain their columnwise order from `A`.

Reshape Multidimensional Array into Matrix

Reshape a 3-by-2-by-3 array of zeros into a 9-by-2 matrix.

```
A = zeros(3,2,3);
B = reshape(A,9,2)

B =

     0     0
     0     0
     0     0
     0     0
     0     0
```

```
     0     0
     0     0
     0     0
     0     0
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `cell` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

sz — Output size

row vector of integers

Output size, specified as a row vector of integers. Each element of `sz` indicates the size of the corresponding dimension in `B`. You must specify `sz` so that the number of elements in `A` and `B` are the same. That is, `prod(sz)` must be the same as `numel(A)`.

Beyond the second dimension, the output, `B`, does not reflect trailing dimensions with a size of `1`. For example, `reshape(A,[3,2,1,1])` produces a 3-by-2 matrix.

Example: `reshape(A,[3,2])`

Example: `reshape(A,[6,4,10])`

Example: `reshape(A,[5,5,5,5])`

sz1,...,szN — Size of each dimension

two or more integers | `[]` (optional)

Size of each dimension, specified as two or more integers with at most one `[]` (optional). You must specify at least 2 dimension sizes, and at most one dimension size can be specified as `[]`, which automatically calculates the size of that dimension to ensure that `numel(B)` matches `numel(A)`. When you use `[]` to automatically calculate a dimension size, the dimensions that you *do* explicitly specify must divide evenly into the number of elements in the input matrix, `numel(A)`.

Beyond the second dimension, the output, `B`, does not reflect trailing dimensions with a size of `1`. For example, `reshape(A,3,2,1,1)` produces a 3-by-2 matrix.

Example: `reshape(A,3,2)`

Example: `reshape(A,6,[],10)`

Example: `reshape(A,2,5,3,[])`

Example: `reshape(A,5,5,5,5)`

Output Arguments

`B` — Reshaped array

vector | matrix | multidimensional array | cell array

Reshaped array, returned as a vector, matrix, multidimensional array, or cell array. The data type and number of elements in `B` are the same as the data type and number of elements in `A`. The elements in `B` preserve their columnwise ordering from `A`.

See Also

`colon (:)` | `permute` | `repmat` | `shiftdim` | `squeeze`

Introduced before R2006a

residue

Convert between partial fraction expansion and ratio of two polynomials

Syntax

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

Description

`[r,p,k] = residue(b,a)` finds the residues, poles, and direct term of a "Partial Fraction Expansion" on page 1-7080 of the ratio of two polynomials, where the expansion is of the form

$$\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + b_3 s^{m-2} + \ldots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + a_3 s^{n-2} + \ldots + a_{n+1}}$$

The inputs to `residue` are vectors of coefficients of the polynomials `b = [bn ... b1 b0]` and `a = [am ... a1 a0]`. The outputs are the residues `r = [rm ... r2 r1]`, the poles `p = [pm ... p2 p1]`, and the polynomial k. For most textbook problems, k is `0` or a constant.

`[b,a] = residue(r,p,k)` converts the partial fraction expansion back to the ratio of two polynomials and returns the coefficients in `b` and `a`.

Examples

Find Partial Fraction Expansion

Find the partial fraction expansion of the following ratio of polynomials *F*(*s*) using `residue`

$$F(s) = \frac{b(s)}{a(s)} = \frac{5s^3 + 3s^2 - 2s + 7}{-4s^3 + 8s + 3}.$$

```
b = [ 5 3 -2 7];
a = [-4 0 8 3];
[r,p,k] = residue(b,a)
```

```
r =

   -1.4167
   -0.6653
    1.3320


p =

    1.5737
   -1.1644
   -0.4093


k =

   -1.2500
```

This represents the partial fraction expansion

$$F(s) = \frac{b(s)}{a(s)} = \frac{-1.4167}{s - 1.5737} - \frac{0.6653}{s + 1.1644} + \frac{1.3320}{s + 0.4093} - 1.2500.$$

Convert the partial fraction expansion back to polynomial coefficients using `residue`.

```
[b,a] = residue(r,p,k)
```

```
b =

   -1.2500   -0.7500    0.5000   -1.7500


a =

    1.0000   -0.0000   -2.0000   -0.7500
```

The result is normalized for the leading coefficient in the denominator and represents

$$F(s) = \frac{b(s)}{a(s)} = \frac{-1.25s^3 - 0.75s^2 + 0.50s - 1.75}{s^3 - 2s - 0.75}.$$

Expansion with Numerator Degree Greater Than Denominator Degree

The ratio of polynomials *F*(*s*) and its partial fraction expansion is

$$F(s) = \frac{b(s)}{a(s)} = \frac{2s^4 + s}{s^2 + 1} = \frac{0.5 - 1i}{s - 1i} + \frac{0.5 + 1i}{s + 1i} + 2s^2 - 2.$$

Find the partial fraction expansion of *F*(*s*) using `residue`. When the degree of the numerator is greater than the degree of the denominator, output `k` is a vector that represents the coefficients of a polynomial in *s*.

```
b = [2 0 0 1 0];
a = [1 0 1];
[r,p,k] = residue(b,a)
```

```
r =

   0.5000 - 1.0000i
   0.5000 + 1.0000i


p =

   0.0000 + 1.0000i
   0.0000 - 1.0000i


k =

     2     0    -2
```

`k` represents the polynomial $2s^2 - 2$.

- "Partial Fraction Expansion"

Input Arguments

b — Coefficients of numerator polynomial

vector of numbers

Coefficients of the polynomial in the numerator, specified as a vector of numbers representing the coefficients of the polynomial in descending powers of *s*.

Data Types: `single | double`
Complex Number Support: Yes

a — Coefficients of denominator polynomial

vector of numbers

Coefficients of the polynomial in the denominator, specified as a vector of numbers representing the coefficients of the polynomial in descending powers of *s*.

Data Types: `single | double`
Complex Number Support: Yes

Output Arguments

r — Residues of partial fraction expansion

column vector of numbers

Residues of partial fraction expansion, returned as a column vector of numbers.

p — Poles of partial fraction expansion

column vector of numbers

Poles of partial fraction expansion, returned as a column vector of numbers.

k — Direct term

row vector of numbers

Direct term, returned as a row vector of numbers that specify the coefficients of the polynomial in descending powers of *s*.

More About

Partial Fraction Expansion

Consider the fraction $F(s)$ of two polynomials b and a of degree n and m, respectively

$$F(s) = \frac{b(s)}{a(s)} = \frac{b_n s^n + \ldots + b_2 s^2 + b_1 s + b_0}{a_m s^m + \ldots + a_2 s^2 + a_1 s + a_0}.$$

The fraction $F(s)$ can be represented as a sum of simple fractions

$$\frac{b(s)}{a(s)} = \frac{r_1}{s - p_1} + \frac{r_2}{s - p_2} + \ldots + \frac{r_n}{s - p_n} + k(s)$$

This sum is called the partial fraction expansion of F. The values $r_m,\ldots,r_1$ are the residues, the values $p_m,\ldots,p_1$ are the poles, and $k(s)$ is a polynomial in s. For most textbook problems, $k(s)$ is 0 or a constant.

The number of poles `n` is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term vector is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b)-length(a)+1
```

If `p(i) = ... = p(i+j-1)` is a pole of multiplicity `j`, then the expansion includes terms of the form

$$\frac{r_j}{s - p_j} + \frac{r_{j+1}}{(s - p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s - p_j)^m}.$$

Algorithms

`residue` first obtains the poles using `roots`. Next, if the fraction is nonproper, the direct term `k` is found using `deconv`, which performs polynomial long division. Finally, `residue` determines the residues by evaluating the polynomial with individual roots removed. For repeated roots, `resi2` computes the residues at the repeated root locations.

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, $a(s)$, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can result in arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

References

[1] Oppenheim, A.V. and R.W. Schafer. *Digital Signal Processing*. Prentice-Hall, 1975, p. 56.

See Also

`deconv` | `poly` | `roots`

Introduced before R2006a

restoredefaultpath

Restore search path to its factory-installed state

Alternatives

As an alternative to the `restoredefaultpath` function, use the `rmpath` function or use the `pathtool` function to open the Set Path dialog box.

Syntax

```
restoredefaultpath
```

Description

`restoredefaultpath` restores the MATLAB search path to its factory-installed state.

Note: `restoredefaultpath` is intended only for situations where MATLAB is experiencing startup problems due to a corrupt search path. For general search-path cleanup, use the `rmpath` function or the Set Path dialog box.

If MATLAB fails to initialize properly on startup, then call both `restoredefaultpath` and `matlabrc`. For more details, see "Path Unsuccessfully Set at Startup".

MATLAB does not support issuing `restoredefaultpath` from a UNC path name. Doing so might result in MATLAB being unable to find files on the search path. If you do use `restoredefaultpath` from a UNC path name, restore the expected behavior by changing the current folder to an absolute path, and then reissuing the `restoredefaultpath` command.

More About

- "Path Unsuccessfully Set at Startup"

- “What Is the MATLAB Search Path?”

See Also

`rmpath` | `pathtool` | `addpath` | `genpath` | `matlabrc` | `savepath`

Introduced before R2006a

rethrow

Reissue error

Note: As of version 7.5, MATLAB supports error handling that is based on the `MException` class. Calling `rethrow` with a structure argument, as described on this page, is now replaced by calling rethrow with an `MException` object, as described on the reference page for `MException.rethrow`. `rethrow` called with a structure input will be removed in a future version.

Syntax

```
rethrow(errorStruct)
```

Description

`rethrow(errorStruct)` reissues the error specified by `errorStruct`. The currently running function terminates and control returns to the keyboard (or to any enclosing `catch` block). The `errorStruct` argument must be a MATLAB structure containing at least the `message` and `identifier` fields:

Fieldname	Description
`message`	Text of the error message
`identifier`	Message identifier of the error message
`stack`	Information about the error from the program stack

See "Message Identifiers" in the MATLAB documentation for more information on the syntax and usage of message identifiers.

Examples

`rethrow` is usually used in conjunction with `try, catch` statements to reissue an error from a `catch` block after performing `catch`-related operations. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(previous_error)
end
```

More About

Tips

The `errorStruct` input can contain the field `stack`, identical in format to the output of the `dbstack` command. If the `stack` field is present, the stack of the rethrown error will be set to that value. Otherwise, the stack will be set to the line at which the rethrow occurs.

See Also

`MException.rethrow` | `MException.throwAsCaller` | `MException.throw` | `MException` | `assert` | `error` | `try, catch`

Introduced before R2006a

return

Return control to invoking function

Syntax

```
return
```

Description

`return` forces MATLAB to return control to the invoking function before it reaches the end of the function. The invoking function is the function that calls the script or function containing the call to `return`. If you call the function or script that contains `return` directly, there is no invoking function and MATLAB returns control to the command prompt.

Note: Be careful when you use `return` within conditional blocks, such as `if` or `switch`, or within loop control statements, such as `for` or `while`. When MATLAB reaches a `return` statement, it does not just exit the loop; it exits the script or function and returns control to the invoking function or command prompt.

Examples

Return Control to Keyboard

In your current working folder, create a function, `findSqrRootIndex`, to find the index of the first occurrence of the square root of a value within an array. If the square root isn't found, the function returns `NaN`.

```
function idx = findSqrRootIndex(target,arrayToSearch)

idx = NaN;
if target < 0
   return
```

```
end

for idx = 1:length(arrayToSearch)
    if arrayToSearch(idx) == sqrt(target)
        return
    end
end
```

At the command prompt, call the function.

```
A = [3 7 28 14 42 9 0];
b = 81;
findSqrRootIndex(b,A)

ans =

     6
```

When MATLAB encounters the `return` statement, it returns control to the keyboard because there is no invoking function.

Return Control to Invoking Function

In a file, `myFunction.m`, in your current working folder, create the following function to find the index of the first occurrence of the square root of a value within an array. This function calls the `findSqrRootIndex` function you created in the previous example.

```
function myFunction(target)

arrayToSearch = [3 7 28 14 42 9 0];
idx = findSqrRootIndex(target,arrayToSearch);

if isnan(idx)
    disp('Square root not found.')
else
    disp(['Square root found at index ' num2str(idx)])
end
```

At the command prompt, call the function.

```
myFunction(49)

Square root found at index 2
```

When MATLAB encounters the `return` statement within `findSqrRootIndex`, it returns control to the invoking function, `myFunction`, and displays the relevant message.

See Also

`break` | `continue` | `disp` | `end` | `error` | `for` | `if` | `keyboard` | `switch` | `while`

Introduced before R2006a

rewriteDirectory

Class: Tiff

Write modified metadata to existing IFD

Syntax

```
rewriteDirectory(tiffobj)
```

Description

`rewriteDirectory(tiffobj)` writes modified metadata (tag) data to an existing directory. Use this tag when you want to change the value of a tag in an existing image file directory.

Examples

Modify Value of Tag

Write a sample TIFF file, `mytif.tif`. Create a TIFF object associated with this file.

```
imdata = peaks(256);
imwrite(imdata,'mytif.tif');
t = Tiff('mytif.tif','r+');
```

Modify the value of a tag.

```
setTag(t,'Software','MATLAB')
rewriteDirectory(t)
close(t);
```

References

This method corresponds to the `TIFFRewriteDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.writeDirectory`

rgb2gray

Convert RGB image or colormap to grayscale

Syntax

```
I = rgb2gray(RGB)
newmap = rgb2gray(map)
```

Description

`I = rgb2gray(RGB)` converts the truecolor image RGB to the grayscale intensity image I. The `rgb2gray` function converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance. If you have Parallel Computing Toolbox installed, `rgb2gray` can perform this conversion on a GPU.

`newmap = rgb2gray(map)` returns a grayscale colormap equivalent to map.

Examples

Convert RGB Image to Grayscale Image

Read and display an RGB image, and then convert it to grayscale.

Read the sample file, `peppers.png`, and display the RGB image.

```
RGB = imread('peppers.png');
imshow(RGB)
```

Convert the RGB image to a grayscale image and display it.

```
I = rgb2gray(RGB);
figure
imshow(I)
```

Convert RGB Colormap to Grayscale Colormap

Read an indexed image with an RGB colormap. Then, convert the colormap to grayscale.

Read the sample file, `corn.tif`, which is an indexed image with an RGB colormap.

```
[X,map] = imread('corn.tif');
```

Display the image.

```
imshow(X,map)
```

Convert the RGB colormap to a grayscale colormap and redisplay the image.

```
newmap = rgb2gray(map);
imshow(X,newmap)
```

Input Arguments

RGB — Truecolor image

3-D numeric array

Truecolor image, specified as 3-D numeric array.

If you have Parallel Computing Toolbox installed, RGB can also be a gpuArray.

Data Types: `single` | `double` | `uint8` | `uint16`

`map` — Colormap
m-by-3 numeric array

Colormap, specified as an *m*-by-3 numeric array.

If you have Parallel Computing Toolbox installed, map can also be a gpuArray.

Data Types: `double`

Output Arguments

`I` — grayscale image
numeric array

Grayscale image, returned as a numeric array.

If you have Parallel Computing Toolbox installed, then I can also be a gpuArray.

`newmap` — grayscale color map
m-by-3 numeric array

Grayscale colormap, returned as an *m*-by-3 numeric array.

If you have Parallel Computing Toolbox installed, then newmap can also be a gpuArray.

More About

Tips

- `rgb2gray` supports the generation of C code using MATLAB Coder.

Algorithms

`rgb2gray` converts RGB values to grayscale values by forming a weighted sum of the *R*, *G*, and *B* components:

```
0.2989 * R + 0.5870 * G + 0.1140 * B
```

These are the same weights used by the `rgb2ntsc` function to compute the *Y* component.

See Also

`ind2gray` | `mat2gray` | `ntsc2rgb` | `rgb2ind` | `rgb2ntsc`

rgb2hsv

Convert RGB colormap to HSV colormap

Syntax

```
cmap = rgb2hsv(M)
hsv_image = rgb2hsv(rgb_image)
```

Description

`cmap = rgb2hsv(M)` converts an RGB colormap `M` to an HSV colormap `cmap`. Both colormaps are *m*-by-3 matrices. The elements of both colormaps are in the range 0 to 1.

The columns of the input matrix `M` represent intensities of red, green, and blue, respectively. The columns of the output matrix `cmap` represent hue, saturation, and value, respectively.

`hsv_image = rgb2hsv(rgb_image)` converts the RGB image to the equivalent HSV image. RGB is an m-by-n-by-3 image array whose three planes contain the red, green, and blue components for the image. HSV is returned as an m-by-n-by-3 image array whose three planes contain the hue, saturation, and value components for the image.

See Also

`brighten` | `colormap` | `hsv2rgb` | `rgbplot`

Introduced before R2006a

rgb2ind

Convert RGB image to indexed image

Syntax

```
[X,map] = rgb2ind(RGB,n)
X = rgb2ind(RGB, map)
[X,map] = rgb2ind(RGB, tol)
[ ___ ] = rgb2ind( ___ ,dither_option)
```

Description

`[X,map] = rgb2ind(RGB,n)` converts the RGB image to an indexed image `X` using minimum variance quantization and dithering. `map` contains at most `n` colors. `n` must be less than or equal to 65,536.

`X = rgb2ind(RGB, map)` converts the RGB image to an indexed image `X` with colormap `map` using the inverse colormap algorithm and dithering. `size(map,1)` must be less than or equal to 65,536.

`[X,map] = rgb2ind(RGB, tol)` converts the RGB image to an indexed image `X` using uniform quantization and dithering. `map` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0.0 and 1.0.

`[ ___ ] = rgb2ind( ___ ,dither_option)` enables or disables dithering. `dither_option` is a string that can have one of these values.

`'dither'` (default)	Dithers, if necessary, to achieve better color resolution at the expense of spatial resolution
`'nodither'`	Maps each color in the original image to the closest color in the new map. No dithering is performed.

Note The values in the resultant image `X` are indexes into the colormap `map` and should not be used in mathematical processing, such as filtering operations.

Class Support

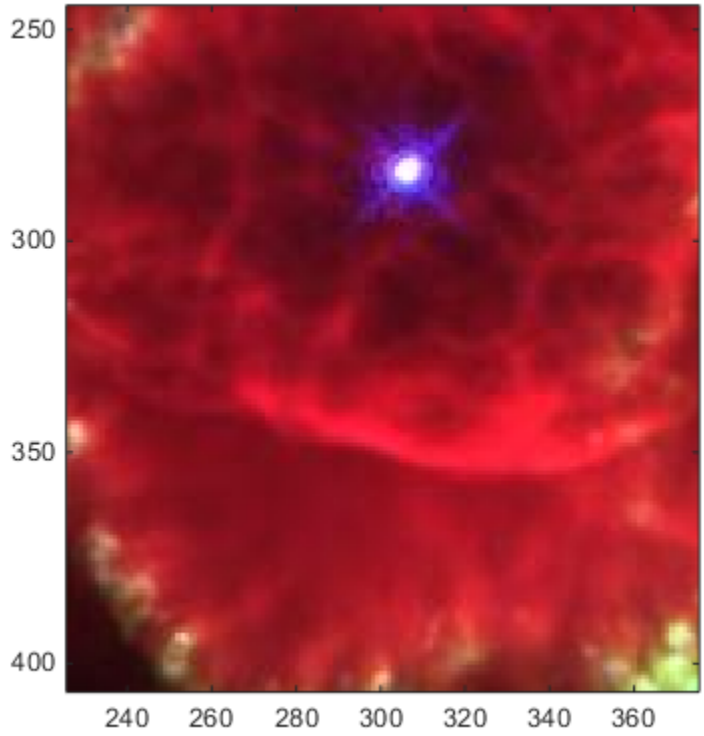
The input image can be of class `uint8`, `uint16`, `single`, or `double`. If the length of `map` is less than or equal to 256, the output image is of class `uint8`. Otherwise, the output image is of class `uint16`.

The value `0` in the output array `X` corresponds to the first color in the colormap.

Examples

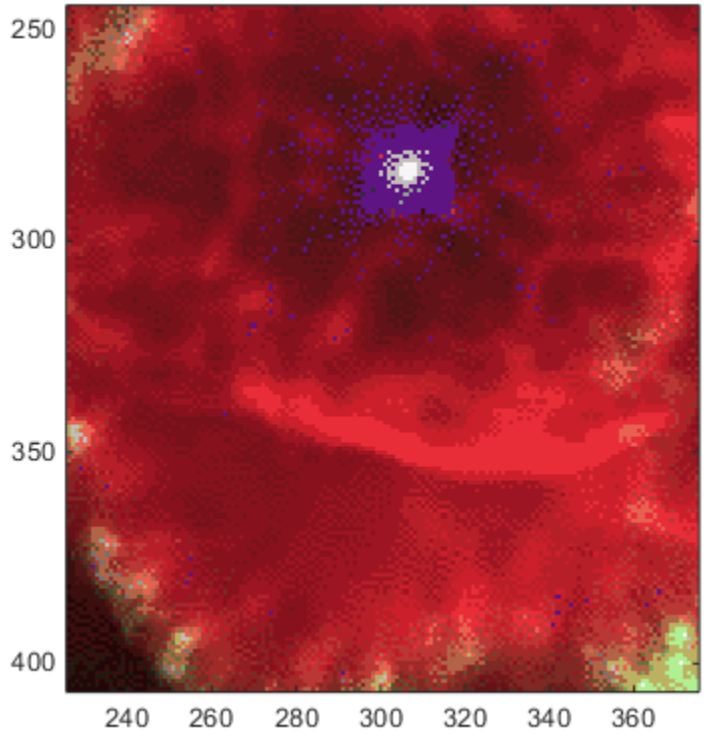
Read and display a truecolor `uint8` JPEG image of a nebula.

```
RGB = imread('ngc6543a.jpg');
figure('Name','RGB Image')
imagesc(RGB)
axis image
zoom(4)
```



Convert RGB to an indexed image with 32 colors

```
[IND,map] = rgb2ind(RGB,32);
figure('Name','Indexed image with 32 Colors')
imagesc(IND)
colormap(map)
axis image
zoom(4)
```



More About

Tips

- If you specify `tol`, `rgb2ind` uses uniform quantization to convert the image. This method involves cutting the RGB color cube into smaller cubes of length `tol`.
- If you specify `n`, `rgb2ind` uses minimum variance quantization. This method involves cutting the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number you specify, the output colormap is also smaller.

- If you specify `map`, `rgb2ind` uses colormap mapping, which involves finding the colors in `map` that best match the colors in the RGB image.

Algorithms

- **Uniform Quantization** — Uniform quantization cuts the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is:

  ```
  n = (floor(1/tol)+1)^3
  ```

  Each cube represents a single color in the output image. Therefore, the maximum length of the colormap is `n`. `rgb2ind` removes any colors that don't appear in the input image, so the actual colormap can be much smaller than `n`.
- **Minimum Variance Quantization** — Minimum variance quantization cuts the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number specified, the output colormap is also smaller.
- **Inverse Colormap** — The inverse colormap algorithm quantizes the specified colormap into 32 distinct levels per color component. Then, for each pixel in the input image, the closest color in the quantized colormap is found.

References

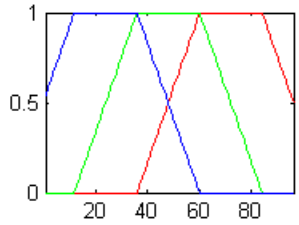
[1] Spencer W. Thomas, "Efficient Inverse Color Map Computation", *Graphics Gems II*, (ed. James Arvo), Academic Press: Boston. 1991. (includes source code)

See Also

`cmunique` | `dither` | `imapprox` | `ind2rgb`

rgbplot

Plot colormap



Syntax

```
rgbplot(cmap)
```

Description

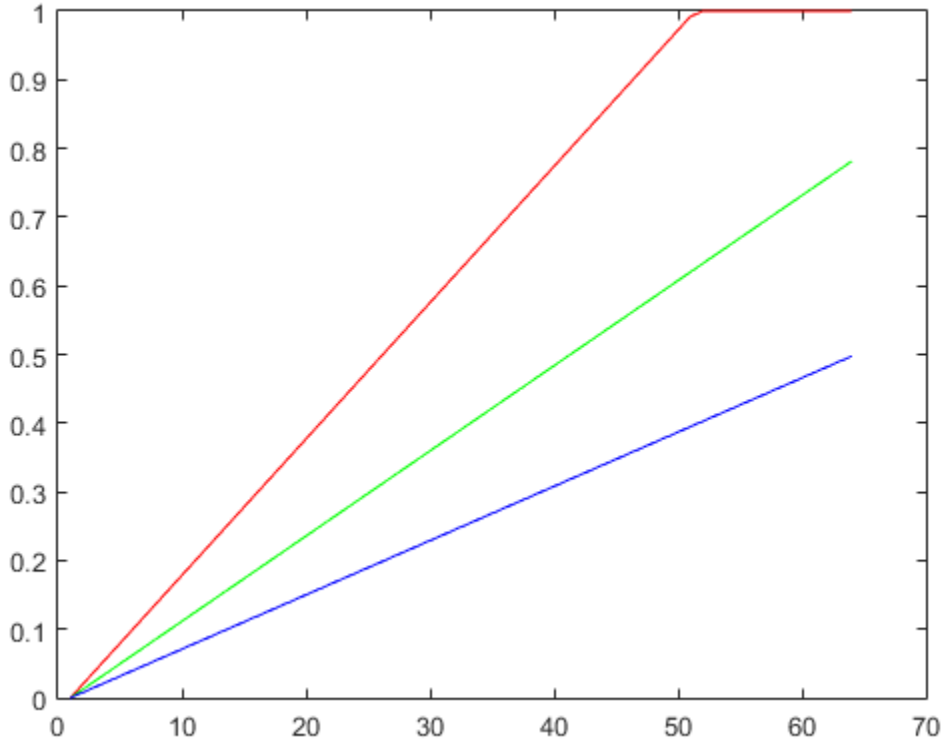
`rgbplot(cmap)` plots the three columns of `cmap`, where `cmap` is an *m*-by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

Examples

Copper Colormap

Plot the RGB values of the `copper` colormap.

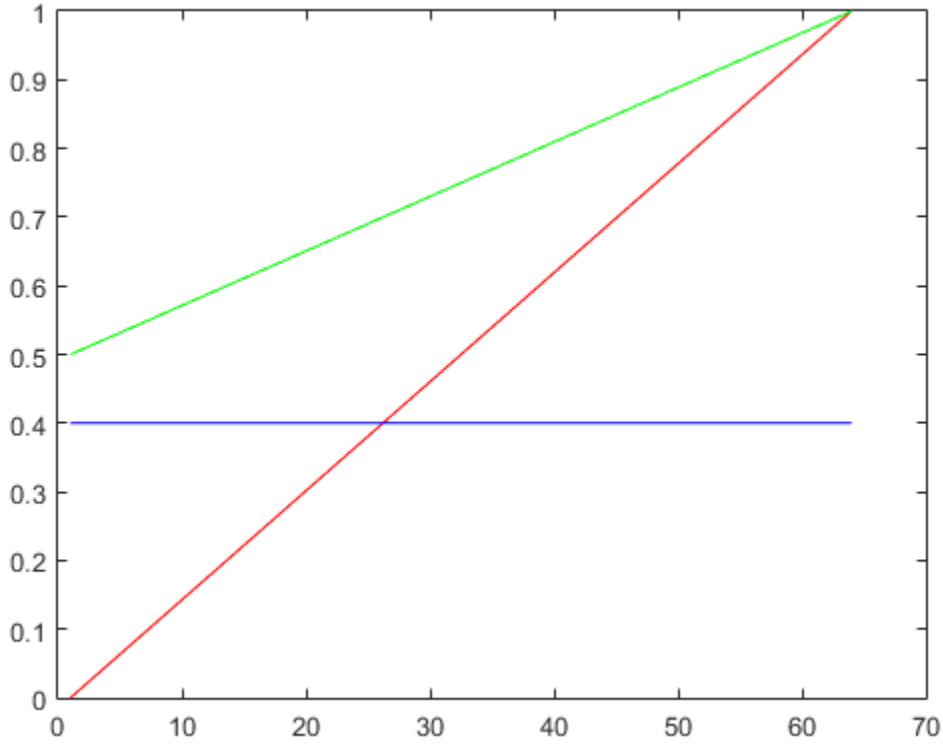
```
rgbplot(copper)
```



Summer Colormap

Plot the RGB values of the `summer` colormap.

```
rgbplot(summer)
```



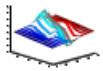
See Also

`colormap`

Introduced before R2006a

ribbon

Ribbon plot



Syntax

```
ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
ribbon(axes_handle,...)
h = ribbon(...)
```

Description

`ribbon(Y)` plots the columns of `Y` as three-dimensional ribbons of uniform width using `X = 1:size(Y,1)`. Ribbons advance along the *x*-axis centered on tick marks at unit intervals, three-quarters of a unit in width. Ribbon maps values in `X` to colors in colormap linearly. To change ribbon colors in the graph, change the `colormap`.

`ribbon(X,Y)` plots three dimensional ribbons for data in `Y`, centered at locations specified in `X`. `X` and `Y` are vectors or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows. When `Y` is a matrix, `ribbon` plots each column in `Y` as a ribbon at the corresponding `X` location.

`ribbon(X,Y,width)` specifies the width of the ribbons. The default is `0.75`. If `width = 1`, the ribbons touch, leaving no space between them when viewed down the *z*-axis. If `width > 1`, ribbons overlap and can intersect.

`ribbon(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

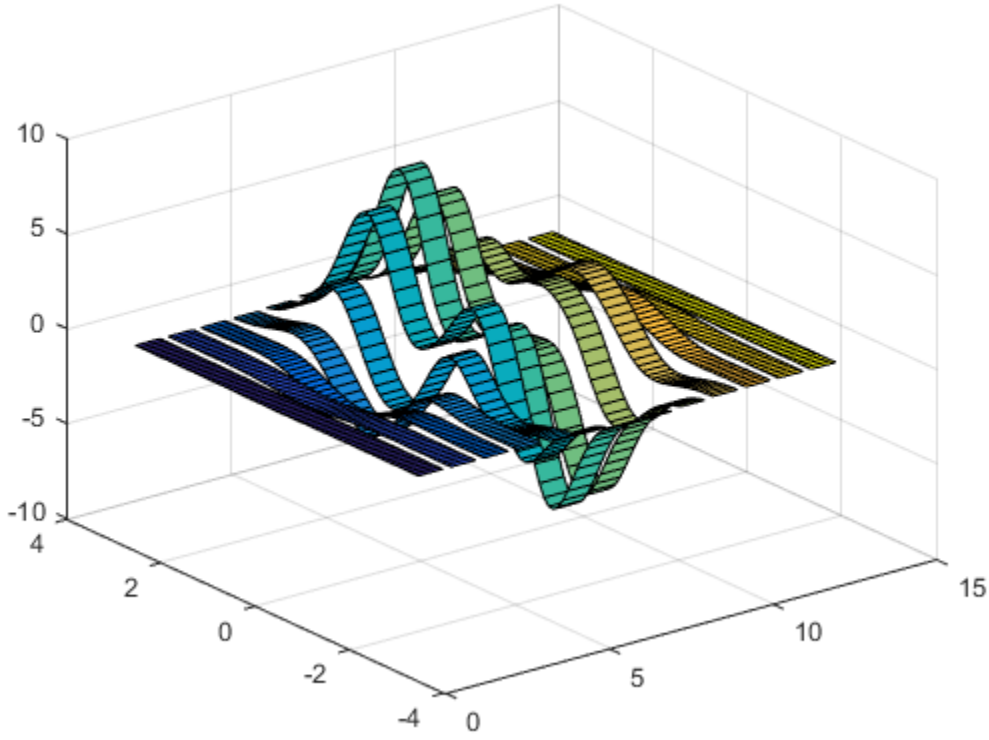
`h = ribbon(...)` returns a vector of handles to surface graphics objects. `ribbon` returns one handle per strip.

Examples

Ribbon Plot

Create a ribbon plot of the `peaks` function.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
z = peaks(x,y);

figure
ribbon(y,z)
```



See Also

`plot` | `plot3` | `surface` | `waterfall`

Introduced before R2006a

rmappdata

Remove application-defined data

Syntax

```
rmappdata(h,name)
```

Description

`rmappdata(h,name)` removes the application-defined data, `name`, from its association with the UI component, `h`.

See Also

`getappdata` | `isappdata` | `setappdata`

Introduced before R2006a

rmdir

Remove folder

Syntax

```
rmdir(folderName)
rmdir(folderName,'s')
[status, message, messageid] = rmdir(folderName,'s')
```

Description

`rmdir(folderName)` removes the folder folderName from the current folder if folderName is empty. If folderName is not in the current folder, then specify the relative path or the full path for folderName.

`rmdir(folderName,'s')` removes the folder `folderName` and its contents from the current folder. With the `'s'` option, `rmdir` attempts to remove all subfolders and files in folderName regardless of their write permissions.

`[status, message, messageid] = rmdir(folderName,'s')` removes the folder folderName and its contents from the current folder, returning the status, a message, and the MATLAB message ID.

Input Arguments

`folderName`

String specifying the absolute or relative path name of the folder you want to remove.

Default: None

`'s'`

Literal string that directs `rmdir` to remove all subfolders and files in the specified folder, regardless of their write permissions. The result for read-only files follows the practices of the operating system.

Default: `rmdir` does not remove subfolders and files in the specified folder.

Output Arguments

`status`

Logical scalar indicating the outcome of the `rmdir` operation. The status value is `1` if the operation was successful and `0` if it returned an error.

`message`

String containing the warning or error message text if the operation is unsuccessful. An empty string, if the operation is successful.

`messageid`

String containing the warning or error message ID, if the operation is unsuccessful. MATLAB returns an empty string if the operation is successful.

Examples

These examples remove an empty folder, `myfiles`, assuming it is in the current folder:

```
% Remove myfiles from the current folder:

rmdir('myfiles')

% Use the relative path to remove myfiles. Assuming
% the current folder is matlab/work and myfiles is in
% d:/matlab/work/project, type this:

rmdir('project/myfiles')

% Use the full path to remove myfiles, assuming
% the current folder is matlab/work and myfiles is in
% d:/matlab/work/project:

rmdir('d:/matlab/work/project/myfiles')
```

This example removes the `myfiles` folder and its contents, assuming `myfiles` is in the current folder:

```
rmdir('myfiles','s')
```

This example unsuccessfully attempts to remove the `myfiles` folder and its contents. It directs MATLAB to display the results.

```
[stat, mess, id]=rmdir('myfiles')
```

MATLAB returns:

```
stat =
     0

mess =

No directories were removed.

id =

MATLAB:RMDIR:NoDirectoriesRemoved
```

This example successfully removes the `myfiles` folder and its contents. It directs MATLAB to display the results.

```
[stat, mess]=rmdir('myfiles','s')
```

MATLAB returns:

```
stat =
     1

mess =

     ''
```

Alternatives

Open the Current Folder browser by running `filebrowser`. Then, in the Current Folder browser, right-click the folder name and select **Delete** from the context menu.

More About

Tips

- If you specify the `'s'` flag or include a wildcard in the folder name, MATLAB produces an error if it is unable to remove all folders. The error message lists the folder and files that MATLAB could not remove.

See Also

`| cd | copyfile | delete | dir | fileattrib | filebrowser | mkdir | movefile`

Introduced before R2006a

rmdir

Class: FTP

Remove folder on FTP server

Syntax

```
rmdir(ftpobj,folder)
```

Description

`rmdir(ftpobj,folder)` removes the specified folder from the current folder on an FTP server.

Input Arguments

`ftpobj`

FTP object created by `ftp`.

`folder`

String enclosed in single quotation marks that specifies the name of the folder to delete.

Examples

Remove the folder `temp` from the hypothetical FTP server `ftp.testsite.com`:

```
test = ftp('ftp.testsite.com');
rmdir(test, 'temp');
```

See Also

`delete` | `dir` | `mkdir` | `cd` | `ftp`

Introduced before R2006a

rmfield

Remove fields from structure

Syntax

```
s = rmfield(s,field)
```

Description

`s = rmfield(s,field)` removes the specified field or fields from structure array `s`. Specify multiple fields using a cell array of strings. The dimensions of `s` remain the same.

Examples

Remove Single Field

Define a scalar structure with fields named `a`, `b`, and `c`.

```
s.a = 1;
s.b = 2;
s.c = 3;
```

Remove field `b`.

```
field = 'b';
s = rmfield(s,field)
```

```
s =
    a: 1
    c: 3
```

Remove Multiple Fields

Define a scalar structure with fields `first`, `second`, `third`, and `fourth`.

```
S.first = 1;
S.second = 2;
```

```
S.third = 3;
S.fourth = 4;
```

Remove fields `first` and `fourth`.

```
fields = {'first','fourth'};
S = rmfield(S,fields)
```

```
S =
    second: 2
     third: 3
```

Input Arguments

s — Input structure

structure array

Input structure, specified as a structure array.

Data Types: `struct`

field — Field name or names

character array | cell array of strings

Field name or names, specified as a character array or a cell array of strings.

Example: `'f1'`

Example: `{'f1';'f2'}`

Data Types: `char` | `cell`

More About

- "Generate Field Names from Variables"

See Also

`fieldnames` | `isfield` | `orderfields`

Introduced before R2006a

rmpath

Remove folders from search path

Syntax

```
rmpath(folderName)
```

Description

`rmpath(folderName)` removes the specified folder from the search path.

Examples

Remove Folder from Search Path

Remove `/usr/local/matlab/mytools` from the search path.

```
rmpath('/usr/local/matlab/mytools')
```

Input Arguments

`folderName` — Name of folder

string

Name of folder to remove from the search path, specified as a string. Use the full path name for `folderName`.

Example: `'c:\matlab\work'`

Example: `'/home/user/matlab'`

More About

- "What Is the MATLAB Search Path?"

See Also

`addpath` | `path` | `savepath`

Introduced before R2006a

rmpref

Remove preference

Syntax

```
rmpref('group','pref')
rmpref('group',{'pref1','pref2',...'prefn'})
rmpref('group')
```

Description

`rmpref('group','pref')` removes the preference specified by `group` and `pref`. It is an error to remove a preference that does not exist.

`rmpref('group',{'pref1','pref2',...'prefn'})` removes each preference specified in the cell array of preference names. It is an error if any of the preferences do not exist.

`rmpref('group')` removes all the preferences for the specified `group`. It is an error to remove a group that does not exist.

Examples

```
addpref('mytoolbox','version','1.0')
rmpref('mytoolbox')
```

See Also

`addpref` | `ispref` | `getpref` | `setpref` | `uigetpref` | `uisetpref`

Introduced before R2006a

rng

Control random number generation

Syntax

```
rng(seed)
rng('shuffle')
rng(seed, generator)
rng('shuffle', generator)
rng('default')
scurr = rng
rng(s)
sprev = rng(...)
```

Description

Note: To use the `rng` function instead of `rand` or `randn` with the `'seed'`, `'state'`, or `'twister'` inputs, see the documentation on " Replace Discouraged Syntaxes of rand and randn".

`rng(seed)` seeds the random number generator using the nonnegative integer `seed` so that `rand`, `randi`, and `randn` produce a predictable sequence of numbers.

`rng('shuffle')` seeds the random number generator based on the current time. Thus, `rand`, `randi`, and `randn` produce a different sequence of numbers after each time you call `rng`.

`rng(seed, generator)` and `rng('shuffle', generator)` additionally specify the type of the random number generator used by `rand`, `randi`, and `randn`. The `generator` input is one of:

- `'twister'`: Mersenne Twister
- `'simdTwister'`: SIMD-oriented Fast Mersenne Twister
- `'combRecursive'`: Combined Multiple Recursive

- `'multFibonacci'`: Multiplicative Lagged Fibonacci
- `'v5uniform'`: Legacy MATLAB 5.0 uniform generator
- `'v5normal'`: Legacy MATLAB 5.0 normal generator
- `'v4'`: Legacy MATLAB 4.0 generator

`rng('default')` puts the settings of the random number generator used by `rand`, `randi`, and `randn` to their default values. This way, the same random numbers are produced as if you restarted MATLAB. The default settings are the Mersenne Twister with seed 0.

`scurr = rng` returns the current settings of the random number generator used by `rand`, `randi`, and `randn`. The settings are returned in a structure `scurr` with fields `'Type'`, `'Seed'`, and `'State'`.

`rng(s)` restores the settings of the random number generator used by `rand`, `randi`, and `randn` back to the values captured previously with a command such as `s = rng`.

`sprev = rng(...)` returns the previous settings of the random number generator used by `rand`, `randi`, and `randn` before changing the settings.

Examples

Example 1 — Retrieve and Restore Generator Settings

Save the current generator settings in `s`:

```
s = rng;
```

Call `rand` to generate a vector of random values:

```
x = rand(1,5)
```

```
x =

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Restore the original generator settings by calling `rng`. Generate a new set of random values and verify that `x` and `y` are equal:

```
rng(s);
```

```
y = rand(1,5)

y =

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Example 2 — Restore Settings for Legacy Generator

Use the legacy generator.

```
sprev = rng(0,'v5uniform')

sprev =
     Type: 'twister'
     Seed: 0
    State: [625x1 uint32]

x = rand

x =

    0.9501
```

Restore the previous settings by calling `rng`:

```
rng(sprev)
```

More About

- "Creating and Controlling a Random Number Stream"
- "Why Do Random Numbers Repeat After Startup?"

See Also

`rand` | `randi` | `randn` | `RandStream` | `now`

Root Properties

Graphics environment and state information

The root object is the root of the graphics object tree. Root properties contain information about the graphics environment and the current state of the graphics system. Starting in R2014b, you can use dot notation to refer to a particular object and property:

```
r = groot;
fig = r.Children;
```

If you are using an earlier release, use the `get` function to query property values.

Display Information

`MonitorPositions` — Width and height of displays

n-by-4 matrix

Width and height of displays, returned as an n-by-4 matrix, where n is the number of displays. Each row corresponds to one display and is a four-element vector of the form `[x y width height]`. For example, if there are two displays, then the matrix has this form:

```
[x1 y1 width1 height1
 x2 y2 width2 height2]
```

The first two elements in each row indicate the display location with respect to the origin point. The last two elements in each row indicate the display size. The origin point is the lower-left corner of the primary display. If the units are pixels, then the origin point is `(1,1)`. For all other units, the origin point is `(0,0)`. The `Units` property determines the units of this measurement.

Note: MATLAB sets the display information values for this property at startup. The values are static. If your system display settings change, the values do not update. To refresh the values, restart MATLAB.

`PointerLocation` — Current location of pointer

two-element vector

Current location of pointer, specified as a two-element vector of the form `[x y]`. The `x` and `y` values are the coordinates of the pointer position measured from the origin point. The origin point is the lower-left corner of the primary display. If the units are pixels, then the origin point is `(1,1)`. For all other units, the origin point is `(0,0)`. The `Units` property determines the units of this measurement.

This property contains the current pointer location, even if the pointer is outside a MATLAB window. Move the pointer by changing the values of this property. On Macintosh systems, you cannot change the pointer location by setting this property.

Querying the `PointerLocation` property in a callback routine might return a value that is different from the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

Example: `[500 400]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ScreenDepth` — Number of bits that define each pixel color

scalar

Number of bits that define each pixel color, specified as a scalar. The default value depends on the computer. The maximum number of simultaneously displayed colors on the current graphics device equals 2 raised to the value of this property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ScreenPixelsPerInch` — Display resolution

scalar

Display resolution, specified as a scalar in pixels per inch. The default value depends on the computer. The value is the setting of the display resolution specified in your system preferences.

Example: `72`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ScreenSize` — Size of primary display

four-element vector

Size of primary display, returned as a four-element vector of the form `[left bottom width height]` that defines the display size. If the units are pixels, then the first two elements are both `1`. For all other units, the first two elements are both `0`. The last two elements are the display dimensions in units specified by the `Units` property.

The values might not represent the usable display size due to the presence of UIs, such as the Microsoft Windows task bar.

Note: MATLAB sets the display size values for this property at startup. The values are static. If your system display settings change, the display size values do not update. To refresh the values, restart MATLAB.

`FixedWidthFontName` — Font name for fixed-width font

string

Font name for fixed-width font, specified as a string giving the name of a system supported font. This property determines the font for axes, text, and uicontrols that have a `FontName` property set to `'FixedWidth'`. The default value depends on the system. `'Courier'` is the default in systems that use Latin-based characters.

Specifying the `FixedWidthFontName` property eliminates the need to hardcode font names in MATLAB applications. These applications can run without modification in systems that require non-ASCII character sets. In these cases, MATLAB attempts to set `FixedWidthFontName` property to the correct value for the system.

If you are a MATLAB application developer and want to use a fixed-width font, set the `FontName` property for axes, text, and uicontrol objects to `'FixedWidth'` instead of setting this root property. Users of the application can set the root property if they do not want to use the preselected value.

Example: `'Courier New'`

`Units` — Units for `MonitorPositions`, `ScreenSize`, and `PointerLocation`

`'pixels'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'characters'` | `'normalized'`

Units for the `MonitorPositions`, `ScreenSize`, and `PointerLocation` properties, specified as one of the values shown in this table.

Units	Description
`'pixels'` (default)	Pixels. Pixel size depends on the display resolution.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72 inch.
`'normalized'`	Normalized with respect to the display. The lower-left corner of the display maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.

All units are measured from the lower-left corner of the primary display. If the units are pixels, then the lower-left corner maps to `(1,1)`. For all other units, the lower-left corner maps to `(0,0)`.

If you change the units, it is good practice to return it to its default value after completing your operation to prevent affecting other functions that assume the `Units` property is set to the default value.

Identifiers

`CallbackObject` — Object whose callback is executing

`[ ]` (default) | graphics object

Object whose callback is executing, returned as a graphics object. For more information, see the `gcbo` command.

`CurrentFigure` — Current figure

empty `GraphicsPlaceholder` (default) | figure object

Current figure, specified as a figure object. The current figure is typically the one most recently created, clicked on, or made current by calling the `figure` function. Setting

this property makes a figure the current figure without sorting it to the front of other figures on the display. However, using the `figure` function to make a figure the current figure sorts that figure to the front of the display. To become the current figure, the `HandleVisibility` property of the figure must be set to `'on'`.

This property returns an empty `GraphicsPlaceholder` array if there are no figures. However, the `gcf` command always returns a figure object. If there are no figure objects, then `gcf` creates one.

`Type` — Type of graphics object

`'root'`

Type of graphics object, returned as the string `'root'`. The root object handle is always visible using the `groot` function.

`Tag` — Tag to associate with root

`''` (default)

Tag to associate with root, specified as a string. There is only one root object, which you can always access using the `groot` function.

`UserData` — Data to associate with root

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the root object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent

empty `GraphicsPlaceholder`

The root object has no parent. This property is always an empty `GraphicsPlaceholder`.

`Children` — Children

empty `GraphicsPlaceholder` | array of figure objects

Children, specified as an array of figure objects that have visible handles. The `HandleVisibility` property of the figure determines if the handle is visible or hidden. This property does not contain figures with hidden handles.

Change the order of the children to change the sorting order of the figures on the display.

`HandleVisibility` — Visibility of root object handle

`'on'` (default) | `'callback'` | `'off'`

This property has no effect. The root object handle is always visible using the `groot` function.

`ShowHiddenHandles` — Hidden handle display

`'off'` (default) | `'on'`

Hidden handle display, specified as one of these values:

- `'off'` — Do not display hidden object handles. The `HandleVisibility` property of the object determines if the handle is visible or hidden.
- `'on'` — Expose all object handles regardless of the `HandleVisibility` property.

See Also

`groot`

More About

- "Access Property Values"
- "Graphics Object Properties"

roots

Polynomial roots

The `roots` function solves polynomial equations of the form $p_1x^n + ... + p_nx + p_{n+1} = 0$. Polynomial equations contain a single variable with nonnegative exponents. To find the roots of other types of equations, use `fzero`.

Syntax

```
r = roots(p)
```

Description

`r = roots(p)` returns the roots of the polynomial represented by `p` as a column vector.

`p` is a vector containing `n+1` polynomial coefficients, starting with the coefficient of x^n. A coefficient of `0` indicates an intermediate power that is not present in the equation. For example, `p = [3 2 -2]` represents the polynomial $3x^2 + 2x - 2$.

Examples

Roots of Quadratic Polynomial

Solve the equation $3x^2 - 2x - 4 = 0$.

Create a vector to represent the polynomial, then find the roots.

```
p = [3 -2 -4];
r = roots(p)
```

```
r =

    1.5352
   -0.8685
```

Roots of Quartic Polynomial

Solve the equation $x^4 - 1 = 0$.

Create a vector to represent the polynomial, then find the roots.

```
p = [1 0 0 0 -1];
r = roots(p)
```

```
r =

  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Input Arguments

p — Polynomial coefficients

vector

Polynomial coefficients, specified as a vector. For example, the vector `[1 0 1]` represents the polynomial $x^2 + 1$, and the vector `[3.13 -2.21 5.99]` represents the polynomial $3.13x^2 - 2.21x + 5.99$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

More About

Tips

- Use the `poly` function to obtain a polynomial from its roots: `p = poly(r)`. The `poly` function is the inverse of the `roots` function.

Algorithms

The `roots` function considers `p` to be a vector with `n+1` elements representing the `n`th degree characteristic polynomial of an `n`-by-`n` matrix, `A`. The roots of the polynomial are calculated by computing the eigenvalues of the companion matrix, `A`.

```
A = diag(ones(n-1,1),-1);
A(1,:) = -p(2:n+1)./p(1);
r = eig(A)
```

The results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix, `A`. However, this does not mean that they are the exact roots of a polynomial whose coefficients are within roundoff error of those in `p`.

See Also

`fzero` | `poly` | `residue`

Introduced before R2006a

rose

Angle histogram plot

Syntax

```
rose(theta)
rose(theta,x)
rose(theta,nbins)
rose(axes_handle,...)
h = rose(...)
[tout,rout] = rose(...)
```

Description

`rose(theta)` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range, showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle of each bin from the origin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0,2*pi]`. The default is `20`.

`rose(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = rose(...)` returns the handle of the line object used to create the graph.

`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

Examples

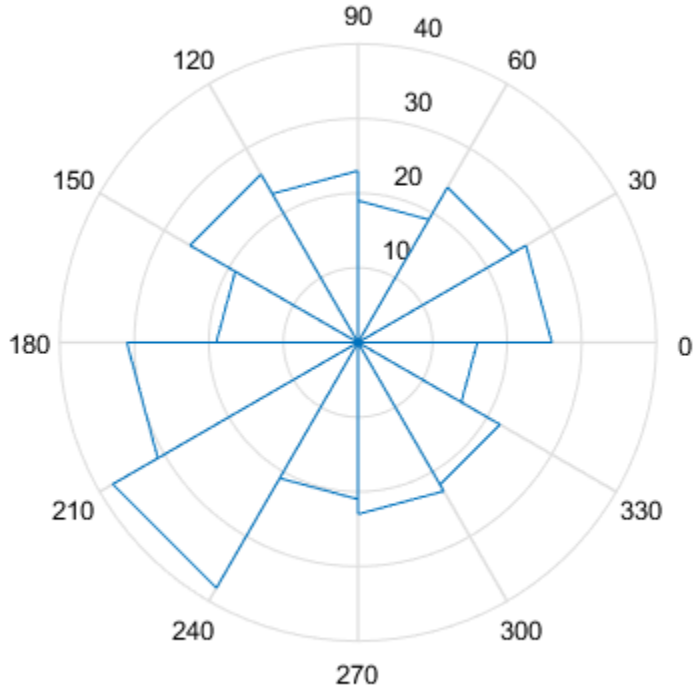
Create Rose Histogram

Load the `sunspot.dat` data set which contains the 2-column matrix `sunspot`. Store the second column of the data set as `theta`.

```
load sunspot.dat
theta = sunspot(:,2);
```

Create a rose histogram of `theta` using 12 bins.

```
figure
rose(theta,12)
```



See Also

`compass` | `feather` | `histogram` | `line` | `polar`

Introduced before R2006a

rosser

Classic symmetric eigenvalue test problem

Syntax

```
A = rosser
A = rosser(classname)
```

Description

`A = rosser` returns the Rosser matrix in double precision.

`A = rosser(classname)` returns the Rosser matrix with a class specified by classname. Specify classname as `'single'` to return the Rosser matrix in single precision.

Examples

Generate the Rosser matrix

`rosser` returns the Rosser matrix.

```
rosser
```

```
ans =
   611   196  -192   407    -8   -52   -49    29
   196   899   113  -192   -71   -43    -8   -44
  -192   113   899   196    61    49     8    52
   407  -192   196   611     8    44    59   -23
    -8   -71    61     8   411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99
```

Generate matrix of class 'single'

Specify classname as `single` to return a Rosser matrix of that class.

```
Y = rosser('single')
whos('Y')
```

```
Y =

   611   196  -192   407    -8   -52   -49    29
   196   899   113  -192   -71   -43    -8   -44
  -192   113   899   196    61    49     8    52
   407  -192   196   611     8    44    59   -23
    -8   -71    61     8   411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99

  Name      Size            Bytes  Class     Attributes

  Y         8x8               256  single
```

Input Arguments

`classname` — Input class

`'double'` (default) | `'single'`

Input class, specified as `'double'` (default) or `'single'`. `rosser(C)` produces a matrix of the specified class.

More About

Rosser Matrix

The Rosser matrix is a well known matrix used, for example, to evaluate eigenvalue algorithms. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of the opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

See Also

`eig`

Introduced before R2006a

rot90

Rotate array 90 degrees

Syntax

```
B = rot90(A)
B = rot90(A,k)
```

Description

`B = rot90(A)` rotates array `A` counterclockwise by 90 degrees. For multidimensional arrays, `rot90` rotates in the plane formed by the first and second dimensions.

`B = rot90(A,k)` rotates array `A` counterclockwise by `k*90` degrees, where `k` is an integer.

Examples

Rotate Column Vector

Create a column vector of sequential elements.

```
A = (1:5)'
```

```
A =

     1
     2
     3
     4
     5
```

Rotate `A` counterclockwise by 90 degrees using `rot90`.

```
B = rot90(A)
```

```
B =

     1     2     3     4     5
```

The result, B, has the same elements as A but a different orientation.

Rotate Multidimensional Array

Create a 3-by-3-by-2 cell array of strings.

```
A = cat(3,{'a' 'b' 'c';'d' 'e' 'f';'g' 'h' 'i'},{'j' 'k' 'l';'m' 'n' 'o';'p' 'q' 'r'})
```

```
A(:,:,1) =

    'a'    'b'    'c'
    'd'    'e'    'f'
    'g'    'h'    'i'


A(:,:,2) =

    'j'    'k'    'l'
    'm'    'n'    'o'
    'p'    'q'    'r'
```

Rotate the cell array by 270 degrees.

```
B = rot90(A,3)
```

```
B(:,:,1) =

    'g'    'd'    'a'
    'h'    'e'    'b'
    'i'    'f'    'c'


B(:,:,2) =

    'p'    'm'    'j'
    'q'    'n'    'k'
    'r'    'o'    'l'
```

The function rotates each page of the array independently. Since a full 360 degree rotation (`k = 4`) leaves the array unchanged, `rot90(A,3)` is equivalent to `rot90(A,-1)`.

Input Arguments

A — Input array

vector | matrix | array | cell array | categorical array

Input array, specified as a vector, matrix, array, cell array, or categorical array of any data type.
Complex Number Support: Yes

k — Rotation constant

integer

Rotation constant, specified as an integer. Specify `k` to rotate by `k*90` degrees rather than nesting calls to `rot90`.

Example: `rot90(A,-2)` rotates `A` by -180 degrees and is equivalent to `rot90(A,2)`, which rotates by 180 degrees.

More About

Tips

- Use the `flip` function to flip arrays in any dimension.

See Also

`flip` | `fliplr` | `flipud`

Introduced before R2006a

rotate

Rotate object about specified origin and direction

Syntax

```
rotate(h,direction,alpha)
rotate(...,origin)
```

Description

The `rotate` function rotates a graphics object in three-dimensional space.

`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin of the axis of rotation. The default origin of the axis of rotation is the center of the plot box. This point is not necessarily the origin of the axes.

Positive `alpha` is defined as the righthand-rule angle about the direction vector as it extends from the origin of rotation.

If `h` is an array of handles, all objects must be children of the same axes.

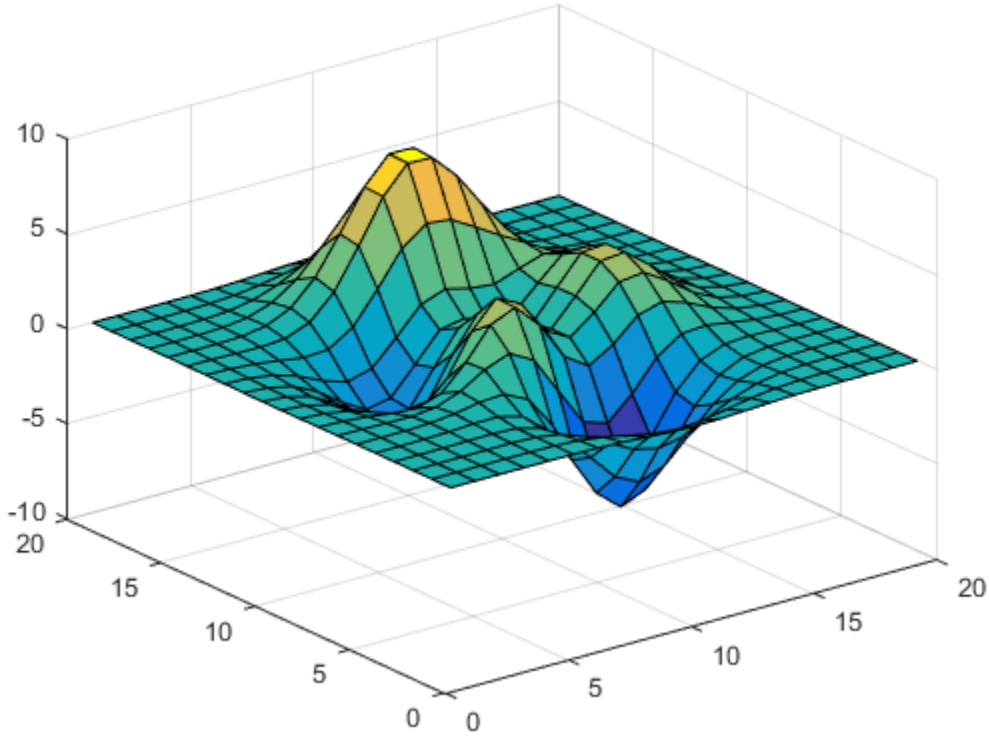
`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector `[x0,y0,z0]`.

Examples

Rotate Plot Around *x*-Axis

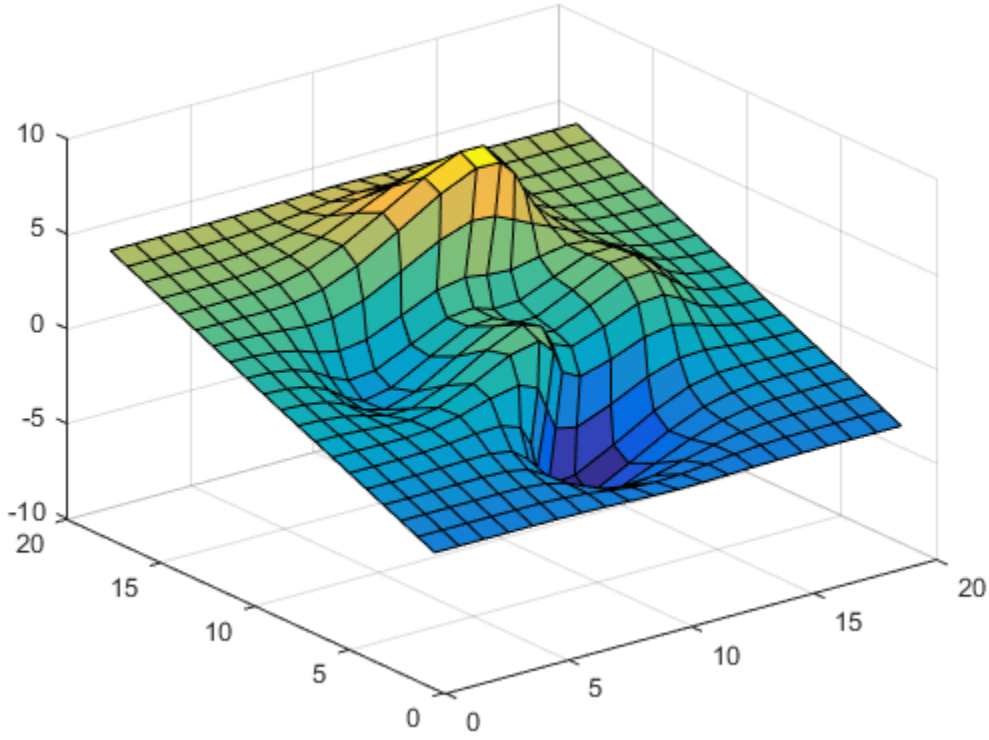
Create a surface plot of the `peaks` function and return the surface handle.

```
hSurface = surf(peaks(20));
```



Rotate the surface plot 25 degrees around its *x*-axis.

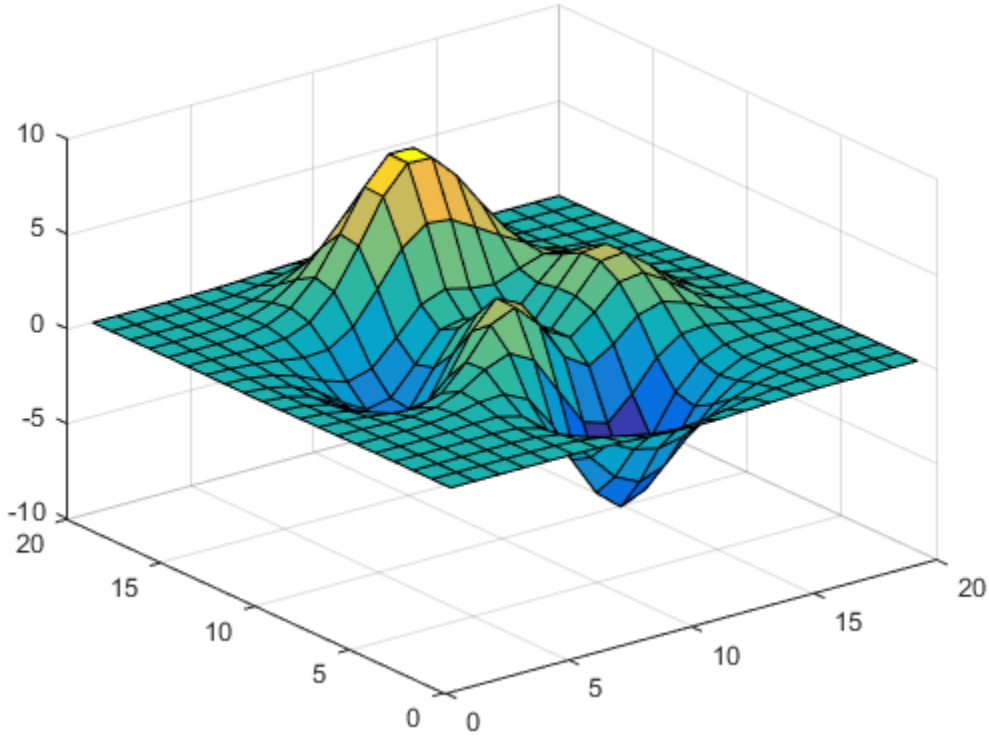
```
direction = [1 0 0];
rotate(hSurface,direction,25)
```



Rotate Plot Around *y*-Axis

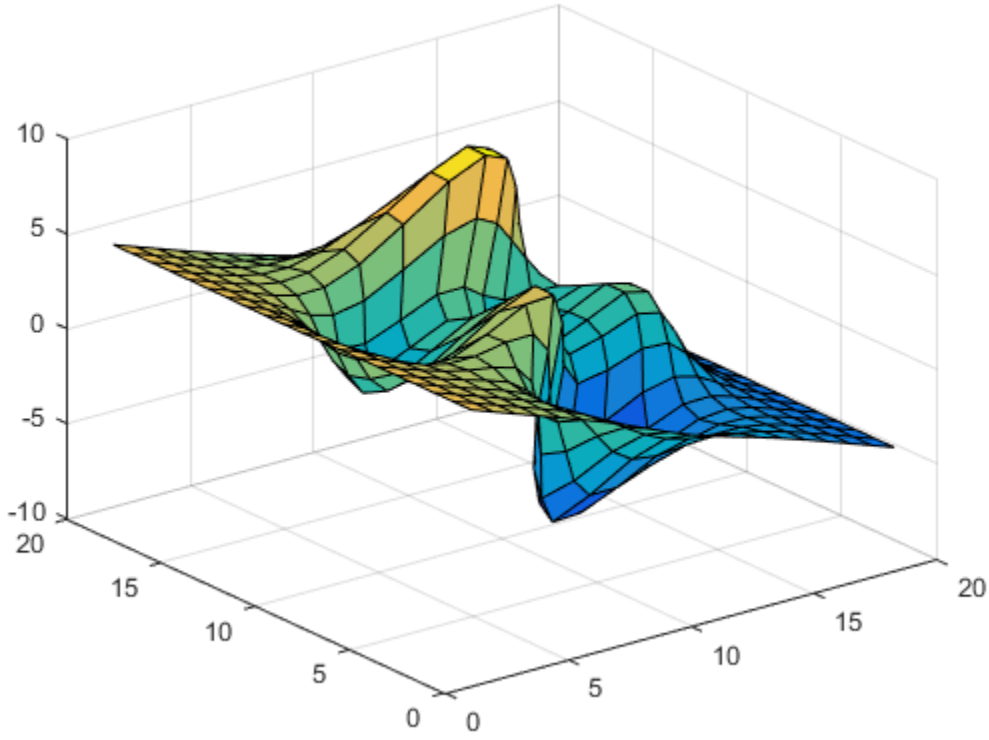
Create a surface plot of the `peaks` function and return the surface handle.

```
hSurface = surf(peaks(20));
```



Rotate the surface plot 25 degrees around its *y*-axis.

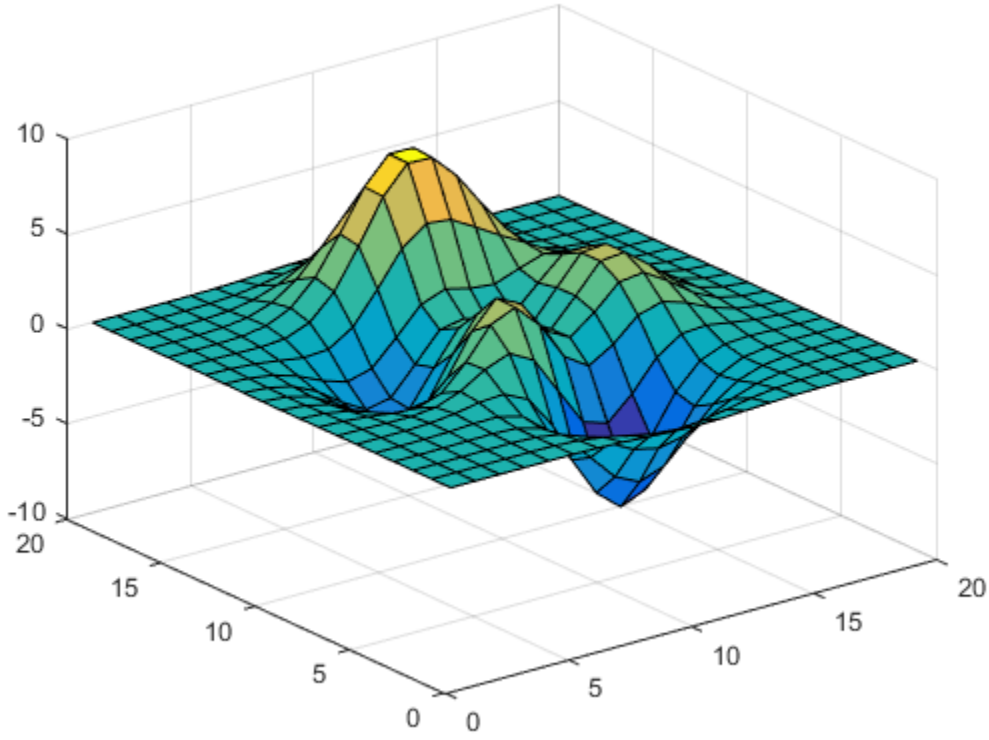
```
direction = [0 1 0];
rotate(hSurface,direction,25)
```



Rotate Plot Around *x*-Axis and *y*-Axis

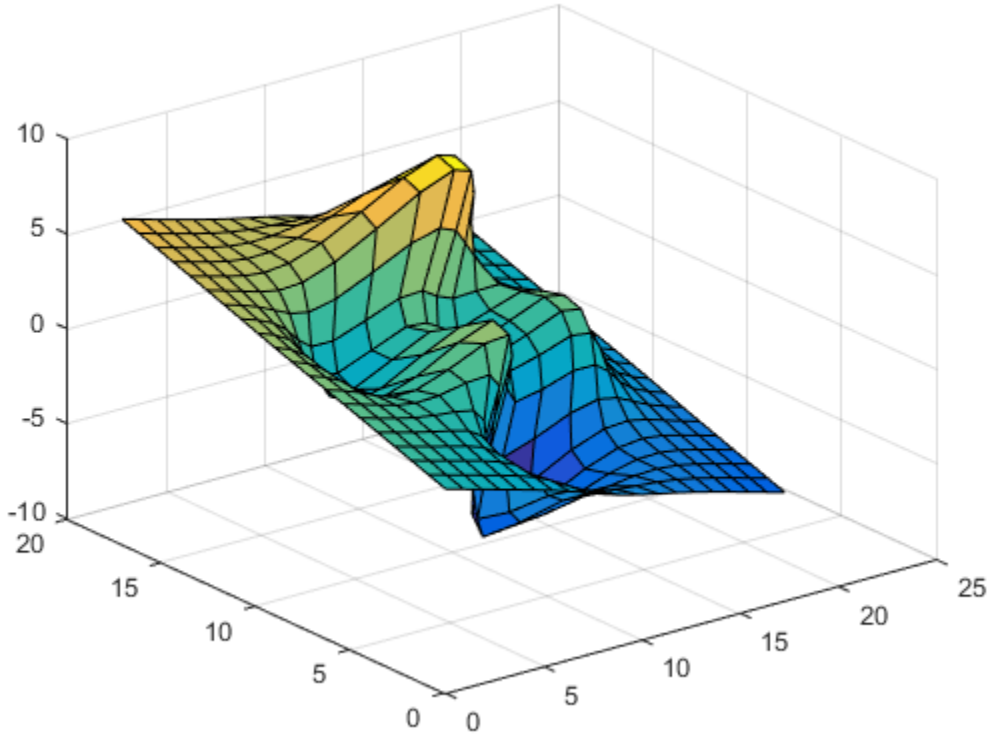
Create a surface plot of the `peaks` function and return the surface handle.

```
hSurface = surf(peaks(20));
```



Rotate the surface plot 25 degrees around its *x*-axis and *y*-axis.

```
direction = [1 1 0];
rotate(hSurface,direction,25)
```

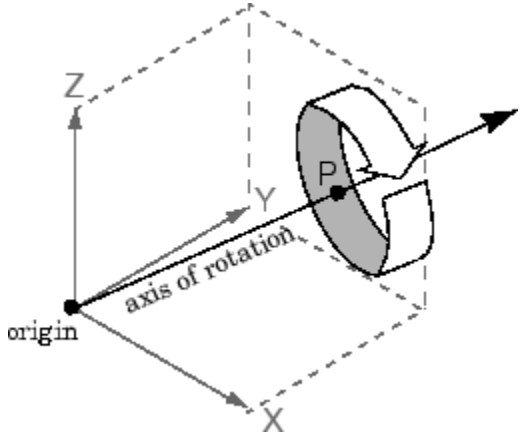


More About

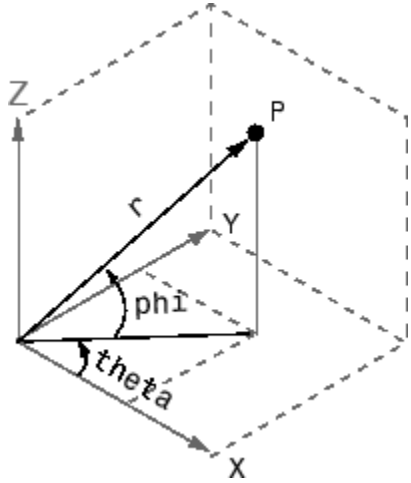
Tips

The rotation transformation modifies the object's data. This technique is different from that used by `view` and `rotate3d`, which modify only the viewpoint.

The axis of rotation is defined by an origin of rotation and a point `P`. Specify `P` as the spherical coordinates `[theta phi]` or as the Cartesian coordinates `[`x_p`,`y_p`,`z_p`]`.



In the two-element form for `direction`, `theta` is the angle in the *x*-*y* plane counterclockwise from the positive *x*-axis. `phi` is the elevation of the direction vector from the *x*-*y* plane.



The three-element form for `direction` specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin of rotation to `P`.

Tips

`rotate` changes the values of the `Xdata`, `Ydata`, and `Zdata` properties to rotate graphics objects.

See Also

`rotate3d` | `sph2cart` | `view`

Introduced before R2006a

rotate3d

Rotate 3-D view using mouse

Syntax

```
rotate3d on
rotate3d off
rotate3d
rotate3d(figure_handle,...)
rotate3d(axes_handle,...)
h = rotate3d(figure_handle)
```

Description

`rotate3d on` enables mouse-base rotation on all axes within the current figure.

`rotate3d off` disables interactive axes rotation in the current figure.

`rotate3d` toggles interactive axes rotation in the current figure.

`rotate3d(figure_handle,...)` enables rotation within the specified figure instead of the current figure.

`rotate3d(axes_handle,...)` enables rotation only in the specified axes.

`h = rotate3d(figure_handle)` returns a rotate3d *mode object* for figure figure_handle for you to customize the mode's behavior.

Using Rotate Mode Objects

You access the following properties of rotate mode objects.

- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set
- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure

- *RotateStyle* `'orbit'|'box'` — Sets the method of rotation

  `'orbit'` rotates the entire axes; `'box'` rotates a plot-box outline of the axes.

Rotate3D Mode Callbacks

You can program the following callbacks for rotate3d mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  The application can inhibit the rotate operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object that has been clicked on
  % event_obj    handle to event data object (empty in this release)
  % res [output] logical flag to determine whether the rotate
  %              operation should take place or the 'ButtonDownFcn'
  %              property of the object should take precedence
  ```

- `ActionPreCallback <function_handle>` — Function to execute before rotating

  Set this callback to listen to when a rotate operation will start. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data
  ```

  The event data has the following field:

`Axes`	The handle of the axes that is being panned

- `ActionPostCallback <function_handle>` — Function to execute after rotating

  Set this callback to listen to when a rotate operation has finished. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  ```

```
% event_obj   object containing struct of event data (same as the
%             event data of the 'ActionPreCallback' callback)
```

Rotate3D Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesRotate(h,axes)` — Function querying permission to rotate axes

  Calling the function `isAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a rotate operation is permitted on the axes objects.
- `setAllowAxesRotate(h,axes,flag)` — Function to set permission to pan axes

  Calling the function `setAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a rotate operation on the axes objects.

Examples

Example 1

Rotate the plot using the mouse:

```
surf(peaks);
rotate3d on;
```

Example 2

Rotate the plot using the "Plot Box" rotate style:

```
surf(peaks);
h = rotate3d;
h.RotateStyle = 'box';
h.Enable = 'on';
```

Example 3

Create two axes as subplots and then prevent one from rotating:

```
ax1 = subplot(1,2,1);
surf(peaks);
h = rotate3d;
h.Enable = 'on';
ax2 = subplot(1,2,2);
surf(membrane);
setAllowAxesRotate(h,ax2,false); % disable rotating for second plot
```

Example 4

Create a buttonDown callback for rotate mode objects to trigger. Copy the following code to a new file, execute it, and observe rotation behavior:

```
function demo_mbd
% Allow a line to have its own 'ButtonDownFcn' callback
hLine = plot(rand(1,10),'ButtonDownFcn','disp(''This executes'')');
hLine.Tag = 'DoNotIgnore';
h = rotate3d;
h.ButtonDownFilter = @mycallback;
h.Enable = 'on';
% mouse-click on the line
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true
objTag = obj.Tag;
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5

Create callbacks for pre- and post-buttonDown events for rotate3D mode objects to trigger. Copy the following code to a new file, execute it, and observe rotation behavior:

```
function demo_mbd2
% Listen to rotate events
surf(peaks);
h = rotate3d;
```

```
h.ActionPreCallback = @myprecallback;
h.ActionPostCallback = @mypostcallback;
h.Enable = 'on';

function myprecallback(obj,evd)
disp('A rotation is about to occur.');

function mypostcallback(obj,evd)
newView = round(evd.Axes.View);
msgbox(sprintf('The new view is [%d %d].',newView));
```

Alternatives

Use the Rotate3D tool  on the figure toolbar to enable and disable rotate3D mode on a plot, or select **Rotate 3D** from the figure's **Tools** menu. For details, see “Rotate in 3-D”.

More About

Tips

When enabled, `rotate3d` provides continuous rotation of axes and the objects it contains through mouse movement. A numeric readout appears in the lower left corner of the figure during rotation, showing the current azimuth and elevation of the axes. Releasing the mouse button removes the animated box and the readout. This differs from the `camorbit` function in that while the `rotate3d` tool modifies the `View` property of the axes, the `camorbit` function fixes the aspect ratio and modifies the `CameraTarget`, `CameraPosition` and `CameraUpVector` properties of the axes. See Axes Properties for more information.

You can also enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

You can create a `rotate3d` mode object once and use it to customize the behavior of different axes, as example 3 illustrates. You can also change its callback functions on the fly.

Note: Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change

any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a UI that updates a figure's callbacks, the UI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different 3-D rotation behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse will carry over to the linked axes, regardless of the behavior you previously set for the other axes.

See Also

`camorbit` | `pan` | `rotate` | `view` | `zoom`

Introduced before R2006a

round

Round to nearest decimal or integer

Syntax

```
Y = round(X)
Y = round(X,N)
Y = round(X,N,type)

Y = round(t)
Y = round(t,unit)
```

Description

`Y = round(X)` rounds each element of `X` to the nearest integer. In the case of a tie, where an element has a fractional part of exactly `0.5`, the `round` function rounds away from zero to the integer with larger magnitude.

`Y = round(X,N)` rounds to `N` digits:

- `N > 0`: round to `N` digits to the *right* of the decimal point.
- `N = 0`: round to the nearest integer.
- `N < 0`: round to `N` digits to the *left* of the decimal point.

`Y = round(X,N,type)` specifies the type of rounding. Specify `'significant'` to round to `N` significant digits (counted from the leftmost digit). In this case, `N` must be a positive integer.

`Y = round(t)` rounds each element of the `duration` array `t` to the nearest number of seconds.

`Y = round(t,unit)` rounds each element of `t` to the nearest number of the specified unit of time.

Examples

Round Matrix Elements

Round the elements of a 2-by-2 matrix to the nearest integer.

```
X = [2.11 3.5; -3.5 0.78];
Y = round(X)
```

```
Y =

     2     4
    -4     1
```

Round to Specified Number of Decimal Digits

Round `pi` to the nearest 3 decimal digits.

```
Y = round(pi,3)
```

```
Y =

    3.1420
```

Round to Nearest Multiple of 100

Round the number `863178137` to the nearest multiple of 100.

```
round(863178137,-2)
```

```
ans =

   863178100
```

Round Elements to Specified Number of Significant Digits

Round the elements of a vector to retain 2 significant digits.

```
format shortg
x = [1253 1.345 120.44]
y = round(x,2,'significant')
```

```
x =

         1253        1.345       120.44
```

```
y =

        1300         1.3         120
```

Controlling Number Display While Rounding

The `format` command controls how MATLAB displays numbers at the command line. If a number has extra digits that cannot be displayed in the current format, then MATLAB automatically rounds the number for display purposes. This can lead to unexpected results when combined with the `round` function.

Consider the result of the following subtraction operation, which displays 5 digits.

```
format short
x = 112.05 - 110
```

```
x =

    2.0500
```

Based on the displayed value of `x`, rounding `x` to 1 decimal should return `2.1`.

```
round(x,1)
```

```
ans =

     2
```

In fact, the problem here is that MATLAB is rounding `x` to 5 digits for display purposes. The `round` function returns the correct answer. Confirm the answer by viewing `x` with `format long`, which displays `x` rounded to 15 digits.

```
format long
x
```

```
x =

   2.049999999999997
```

Round Duration Values

Round each value in a duration array to the nearest number of seconds.

```
t = hours(8) + minutes(29:31) + seconds(1.3:0.5:2.3);
```

```
t.Format = 'hh:mm:ss.SS'

t =

   08:29:01.30   08:30:01.80   08:31:02.30

Y1 = round(t)

Y1 =

   08:29:01.00   08:30:02.00   08:31:02.00
```

Round each value in t to the nearest number of hours.

```
Y2 = round(t,'hours')

Y2 =

   08:00:00.00   09:00:00.00   09:00:00.00
```

Input Arguments

X — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex `X`, `round` treats the real and imaginary parts independently.

`X` must be `single` or `double` when you use `round` with more than one input.

`round` converts logical and `char` elements of `X` into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

N — Number of digits

scalar integer

Number of digits, specified as a scalar integer. When you specify `N`, the `round` function rounds `X` to the nearest multiple of 10^{-N}.

If you specify the `'significant'` rounding type, then `N` must be a positive integer.

`type` — Rounding type

`'decimals'` (default) | `'significant'`

Rounding type, specified as `'decimals'` or `'significant'`. The rounding type determines whether `round` considers digits in relation to the decimal point or the overall number of significant digits. `N` must be a positive integer when you specify `'significant'`. In that case, the `round` function rounds to the nearest number with `N` significant digits.

The default value is `'decimals'`, so that `round(X,N,'decimals')` is equivalent to `round(X,N)`.

Example: `round(3132,2,'significant')` returns `3100`, which is the closest number to `3132` that has `2` significant digits.

Data Types: `char`

`t` — Input duration

`duration` array

Input duration, specified as a `duration` array.

`unit` — Unit of time

`'seconds'` (default) | `'minutes'` | `'hours'` | `'days'`

Unit of time, specified as `'seconds'`, `'minutes'`, `'hours'`, or `'days'`.

More About

Tips

- `format short` and `format long` both display rounded numbers. This can cause unexpected results when combined with the `round` function.
- For display purposes, use `sprintf` to control the exact display of a number as a string. For example, to display exactly 2 decimal digits of `pi` (and no trailing zeros), use `sprintf(%.2f,pi)`.

- “Integers”
- “Floating-Point Numbers”

See Also

`ceil` | `fix` | `floor`

Introduced before R2006a

rowfun

Apply function to table rows

Syntax

```
B = rowfun(func,A)
B = rowfun(func,A,Name,Value)
```

Description

`B = rowfun(func,A)` applies the function func to each row of the table `A` and returns the results in the table `B`.

`func` accepts `size(A,2)` inputs.

`B = rowfun(func,A,Name,Value)` applies the function `func` to each row of the table `A` with additional options specified by one or more Name,Value pair arguments.

For example, you can specify which variables to pass to the function `func` and how to call `func`.

Examples

Apply Function with Single Output to Rows

Apply the function `hypot` to each row of the 5-by-2 table `A` to find the shortest distance between the variables `x` and `y`.

Create a table, `A`, with two variables of numeric data.

```
x = gallery('integerdata',10,[5,1],2);
y = gallery('integerdata',10,[5,1],8);

A = table(x,y)
```

```
A =

    x    y
```

```
    _    __

    9     1
    4     5
    3     2
    7     3
    1    10
```

Apply the function, `hypot` to each row of `A`. The function `hypot` takes two inputs and returns one output.

```
B = rowfun(@hypot,A,'OutputVariableNames','z')
```

```
B =

      z
    ______

    9.0554
    6.4031
    3.6056
    7.6158
     10.05
```

`B` is a table.

Append the function output, `B`, to the input table, `A`.

```
[A B]
```

```
ans =

    x    y       z
    _    __    ______

    9     1    9.0554
    4     5    6.4031
    3     2    3.6056
    7     3    7.6158
    1    10     10.05
```

Apply Function with Multiple Outputs to Rows

Define and apply a geometric Brownian motion model to a range of parameters.

Create a function in a file named `gbmSim.m` that contains the following code.

```
function [m,mtrue,s,strue] = gbmSim(mu,sigma)
% Discrete approximation to geometric Brownian motion
%
% [m,mtrue,s,strue] = gbmSim(mu,sigma) computes the
% simulated mean, true mean, simulated standard deviation,
% and true standard deviation based on the parameters mu and sigma.

numReplicates = 1000; numSteps = 100;
y0 = 1;
t1 = 1;
dt = t1 / numSteps;
y1 = y0*prod(1 + mu*dt + sigma*sqrt(dt)*randn(numSteps,numReplicates));
m = mean(y1); s = std(y1);
% Theoretical values
mtrue = y0 * exp(mu*t1); strue = mtrue * sqrt(exp(sigma^2*t1) - 1);
```

`gbmSim` accepts two inputs, `mu` and `sigma`, and returns four outputs, `m`, `mtrue`, `s`, and `strue`.

Define the table, `params`, containing the parameters to input to the Brownian Motion Model.

```
mu = [-.5; -.25; 0; .25; .5];
sigma = [.1; .2; .3; .2; .1];

params = table(mu,sigma)
```

```
params =

     mu      sigma
    -----    -----
     -0.5    0.1
    -0.25    0.2
       0     0.3
     0.25    0.2
      0.5    0.1
```

Apply the function, `gbmSim` to the rows of the table, `params`.

```
stats = rowfun(@gbmSim,params,...
    'OutputVariableNames',...
    {'simulatedMean' 'trueMean' 'simulatedStd' 'trueStd'})
```

```
stats =
```

```
    simulatedMean    trueMean    simulatedStd    trueStd
    _____________    ________    ____________    ________

    0.60501          0.60653     0.05808         0.060805
    0.77916           0.7788       0.161          0.15733
     1.0024                1      0.3048          0.30688
     1.2795            1.284     0.25851          0.25939
     1.6498           1.6487     0.16285          0.16529
```

The 4 strings specified by the `'OutputVariableNames'` name-value pair argument indicate that `rowfun` should obtain 4 outputs from `gbmSim`. You can specify fewer output variable names to return fewer outputs from `gbmSim`.

Append the function output, `stats`, to the input, `params`.

```
[params stats]
```

```
ans =

     mu      sigma    simulatedMean    trueMean    simulatedStd    trueStd
    _____    _____    _____________    ________    ____________    ________

     -0.5    0.1      0.60501          0.60653     0.05808         0.060805
    -0.25    0.2      0.77916           0.7788       0.161          0.15733
        0    0.3       1.0024                1      0.3048          0.30688
     0.25    0.2       1.2795            1.284     0.25851          0.25939
      0.5    0.1       1.6498           1.6487     0.16285          0.16529
```

Apply Function to Groups of Rows

Create a table, `A`, where `g` is a grouping variable.

```
g = gallery('integerdata',3,[15,1],1);
x = gallery('uniformdata',[15,1],9);
y = gallery('uniformdata',[15,1],2);

A = table(g,x,y)
```

```
A =

    g       x          y
    _    _______    ________

    3    0.24756     0.87516
    3     0.4358      0.3179
    3    0.97755     0.27323
```

```
    2    0.85995      0.6765
    3    0.30063    0.071171
    2    0.26589     0.19659
    3    0.13338     0.52908
    2     0.7425     0.17176
    1    0.85692     0.86996
    2    0.24286     0.24369
    3    0.19492     0.84291
    2    0.39076     0.55766
    1    0.29683     0.35681
    1    0.39031      0.2324
    2    0.18726      0.6476
```

Define the anonymous function, `func`, to compute the average difference between `x` and `y`.

```
func = @(x,y) mean(x-y);
```

Find the average difference between variables in groups `1`, `2`, and `3` defined by the grouping variable, `g`.

```
B = rowfun(func,A,...
    'GroupingVariable','g',...
    'OutputVariableName','MeanDiff')
```

```
B =

    g    GroupCount    MeanDiff
    _    __________    ________

1   1    3             0.028298
2   2    6             0.032569
3   3    6             -0.10327
```

The variable `GroupCount` indicates the number of rows in `A` for each group.

Input Arguments

`func` — Function

function handle

Function, specified as a function handle. You can define the function in a file or as an anonymous function. If `func` corresponds to more than one function file (that is, if `func`

represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

`func` can accept no more than `size(A,2)` inputs. By default, `rowfun` returns the first output of `func`. To return more than one output from `func`, use the `'NumOutputs'` or `'OutputVariableNames'` name-value pair arguments.

Example: `func = @(x,y) x.^2+y.^2;` takes two inputs and finds the sum of the squares.

A — Input table

table

Input table, specified as a table.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputVariables',2` uses only the second variable in `A` as an input to `func`.

'InputVariables' — Variables of A to pass to func

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector | ...

Variables of `A` to pass to `func`, specified as the comma-separated pair consisting of `'InputVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, logical vector, or an anonymous function that returns a logical scalar. If you specify `'InputVariables'` as an anonymous function that returns a logical scalar, `rowfun` only passes the variables in `A` where the specified function returns `1` (`true`).

'GroupingVariables' — One or more variables in A that define groups of rows

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

One or more variables in `A` that define groups of rows, specified as the comma-separated pair consisting of `'GroupingVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

A grouping variable can be a numeric vector, logical vector, string (or character array), cell array of strings, or a categorical vector. Rows in A that have the same grouping variable values belong to the same group. rowfun applies func to each group of rows, rather than separately to each row of A. The output, B, contains one row for each group.

'SeparateInputs' — Indicator for calling func with separate inputs
true (default) | false | 1 | 0

Indicator for calling func with separate inputs, specified as the comma-separated pair consisting of 'SeparateInputs' and either true, false, 1, or 0.

true	func expects separate inputs. rowfun calls func with size(A,2) inputs, one argument for each data variable. This is the default behavior.
false	func expects one vector containing all inputs. rowfun creates the input vector to func by concatenating the values in each row of A.

'ExtractCellContents' — Indicator to pass values from cell variables to func
false (default) | true | 0 | 1

Indicator to pass values from cell variables to func, specified as the comma-separated pair consisting of 'ExtractCellContents' and either false, true, 0, or 1.

true	rowfun extracts the contents of a variable in A whose data type is cell and passes the values, rather than the cells, to func For grouped computation, the values within each group in a cell variable must allow vertical concatenation.
false	rowfun passes the cells of a variable in A whose data type is cell to func. This is the default behavior.

'OutputVariableNames' — Variable names for outputs of func
string | cell array of nonempty, distinct strings

Variable names for outputs of func, specified as the comma-separated pair consisting of 'OutputVariableNames' and a string or a cell array of nonempty, distinct strings. The number of strings must equal the number of outputs desired from func.

Furthermore, the strings must be valid MATLAB identifiers. If valid MATLAB identifiers are not available for use as variable names, MATLAB uses a cell array of *N* strings of the form `{'Var1' ... 'VarN'}` where *N* is the number of variables. You can determine valid MATLAB variable names using the function `isvarname`.

`'NumOutputs'` — Number of outputs from `func`

`0` | positive integer

Number of outputs from `func`, specified as the comma-separated pair consisting of `'NumOutputs'` and `0` or a positive integer. The integer must be less than or equal to the possible number of outputs from `func`.

Example: `'NumOutputs',2` causes `rowfun` to call `func` with two outputs.

`'OutputFormat'` — Format of `B`

`'table'` (default) | `'uniform'` | `'cell'`

Format of `B`, specified as the comma-separated pair consisting of `'OutputFormat'` and either the string `'table'`, `'uniform'`, or `'cell'`.

`'table'`	`rowfun` returns a table with one variable for each output of `func`. For grouped computation, `B`, also contains the grouping variables. `'table'` allows you to use a function that returns values of different sizes or data types. However, for ungrouped computation, all of the outputs from `func` must have one row each time it is called. For grouped computation, all of the outputs from `func` must have the same number of rows. This is the default output format.
`'uniform'`	`rowfun` concatenates the values returned by `func` into a vector. All of the outputs from `func` must be scalars with the same data type.
`'cell'`	`rowfun` returns `B` as a cell array. `'cell'` allows you to use a function that returns values of different sizes or data types.

`'ErrorHandler'` — Function to call if `func` fails

function handle

Function to call if `func` fails, specified as the comma-separated pair consisting of `'ErrorHandler'` and a function handle. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Row or group index at which the error occurred.

- The set of input arguments to function `func` at the time of the error.

For example,

```
function [A, B] = errorFunc(S, varargin)
warning(S.identifier, S.message);
A = NaN; B = NaN;
```

Output Arguments

`B` — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

More About

- “Anonymous Functions”

See Also

`arrayfun` | `cellfun` | `isvarname` | `structfun` | `varfun`

rref

Reduced row echelon form (Gauss-Jordan elimination)

Syntax

```
R = rref(A)
[R,jb] = rref(A)
[R,jb] = rref(A,tol)
```

Description

`R = rref(A)` produces the reduced row echelon form of `A` using Gauss Jordan elimination with partial pivoting. A default tolerance of (`max(size(A))*eps *norm(A,inf)`) tests for negligible column elements.

`[R,jb] = rref(A)` also returns a vector `jb` such that:

- `r = length(jb)` is this algorithm's idea of the rank of `A`.
- `x(jb)` are the pivot variables in a linear system `Ax = b`.
- `A(:,jb)` is a basis for the range of `A`.
- `R(1:r,jb)` is the `r`-by-`r` identity matrix.

`[R,jb] = rref(A,tol)` uses the given tolerance in the rank tests.

Roundoff errors may cause this algorithm to compute a different value for the rank than `rank`, `orth` and `null`. Additionally, use `mldivide` to solve linear systems when high precision is required.

Examples

Use `rref` on a rank-deficient magic square:

```
A = magic(4), R = rref(A)
```

```
A =
```

```
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

R =
     1     0     0     1
     0     1     0     3
     0     0     1    -3
     0     0     0     0
```

See Also

`inv` | `lu` | `rank`

Introduced before R2006a

rsf2csf

Convert real Schur form to complex Schur form

Syntax

```
[U,T] = rsf2csf(U,T)
```

Description

The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

`[U,T] = rsf2csf(U,T)` converts the real Schur form to the complex form.

Arguments `U` and `T` represent the unitary and Schur forms of a matrix `A`, respectively, that satisfy the relationships: `A = U*T*U'` and `U'*U = eye(size(A))`. See `schur` for details.

Examples

Given matrix `A`,

```
 1     1     1     3
 1     2     1     1
 1     1     3     1
-2     1     1     4
```

with the eigenvalues

```
4.8121    1.9202 + 1.4742i    1.9202 + 1.4742i    1.3474
```

Generating the Schur form of `A` and converting to the complex Schur form

```
[u,t] = schur(A);
[U,T] = rsf2csf(u,t)
```

yields a triangular matrix T whose diagonal (underlined here for readability) consists of the eigenvalues of A.

```
U =

-0.4916       -0.2756 - 0.4411i    0.2133 + 0.5699i   -0.3428
-0.4980       -0.1012 + 0.2163i   -0.1046 + 0.2093i    0.8001
-0.6751        0.1842 + 0.3860i   -0.1867 - 0.3808i   -0.4260
-0.2337        0.2635 - 0.6481i    0.3134 - 0.5448i    0.2466

T =

4.8121        -0.9697 + 1.0778i   -0.5212 + 2.0051i   -1.0067
     0         1.9202 + 1.4742i    2.3355              0.1117 + 1.6547i
     0              0              1.9202 - 1.4742i    0.8002 + 0.2310i
     0              0                   0              1.3474
```

See Also

schur

Introduced before R2006a

run

Run MATLAB script

Syntax

```
run(scriptname)
```

Description

`run(scriptname)` runs the MATLAB script specified by `scriptname`.

Examples

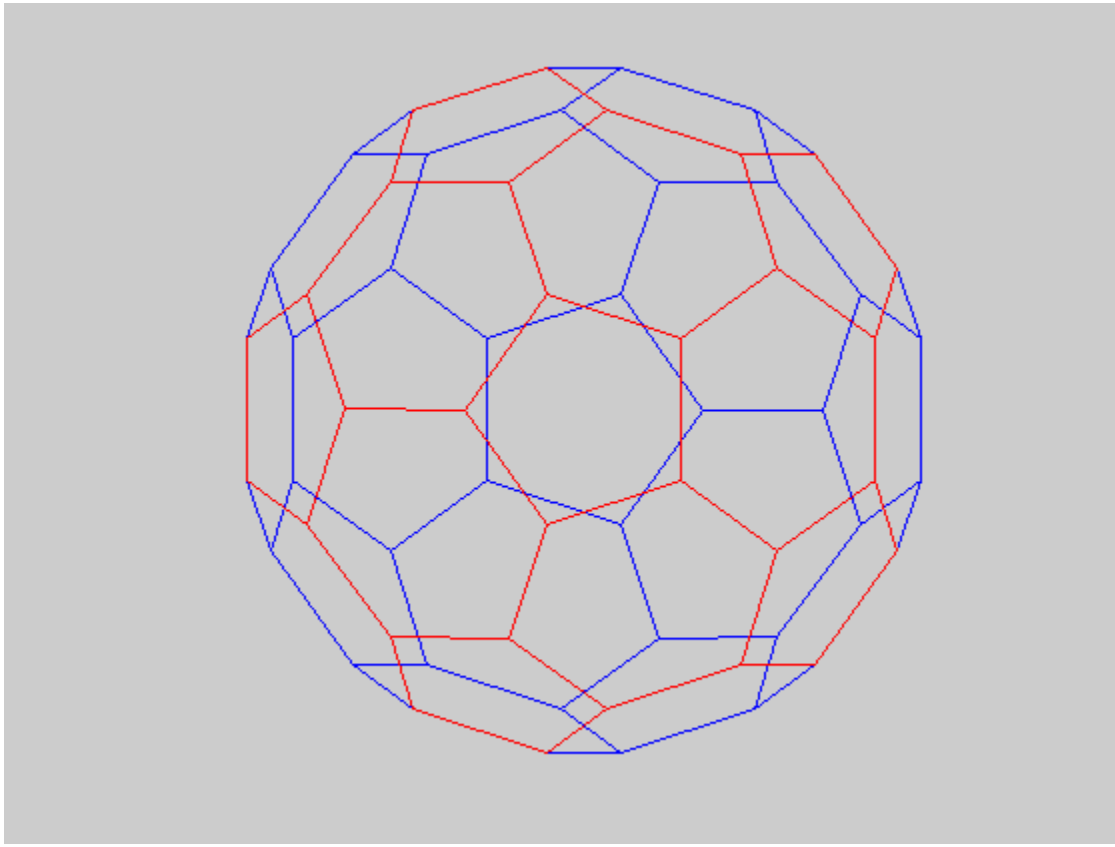
Run Script Not on Current Path

Create a temporary folder and copy an example MATLAB script to it.

```
tmp = tempname;
mkdir(tmp);
runtmp = fullfile(tmp,'buckyball.m');
demodir = fullfile(matlabroot,'toolbox','matlab',...
    'demos','buckydem.m');
copyfile(demodir,runtmp);
```

Run the new script.

```
run(runtmp)
```



Input Arguments

`scriptname` — Full or relative script path

string

Full or relative script path to a MATLAB script, specified as a string. scriptname can specify any file type that MATLAB can execute, such as MATLAB script files, Simulink models, or MEX-files.

More About

Tips

- `run` executes scripts not currently on the MATLAB path. However, you should use `cd` or `addpath` to navigate to or to add the appropriate folder, making a script executable by entering its name alone.
- `scriptname` can access any variables in the current workspace.
- `run` changes to the folder that contains the script, executes it, and resets back to the original folder. If the script itself changes folders, then `run` does not revert to the original folder, unless `scriptname` changes to the folder in which this script resides.
- If `scriptname` corresponds to both a `.m` file and a P-file residing in the same folder, then `run` executes the P-file. This occurs even if you specify `scriptname` with a `.m` extension.
- If a script is not on the MATLAB path, executing the `run` command caches the script. In the same session and after calling `run`, you can edit the script using an external editor. Call `clear scriptname` before calling `run` again to use the changed version of the script rather than the cached version. If you edit the script with the MATLAB editor, `run` executes the changed version and there is no need to call `clear scriptname`.
- “Files and Folders that MATLAB Accesses”

See Also

`addpath` | `cd` | `path` | `pwd`

Introduced before R2006a

runtests

Run set of tests

Syntax

```
results = runtests
results = runtests(tests)
results = runtests(tests,Name,Value)
```

Description

`results = runtests` runs all the tests in your current folder, and returns the results as a `TestResult` object.

`results = runtests(tests)` runs a specified set of tests.

`results = runtests(tests,Name,Value)` runs a set of tests with additional options specified by one or more Name,Value pair arguments.

Examples

Run Tests in Working Folder

Create a folder `myExample` in your current working folder, and change into that folder.

In the `myExample` folder, create a test script, `typeTest.m`.

```
%% Test double class
exp = 'double';
act = ones;
assert(isa(act,exp))

%% Test single class
exp = 'single';
act = ones('single');
assert(isa(act,exp))
```

```
%% Test uint16 class
exp = 'uint16';
act = ones('uint16');
assert(isa(act,exp))
```

In the `myExample` folder, create a test script, `sizeValueTest.m`.

```
%% Test size
exp = [7 13];
act = ones([7 13]);
assert(isequal(size(act),exp))

%% Test values
act = ones(42);
assert(unique(act) == 1)
```

Run all tests in the current folder.

```
runtests
```

```
Running sizeValueTest
..
Done sizeValueTest
__________

Running typeTest
...
Done typeTest
__________


ans =

  1x5 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration

Totals:
   5 Passed, 0 Failed, 0 Incomplete.
   0.013069 seconds testing time.
```

MATLAB ran 5 tests. There are 2 passing tests from `sizeValueTest` and 3 passing tests from `typeTest`.

Run Tests Using File Name

Create the test file shown below, and save it as `runtestsExampleTest.m` on your MATLAB path.

```
function tests = runtestsExampleTest
tests = functiontests(localfunctions);

function testFunctionOne(testCase)
```

Run the tests.

```
results = runtests('runtestsExampleTest.m');
```

```
Running runtestsExampleTest
.
Done runtestsExampleTest
__________
```

Run Tests in Subdirectory

If it doesn't already exist, create the test file, `runtestsExampleTest.m`, in the example above.

Create a subdirectory, `tmpTest`, and, in that directory, create the following `runtestsExampleSubFolderTest.m` file.

```
function tests = runtestsExampleSubFolderTest
tests = functiontests(localfunctions);

function testFunctionTwo(testCase)
```

Run the tests from the directory above `tmpTest` by setting `'Recursively'` to true.

```
results = runtests(pwd,'Recursively',true);
```

```
Running runtestsExampleTest
.
Done runtestsExampleTest
__________

Running runtestsExampleSubFolderTest
.
```

```
Done runtestsExampleSubFolderTest
__________
```

`runtests` ran the tests in both the current directory and the subdirectory.

If you do not specify the `'Recursively'` property for the `runtests` function, it does not run the test in the subdirectory.

```
results = runtests(pwd);
```

```
Running runtestsExampleTest
.
Done runtestsExampleTest
__________
```

Run Select Parameterized Tests

In your working folder, create `testZeros.m`. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
```

```
end
```

The full test suite has 11 test elements: 6 from the `testClass` method, 2 from the `testSize` method, and 1 each from the `testDefaultClass`, `testDefaultSize`, and `testDefaultValue` methods.

At the command prompt, run the test elements that use the `outSize` parameter property.

```
runtests('testZeros','ParameterProperty','outSize')
```

```
Running testZeros
........
Done testZeros
__________


ans =

  1x8 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration

Totals:
   8 Passed, 0 Failed, 0 Incomplete.
   0.013499 seconds testing time.
```

`runtests` executed eight tests that use the `outSize` parameter property: six from the `testClass` method and two from the `testSize` method.

At the command prompt, run the test elements that use the `singe` parameter name.

```
runtests('testZeros','ParameterName','single')
```

```
Running testZeros
..
Done testZeros
__________


c =
```

```
  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.0034442 seconds testing time.
```

`runtests` executed the two tests from the `testClass` method that use the `outSize` parameter name.

- "Write Function-Based Unit Tests"
- "Write Simple Test Case Using Functions"
- "Write Test Using Setup and Teardown Functions"

Input Arguments

`tests` — Array of tests

string | cell array of strings

Suite of tests specified as a string or cell array of strings. Each string in the cell array can contain the name of a test class, a test file, a package that containing your test classes, or a folder containing your test files.

Example: `'mypackage.MyTestClass'`

Example: `'ATestFile.m'`

Example: `pwd`

Example:
`{'mypackage.MyTestClass','ATestFile.m',pwd,'mypackage.subpackage'}`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

`'Recursively'` — Indicator to run tests in subfolders and subpackages
`false` (default) | `true` | `0` | `1`

Indicator to run tests in subfolders and subpackages, specified as `false` or `true` (`0` or `1`). By default `runtests` runs tests in the specified folder or package and not in their subfolders or subpackages.

Data Types: `logical`

`'Name'` — Name of suite element
string

String indicating the name of the suite element. For the testing framework to run a test, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterProperty'` — Name of parameterization property
string

String indicating the name of a property that defines a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterName'` — Name of parameter
string

String indicating the name of a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'BaseFolder'` — Name of base folder
string

String indicating the name of the folder that contains the file defining the test class, function, or script. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one

character. For test files defined in packages, the base folder is the parent of the top-level package folder.

`'Tag'` — Name of test element tag

string

Name of test element tag, specified as a string. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `functiontests`

Introduced in R2013b

Chart Surface Properties

Control chart surface appearance and behavior

Chart surface properties control the appearance and behavior of chart surface objects. By changing property values, you can modify certain aspects of the surface.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = surf(...);
c = h.CData;
h.CDataMapping = 'direct';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Faces

`FaceColor` — Face color

`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet or color string

Face color, specified as one of these values:

- `'flat'` — Use uniform face colors. Use the CData values. The color data at the first vertex determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`.
- `'interp'` — Interpolate the face colors. Bilinear interpolation of the CData values at each vertex determines the colors. You cannot use this value when the `FaceAlpha` property is set to `'flat'`.
- `'none'` — Do not draw the faces.
- `'texturemap'` — Transform the color data in CData so that it conforms to the surface.
- RGB triplet or color string — Use the same color for all of the faces.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`

Long Name	Short Name	RGB Triplet
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

FaceAlpha — Face transparency

1 (default) | scalar in range [0,1] | 'flat' | 'interp'

Face transparency, specified as one of these values:

- Scalar in range [0,1] — Use the same transparency value for all of the faces. A value of 1 is fully opaque and 0 is completely transparent.
- 'flat' — Use uniform transparency across each face. Use the AlphaData values to determine the transparency values. The alpha data at the first vertex determines the transparency for the entire face. You must first specify the AlphaData property as a matrix equal in size to ZData. You cannot use this value when the FaceColor property is set to 'interp'.
- 'interp' — Interpolate the transparency across each face. Bilinear interpolation of the AlphaData values at each vertex determines the transparency values. You must first specify the AlphaData property as a matrix equal in size to ZData. You cannot use this value when the FaceColor property is set to 'flat'.

FaceLighting — Effect of light objects on faces

'flat' (default) | 'gouraud' | 'none'

Effect of light objects on faces, specified as one of these values:

- 'flat' — Apply light uniformly across the faces. Use this value to view faceted objects.
- 'gouraud' — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- 'none' — Do not apply light from light objects to the faces.

Note: The `'phong'` value has been removed. Use `'gouraud'` instead.

`BackFaceLighting` — Face lighting when normals point away from camera

`'reverslit'` (default) | `'unlit'` | `'lit'`

Face lighting when the vertex normals point away from camera, specified as one of these values:

- `'reverslit'` — Light the face as if the vertex normal pointed towards the camera.
- `'unlit'` — Do not light the face.
- `'lit'` — Light the face according to the vertex normal.

Use this property to discriminate between the internal and external surfaces of an object. For an example, see "Back Face Lighting".

Edges

`MeshStyle` — Edges to display

`'both'` (default) | `'row'` | `'column'`

Edges to display, specified as `'both'`, `'row'`, or `'column'`.

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- - - - -
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	_._._._.-
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

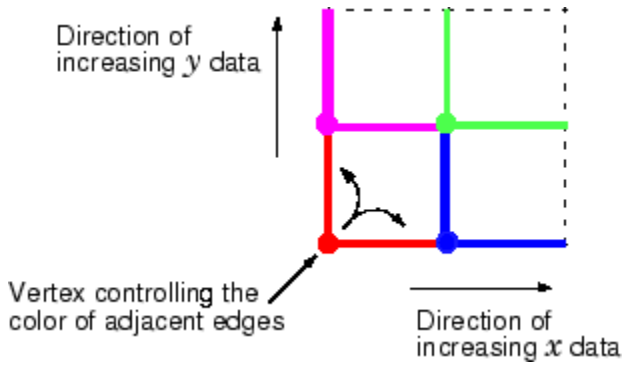
Example: `0.75`

`EdgeColor` — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet or color string

Surface edge line color, specified as one of these values:

- `'none'` — Do not draw edges.
- `'flat'` — Draw uniform edge colors. Use the `CData` value of the first vertex of the face to determine the color for each edge. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`.



- `'interp'` — Interpolate the edge colors. Use a linear interpolation of the `CData` values at the face vertices to determine the edge color. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`.
- RGB triplet or color string — Use the same color for all edges.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`

Long Name	Short Name	RGB Triplet
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

EdgeAlpha — Surface edge transparency

1 (default) | scalar value in range[0,1] | 'flat' | 'interp'

Surface edge transparency, specified as one of these values:

- Scalar in range[0,1] — Use the same transparency value for all of the edges. A value of 1 is fully opaque and 0 is completely transparent.
- 'flat' — Use uniform transparency across each edge. Use the values in AlphaData to determine the transparency for each edge. The value at the first vertex of the face determines the transparency for the edge. You must specify AlphaData as a matrix equal in size to ZData. You cannot use this value when the EdgeColor property is set to 'interp'.
- 'interp' — Interpolate the transparency across each edge. Use a bilinear interpolation of the AlphaData values for each vertex. You must specify AlphaData as a matrix equal in size to ZData. You cannot use this value when the EdgeColor property is set to 'flat'.

EdgeLighting — Effect of light objects on edges

'none' (default) | 'flat' | 'gouraud'

Effect of light objects on edges, specified as one of these values:

- 'flat' — Apply light uniformly across the each edges.
- 'none' — Do not apply lights from light objects to the edges.
- 'gouraud' — Calculate the light at the vertices, and then linearly interpolate across the edges.

Note: The 'phong' value has been removed. Use 'gouraud' instead.

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the chart surface object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond

String	Marker Symbol
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

Example: '+'

Example: 'diamond'

MarkerSize — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: 10

MarkerEdgeColor — Marker outline color

'auto' (default) | 'none' | 'flat' | RGB triplet or color string

Marker outline color, specified as specified as one of these values:

- 'auto' — Use the same color as the EdgeColor property.
- 'none' — Use no color, which makes unfilled markers invisible.
- 'flat' — Use the CData value at the vertex to set the color.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]

Long Name	Short Name	RGB Triplet
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | `'flat'` | RGB triplet or color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which allows the background to show through.
- `'auto'` — Use the same color as the `Color` property for the axes.
- `'flat'` — Use the CData value of the vertex to set the color.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]

Long Name	Short Name	RGB Triplet
'black	'k'	[0 0 0]

This property affects only the circle, square, diamond, pentagram, hexagram, and the four triangle marker types.

Example: [0.3 0.2 0.1]

Example: 'green'

Example:

Face and Vertex Normals

FaceNormals — Normal vectors for each surface face

[ ] (default) | (m-1)-by-(n-1)-by-3 array

Normal vectors for each surface face, specified as a (m-1)-by-(n-1)-by-3 array, where [m,n] = size(ZData). Specify one normal vector per face.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: single | double

FaceNormalsMode — Selection mode for FaceNormals

'auto' (default) | 'manual'

Selection mode for FaceNormals, specified as one of these values:

- 'auto' — Calculate the normal vectors based on the coordinate data.
- 'manual' — Use manually specified values. To specify the values, set the FaceNormals property.

VertexNormals — Normal vectors for each surface vertex

[ ] (default) | m-by-n-by-3 array

Normal vectors for each surface vertex, specified as a m-by-n-by-3 array, where [m,n] = size(ZData). Specify one normal vector per vertex.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: `single` | `double`

`VertexNormalsMode` — Selection mode for `VertexNormals`

`'auto'` (default) | `'manual'`

Selection mode for `VertexNormals`, specified as one of these values:

- `'auto'` — Calculate the normal vectors based on the coordinate data.
- `'manual'` — Use manually specified values. To specify the values, set the VertexNormals property.

`NormalMode` — (removed) Selection mode for `VertexNormals`

`'auto'` (default) | `'manual'`

The `NormalMode` property will be removed in a future release. Use VertexNormalsMode instead.

Color and Transparency Mapping

`CData` — Vertex colors

2-D or 3-D array

Vertex colors, specified in one of these forms:

- 2-D array — Use colormap colors. Specify a color for each vertex by setting CData to an array the same size as `ZData`. The CDataMapping property determines how these values map into the current colormap. If the FaceColor property is set to `'texturemap'`, then CData does not need to be the same size as ZData. However, it must be of type `double` or `uint8`. The CData values map to conform to the surface defined by ZData.
- 3-D array — Use true colors. Specify an RGB triplet color for each vertex by setting `CData` to an m-by-n-by-3 array where `[m,n] = size(ZData)`. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of a color. The first page of the array contains the red components, the second the green components, and the third the blue components of the colors. Since the surface uses true colors instead of colormap colors, the `CDataMapping` property has no effect.
  - If `CData` is of type `double` or `single`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.

- If `CData` is an integer type, then the surface uses the full range of data to determine the color. For example, if `CData` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CDataMode` — Selection mode for `CData`

`'auto'` (default) | `'manual'`

Selection mode for `CData`, specified as one of these values:

- `'auto'` — Use the `ZData` values to set the colors.
- `'manual'` — Use manually specified values. To specify the values, set the `CData` property.

`CDataSource` — Variable linked to `CData`

`''` | string containing MATLAB workspace variable name

Variable linked to `CData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `CData`.

By default, there is no linked variable, so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `CData` values immediately. To force an update of the values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, then you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`CDataMapping` — Color mapping method

`'scaled'` (default) | `'direct'`

Color mapping method, specified as `'scaled'` or `'direct'`. Use this property to control the mapping of color data values in CData into the colormap.

The methods have these effects:

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
  - If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
  - If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.
- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

`AlphaData` — Transparency data

1 (default) | scalar | m-by-n array

Transparency data for each face or each vertex, specified in one of these forms:

- Scalar — Use the same transparency value.
- m-by-n array — Use different transparency values for each face or vertex. Specify an m-by-n array of numeric values where `[m,n] = size(ZData)`.

The AlphaDataMapping property determines the mapping of transparency data values into the alphamap. The `Alphamap` property for the figure contains the alphamap.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Transparency data mapping method

`'scaled'` (default) | `'direct'` | `'none'`

Transparency data mapping method, specified as `'scaled'`, `'none'`, or `'direct'`. Use this property to control the mapping of transparency values contained in AlphaData into the figure alphamap.

The methods have these effects:

- `'scaled'` — Transform the values to range between the alpha limits and then linearly map the data values to the alpha values. The `ALim` property for the axes contains the alpha limits.

- `'none'` — Clamp the values between `0` and `1`. A value of 1 or greater is completely opaque and 0 or less is transparent.
- `'direct'` — Interpret the values as indices into the figure alphamap. Fix values with a decimal portion to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of 1 or less map to the first value in the alphamap. Values equal to or greater than the length of the alphamap map to the last value in the alphamap.
  - If the values are of type `uint8`, `uint16`. `uint32`, `uint64`. `int8`, `int16`, `int32`, or `int64`, then values of 0 or less map to the first value in the alphamap. Values equal to or greater than the length of the alphamap map to the last value in the alphamap (or up to the range limits of the type).
  - If the values are of type `logical`, then 0 maps the first value in the alphamap and 1 maps to the second value in the alphamap.

Ambient Lighting

`AmbientStrength` — Strength of ambient light

`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light

`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections

1 (default) | scalar in range `[0,1]`

Color of specular reflections, specified as a scalar value in the range `[0,1]`. A value of `1` sets the color using only the color of the light source. A value of `0` sets the color using both the color of the object from which it reflects and the color of the light source. The `Color` property of the light contains the color of the light source. The proportions vary linearly for values in between.

Example: `0.5`

Data Types: `double`

`SpecularExponent` — Size of specular spot

10 (default) | scalar greater than or equal to 1

Size of specular spot, specified as a scalar value greater than or equal to 1. Most materials have exponents in the range `[5 20]`.

Example: `7`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection

`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

Data

`XData` — x-coordinate data

matrix | vector

x-coordinate data specified as a matrix that is the same size as ZData or as a vector of length `n`, where `[m,n] = size(ZData)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

YData — y-coordinate data

matrix | vector

y-coordinate data, specified as a matrix that is the same size as ZData or a vector of length `m`, where `[m,n] = size(ZData)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

ZData — z-coordinate data

matrix

z-coordinate data, specified as a matrix of numeric values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

XDataSource — Variable linked to XData

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

YDataSource — Variable linked to YData

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZDataSource` — Variable linked to `ZData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `ZData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

`XDataMode` — Selection mode for `XData`

`'auto' | 'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the column indices of `ZData`..
- `'manual'` — Use manually specified values. To specify the values, use an input argument to the plotting function or directly set the `XData` property.

`YDataMode` — Selection mode for `YData`

`'auto' | 'manual'`

Selection mode for `YData`, specified as one of these values:

- `'auto'` — Use the row indices of `ZData`.
- `'manual'` — Use manually specified values. To specify the values, use an input argument to the plotting function or directly set the `YData` property.

Visibility

`Visible` — Visibility of chart surface

`'on'` (default) | `'off'`

Visibility of chart surface, specified as one of these values:

- `'on'` — Display the chart surface.
- `'off'` — Hide the chart surface without deleting it. You still can access the properties of an invisible chart surface object.

`Clipping` — Clipping of chart surface to axes limits

`'on'` (default) | `'off'`

Clipping of chart surface to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the chart surface that are outside the axes limits.
- `'off'` — Display the entire chart surface, even if parts of it appear outside the axes limits. Parts of the chart surface might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the chart surface that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.

- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'surface'`

Type of graphics object, returned as the string `'surface'`.

`Tag` — User-specified tag

`''` (default) | any string

Tag to associate with the chart surface, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with chart surface

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the chart surface object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the chart surface.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the chart surface object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the chart surface from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:

- `'on'` — Include the chart surface object in the legend as one entry (default).
- `'off'` — Do not include the chart surface object in the legend.
- `'children'` — Include only children of the chart surface object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of chart surface

axes object | group object | transform object

Parent of chart surface, specified as an axes, group, or transform object.

`Children` — Children of chart surface

empty `GraphicsPlaceholder` array

The chart surface has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of chart surface object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The chart surface object handle is always visible.
- `'off'` — The chart surface object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The chart surface object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the chart surface at the command-line, but allows callback functions to access it.

If the chart surface object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle

properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the chart surface. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The chart surface object — You can access properties of the chart surface object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the chart surface. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the chart surface when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the chart surface.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the chart surface that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the chart surface responds to the click or if an ancestor does.

- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'` and you can click a part of the chart surface that has no color. The HitTest property determines if the chart surface responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the chart surface passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the chart surface. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the chart surface that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the chart surface object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the chart surface is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the chart surface tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.

- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the chart surface. Setting the `CreateFcn` property on an existing chart surface has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during chart surface creation. MATLAB executes the callback after creating the chart surface and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The chart surface object — You can access properties of the chart surface object from within the callback function. You also can access the chart surface object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the chart surface. MATLAB executes the callback before destroying the chart surface so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The chart surface object — You can access properties of the chart surface object from within the callback function. You also can access the chart surface object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of chart surface

`'off'` (default) | `'on'`

Deletion status of chart surface, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the chart surface begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the chart surface no longer exists.

Check the value of the BeingDeleted property to verify that the chart surface is not about to be deleted before querying or modifying it.

See Also

`ezmesh` | `ezmeshc` | `ezsurf` | `ezsurfc` | `mesh` | `meshc` | `meshz` | `surf` | `surfc`

More About

- "Access Property Values"

- “Graphics Object Properties”

Primitive Surface Properties

Control primitive surface appearance and behavior

Primitive surface properties control the appearance and behavior of primitive surface objects. By changing property values, you can modify certain aspects of the primitive surface.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = surface;
c = h.CData;
h.CDataMapping = 'direct';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Faces

`FaceColor` — Face color

`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet or color string

Face color, specified as one of these values:

- `'flat'` — Use uniform face colors. Use the CData values. The color data at the first vertex determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`.
- `'interp'` — Interpolate the face colors. Bilinear interpolation of the CData values at each vertex determines the colors. You cannot use this value when the `FaceAlpha` property is set to `'flat'`.
- `'none'` — Do not draw the faces.
- `'texturemap'` — Transform the color data in CData so that it conforms to the surface.
- RGB triplet or color string — Use the same color for all of the faces.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

FaceAlpha — Face transparency

1 (default) | scalar in range [0,1] | 'flat' | 'interp'

Face transparency, specified as one of these values:

- Scalar in range [0,1] — Use the same transparency value for all of the faces. A value of 1 is fully opaque and 0 is completely transparent.
- 'flat' — Use uniform transparency across each face. Use the AlphaData values to determine the transparency values. The alpha data at the first vertex determines the transparency for the entire face. You must first specify the AlphaData property as a matrix equal in size to ZData. You cannot use this value when the FaceColor property is set to 'interp'.
- 'interp' — Interpolate the transparency across each face. Bilinear interpolation of the AlphaData values at each vertex determines the transparency values. You must first specify the AlphaData property as a matrix equal in size to ZData. You cannot use this value when the FaceColor property is set to 'flat'.

FaceLighting — Effect of light objects on faces

'flat' (default) | 'gouraud' | 'none'

Effect of light objects on faces, specified as one of these values:

- 'flat' — Apply light uniformly across the faces. Use this value to view faceted objects.
- 'gouraud' — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.

- `'none'` — Do not apply light from light objects to the faces.

Note: The `'phong'` value has been removed. Use `'gouraud'` instead.

`BackFaceLighting` — Face lighting when normals point away from camera

`'reverslit'` (default) | `'unlit'` | `'lit'`

Face lighting when the vertex normals point away from camera, specified as one of these values:

- `'reverslit'` — Light the face as if the vertex normal pointed towards the camera.
- `'unlit'` — Do not light the face.
- `'lit'` — Light the face according to the vertex normal.

Use this property to discriminate between the internal and external surfaces of an object. For an example, see “Back Face Lighting”.

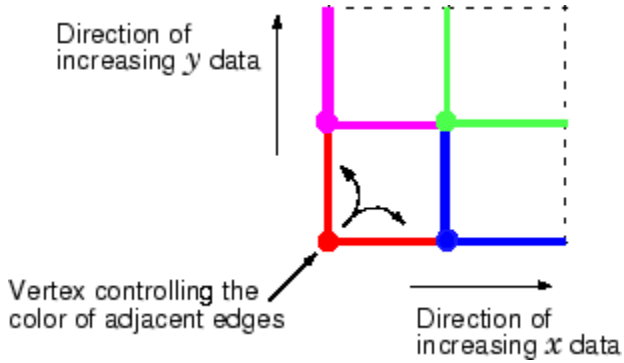
Edges

`EdgeColor` — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet or color string

Surface edge line color, specified as one of these values:

- `'none'` — Do not draw edges.
- `'flat'` — Draw uniform edge colors. Use the `CData` value of the first vertex of the face to determine the color for each edge. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`.



- `'interp'` — Interpolate the edge colors. Use a linear interpolation of the `CData` values at the face vertices to determine the edge color. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`.
- RGB triplet or color string — Use the same color for all edges.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

`EdgeAlpha` — Surface edge transparency

1 (default) | scalar value in range `[0,1]` | `'flat'` | `'interp'`

Surface edge transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use the same transparency value for all of the edges. A value of `1` is fully opaque and `0` is completely transparent.
- `'flat'` — Use uniform transparency across each edge. Use the values in `AlphaData` to determine the transparency for each edge. The value at the first vertex of the face determines the transparency for the edge. You must specify `AlphaData` as a matrix equal in size to `ZData`. You cannot use this value when the `EdgeColor` property is set to `'interp'`.
- `'interp'` — Interpolate the transparency across each edge. Use a bilinear interpolation of the `AlphaData` values for each vertex. You must specify `AlphaData` as a matrix equal in size to `ZData`. You cannot use this value when the `EdgeColor` property is set to `'flat'`.

`EdgeLighting` — Effect of light objects on edges

`'none'` (default) | `'flat'` | `'gouraud'`

Effect of light objects on edges, specified as one of these values:

- `'flat'` — Apply light uniformly across the each edges.
- `'none'` — Do not apply lights from light objects to the edges.
- `'gouraud'` — Calculate the light at the vertices, and then linearly interpolate across the edges.

Note: The `'phong'` value has been removed. Use `'gouraud'` instead.

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- - - - -
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	_._._._.-

String	Line Style	Resulting Line
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note: You must have a graphics card that supports this feature. To see if the feature is supported, type `opengl info`. If it is supported, then the returned fields contain the line `SupportsAlignVertexCenters: 1`.

`MeshStyle` — Edges to display

`'both'` (default) | `'row'` | `'column'`

Edges to display, specified as `'both'`, `'row'`, or `'column'`.

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the primitive surface object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | `'flat'` | RGB triplet or color string

Marker outline color, specified as specified as one of these values:

- `'auto'` — Use the same color as the EdgeColor property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- `'flat'` — Use the CData value at the vertex to set the color.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | `'flat'` | RGB triplet or color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which allows the background to show through.
- `'auto'` — Use the same color as the `Color` property for the axes.
- `'flat'` — Use the CData value of the vertex to set the color.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

`[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

This property affects only the circle, square, diamond, pentagram, hexagram, and the four triangle marker types.

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example:

Face and Vertex Normals

`FaceNormals` — Normal vectors for each surface face

`[ ]` (default) | (m-1)-by-(n-1)-by-3 array

Normal vectors for each surface face, specified as a (m-1)-by-(n-1)-by-3 array, where `[m,n] = size(ZData)`. Specify one normal vector per face.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: `single` | `double`

`FaceNormalsMode` — Selection mode for `FaceNormals`

`'auto'` (default) | `'manual'`

Selection mode for `FaceNormals`, specified as one of these values:

- `'auto'` — Calculate the normal vectors based on the coordinate data.
- `'manual'` — Use manually specified values. To specify the values, set the FaceNormals property.

`VertexNormals` — Normal vectors for each surface vertex

`[ ]` (default) | m-by-n-by-3 array

Normal vectors for each surface vertex, specified as a m-by-n-by-3 array, where `[m,n] = size(ZData)`. Specify one normal vector per vertex.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: `single` | `double`

`VertexNormalsMode` — Selection mode for `VertexNormals`

`'auto'` (default) | `'manual'`

Selection mode for `VertexNormals`, specified as one of these values:

- `'auto'` — Calculate the normal vectors based on the coordinate data.
- `'manual'` — Use manually specified values. To specify the values, set the VertexNormals property.

Color and Transparency Mapping

`AlphaData` — Transparency data

1 (default) | scalar | m-by-n array

Transparency data for each face or each vertex, specified in one of these forms:

- Scalar — Use the same transparency value.
- m-by-n array — Use different transparency values for each face or vertex. Specify an m-by-n array of numeric values where `[m,n] = size(ZData)`.

The AlphaDataMapping property determines the mapping of transparency data values into the alphamap. The `Alphamap` property for the figure contains the alphamap.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Transparency data mapping method

`'scaled'` (default) | `'direct'` | `'none'`

Transparency data mapping method, specified as `'scaled'`, `'none'`, or `'direct'`. Use this property to control the mapping of transparency values contained in AlphaData into the figure alphamap.

The methods have these effects:

- `'scaled'` — Transform the values to range between the alpha limits and then linearly map the data values to the alpha values. The `ALim` property for the axes contains the alpha limits.
- `'none'` — Clamp the values between `0` and `1`. A value of 1 or greater is completely opaque and 0 or less is transparent.
- `'direct'` — Interpret the values as indices directly into the figure alphamap. When not scaled, the values are usually integer values ranging from 1 to `length(alphamap)`. MATLAB maps values less than 1 to the first value in the alphamap and values greater than `length(alphamap)` to the last value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at 0 (that is, MATLAB maps a value of 0 to the first alpha value in the alphamap).

`CData` — Vertex colors

2-D or 3-D array

Vertex colors, specified in one of these forms:

- 2-D array — Use colormap colors. Specify a color for each vertex by setting CData to an array the same size as `ZData`. The CDataMapping property determines how these values map into the current colormap. If the FaceColor property is set to `'texturemap'`, then CData does not need to be the same size as ZData. However, it must be of type `double` or `uint8`. The CData values map to conform to the surface defined by ZData.
- 3-D array — Use true colors. Specify an RGB triplet color for each vertex by setting `CData` to an m-by-n-by-3 array where `[m,n] = size(ZData)`. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of a color. The first page of the array contains the red components, the second the green components, and the third the blue components of the colors. Since the surface uses true colors instead of colormap colors, the `CDataMapping` property has no effect.

- If `CData` is of type `double` or `single`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.
- If `CData` is an integer type, then the surface uses the full range of data to determine the color. For example, if `CData` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CDataMapping` — Direct or scaled colormapping

`'scaled'` (default) | `'direct'`

Direct or scaled colormapping, specified as one of these values:

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`CDataMode` — Selection mode for `CData`

`'auto'` (default) | `'manual'`

Selection mode for `CData`, specified as one of these values:

- `'auto'` — Use the `ZData` values to set the colors.
- `'manual'` — Use manually specified values. To specify the values, set the `CData` property.

Lighting

`AmbientStrength` — Strength of ambient light

`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light

`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections

1 (default) | scalar in range `[0,1]`

Color of specular reflections, specified as a scalar value in the range `[0,1]`. A value of `1` sets the color using only the color of the light source. A value of `0` sets the color using both the color of the object from which it reflects and the color of the light source. The `Color` property of the light contains the color of the light source. The proportions vary linearly for values in between.

Example: `0.5`

Data Types: `double`

`SpecularExponent` — Size of specular spot

10 (default) | scalar greater than or equal to 1

Size of specular spot, specified as a scalar value greater than or equal to 1. Most materials have exponents in the range `[5 20]`.

Example: `7`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection

`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

Data

`XData` — x-coordinate data

[3x3 double] (default) | vector or matrix

x-coordinate data specified as a matrix that is the same size as `ZData` or a vector of `length(n)`, where `[m,n] = size(ZData)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`YData` — y-coordinate data

[3x3 double] (default) | vector or matrix

y-coordinate data specified as a matrix that is the same size as `ZData` or a vector of `length(m)`, where `[m,n] = size(ZData)`.

Example:

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ZData` — z-coordinate data

[3x3 double] (default) | 2–D array

z-coordinate data specified as a 2-D array of numeric values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`XDataMode` — Selection mode for `XData`

`'auto'` | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the column indices of `ZData`.

- `'manual'` — Use manually specified value. To specify the value, pass an input argument to the plotting function or directly set the `XData` property.

`YDataMode` — Selection mode for `YData`

`'auto' | 'manual'`

Selection mode for `YData`, specified as one of these values:

- `'auto'` — Use the row indices of `ZData`.
- `'manual'` — Use manually specified value. To specify the value, pass an input argument to the plotting function or directly set the `YData` property.

Visibility

`Visible` — Visibility of primitive surface

`'on'` (default) | `'off'`

Visibility of primitive surface, specified as one of these values:

- `'on'` — Display the primitive surface.
- `'off'` — Hide the primitive surface without deleting it. You still can access the properties of an invisible primitive surface object.

`Clipping` — Clipping of primitive surface to axes limits

`'on'` (default) | `'off'`

Clipping of primitive surface to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the primitive surface that are outside the axes limits.
- `'off'` — Display the entire primitive surface, even if parts of it appear outside the axes limits. Parts of the primitive surface might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the primitive surface that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Tag` — User-specified tag

`''` (default) | any string

Tag to associate with the primitive surface, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`Type` — Type of graphics object

`'surface'`

Type of graphics object, returned as the string `'surface'`

`UserData` — Data to associate with primitive surface

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the primitive surface object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the primitive surface.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the primitive surface object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the primitive surface from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the primitive surface object in the legend as one entry (default).
   - `'off'` — Do not include the primitive surface object in the legend.
   - `'children'` — Include only children of the primitive surface object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of primitive surface

axes object | group object | transform object

Parent of primitive surface, specified as an axes, group, or transform object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of primitive surface object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The primitive surface object handle is always visible.
- `'off'` — The primitive surface object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The primitive surface object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the primitive surface at the command-line, but allows callback functions to access it.

If the primitive surface object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

`Children` — Children of primitive surface

empty `GraphicsPlaceholder` array

The primitive surface has no children. You cannot set this property.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the primitive surface. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The primitive surface object — You can access properties of the primitive surface object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the primitive surface. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the primitive surface when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the primitive surface.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the primitive surface that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the primitive surface responds to the click or if an ancestor does.
- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'` and you can click a part of the primitive surface that has no color. The HitTest property determines if the primitive surface responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the primitive surface passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the primitive surface. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the primitive surface that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the primitive surface object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the primitive surface tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the primitive surface is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the primitive surface. Setting the `CreateFcn` property on an existing primitive surface has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during primitive surface creation. MATLAB executes the callback after creating the primitive surface and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The primitive surface object — You can access properties of the primitive surface object from within the callback function. You also can access the primitive surface object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the primitive surface. MATLAB executes the callback before destroying the primitive surface so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The primitive surface object — You can access properties of the primitive surface object from within the callback function. You also can access the primitive surface object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of primitive surface

`'off'` (default) | `'on'`

Deletion status of primitive surface, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the primitive surface begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the primitive surface no longer exists.

Check the value of the BeingDeleted property to verify that the primitive surface is not about to be deleted before querying or modifying it.

See Also

`pcolor` | `surface`

More About

- "Access Property Values"
- "Graphics Object Properties"

save

Save workspace variables to file

Syntax

```
save(filename)
save(filename,variables)
save(filename,variables,fmt)
save(filename,variables,version)
save(filename,variables,'-append')

save filename
```

Description

`save(filename)` saves all variables from the current workspace in a MATLAB formatted binary file (MAT-file) called `filename`. If `filename` exists, `save` overwrites the file.

`save(filename,variables)` saves only the variables or fields of a structure array specified by `variables`.

`save(filename,variables,fmt)` saves in the file format specified by `fmt`. The `variables` argument is optional. If you do not specify `variables`, the `save` function saves all variables in the workspace.

`save(filename,variables,version)` saves to the MAT-file version specified by `version`. The `variables` argument is optional.

`save(filename,variables,'-append')` adds new variables to an existing file, and does not overwrite it. The `variables` argument is optional.

To append to a Version 6 MAT-file, you must also include `'-v6'` as an input argument.

`save filename` is the command form of the syntax. Command form requires fewer special characters. You do not need to type parentheses or enclose input strings in single quotes. Separate inputs with spaces instead of commas.

For example, to save a file named `test.mat`, these statements are equivalent:

```
save test.mat      % command form
save('test.mat')   % function form
```

You can include any of the inputs described in previous syntaxes. For example, to save the variable named `X`:

```
save test.mat X       % command form
save('test.mat','X')  % function form
```

Do not use command form when any of the inputs, such as `filename`, are variables.

Note: If you save a figure to a MAT-file in MATLAB release R2014b or later, you cannot open the MAT-file in earlier versions of MATLAB. Use `savefig` to save figures that are compatible with earlier versions of MATLAB.

Saving graphics handle variables can cause the creation of very large files because graphics objects contain their defining data. To save graphics figures, use `savefig`.

Examples

Save All Workspace Variables to MAT-File

Save all variables from the workspace in a binary MAT-file, `test.mat`. If `filename` is a variable, use function syntax.

```
filename = 'test.mat';
save(filename)
```

Otherwise, you also can use command syntax.

```
save test.mat
```

Remove the variables from the workspace, and then retrieve the data with the `load` function.

```
clear
load('test.mat')
```

Save Specific Variables to MAT-File

Create and save two variables, `p` and `q`, to a file called `pqfile.mat`.

```
p = rand(1,10);
q = ones(10);
save('pqfile.mat','p','q')
```

MATLAB® saves the variables to the file, `pqfile.mat`, in the current folder.

You also can use command syntax to save the variables, `p` and `q`.

```
save pqfile.mat p q
```

Save Data to ASCII File

Create two variables, save them to an ASCII file, and then view the contents of the file.

```
p = rand(1,10);
q = ones(10);
save('pqfile.txt','p','q','-ascii')
type('pqfile.txt')
```

The `type` function displays the contents of the file.

Alternatively, use command syntax for the `save` operation.

```
save pqfile.txt p q -ascii
```

Save Structure Fields as Individual Variables

Create a structure, `s1`, that contains three fields, `a`, `b`, and `c`.

```
s1.a = 12.7;
s1.b = {'abc',[4 5; 6 7]};
s1.c = 'Hello!';
```

Save the fields of structure `s1` as individual variables in a file called `newstruct.mat`.

```
save('newstruct.mat','-struct','s1');
```

Check the contents of the file using the `whos` function.

```
disp('Contents of newstruct.mat:')
whos('-file','newstruct.mat')
```

```
Contents of newstruct.mat:
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  b         1x2               262  cell
  c         1x6                12  char
```

Save Variables to Version 7.3 MAT-File

Create two variables and save them to a version 7.3 MAT-file called `example.mat`.

```
A = rand(5);
B = magic(10);
save('example.mat','A','B','-v7.3')
```

You also can use command syntax for the `save` operation.

```
save example.mat A B -v7.3
```

Append Variable to MAT-File

Save two variables to a MAT-file. Then, append a third variable to the same file.

```
p = rand(1,10);
q = ones(10);
save('test.mat','p','q')
```

View the contents of the MAT-file.

```
whos('-file','test.mat')
```

```
  Name        Size             Bytes  Class     Attributes

  p           1x10                80  double
  q          10x10               800  double
```

Create a new variable, `a`, and append it to the MAT-file.

```
a = 50;
save('test.mat','a','-append')
```

View the contents of the MAT-file.

```
whos('-file','test.mat')
```

```
  Name        Size             Bytes  Class     Attributes

  a           1x1                  8  double
  p           1x10                80  double
  q          10x10               800  double
```

The variable, `a`, is appended to `test.mat`, without overwriting the previous variables, `p` and `q`.

Note: To append to a Version 6 MAT-file, specify both `'-v6'` and `'-append'`. For example, to save variable `a` to the file, `test.mat`, call:

```
save('test.mat','a','-v6','-append')
```

Input Arguments

`filename` — Name of file

`'matlab.mat'` (default) | string

Name of file, specified as a string. If you do not specify `filename`, the `save` function saves to a file named `matlab.mat`.

If `filename` has no extension (that is, no period followed by text), and the value of `format` is `-mat` (the default), then MATLAB appends `.mat`. If `filename` does not include a full path, MATLAB saves to the current folder. You must have permission to write to the file.

When using the command form of `save`, it is unnecessary to enclose input strings in single quotes. However, if `filename` contains a space, you must enclose the argument in single quotes. For example, `save 'filename withspace.mat'`.

Example: `'myFile.mat'`

Data Types: `char`

`variables` — Names of variables to save

string

Names of variables to save, specified as one or more strings. When using the command form of `save`, you do not need to enclose input strings in single quotes. `variables` can be in one of the following forms.

Form of `variables` Input	Variables to Save
`var1,...,varN`	Save the listed variables, specified as individual strings. Use the `'*'` wildcard to match patterns. For example, `save('filename.mat','A*')` saves all variables in the file that start with `A`.

Form of variables Input	Variables to Save
`'-regexp',expr1,...,exprN`	Save only the variables whose names match the regular expressions, specified as strings. For example, `save('filename.mat','-regexp','^Mon','^Tues')` saves only the variables in the file whose names begin with `Mon` or `Tues`.
`'-struct',structName`	Store the fields of the scalar structure specified by *`structName`* as individual variables in the file. For example, `save('filename.mat','-struct','S')` saves the scalar structure, `S`.
`'-struct',structName,field1,...,`	Store the specified fields of the specified scalar structure as individual variables in the file. For example, `save('filename.mat,'-struct','S','a','b')` saves the fields `S.a` and `S.b`.
`'-struct',structName,'-regexp',expr1,...,exprN`	Store only the fields whose names match the regular expressions, specified as strings.

Data Types: `char`

`fmt` — File format

`'-mat'` (default) | `'-ascii'` | `'-ascii','-tabs'` | `'-ascii','-double'` | `'-ascii','-double','-tabs'`

File format, specified as one of the following strings. When using the command form of `save`, you do not need to enclose input strings in single quotes, for example, `save myFile.txt -ascii -tabs`.

Value of `fmt`	File Format
`'-mat'`	Binary MAT-file format.
`'-ascii'`	Text format with 8 digits of precision.
`'-ascii','-tabs'`	Tab-delimited text format with 8 digits of precision.
`'-ascii','-double'`	Text format with 16 digits of precision.

Value of `fmt`	File Format
`'-ascii','-double','-tabs'`	Tab-delimited text format with 16 digits of precision.

For MAT-files, data saved on one machine and loaded on another machine retains as much accuracy and range as the different machine floating-point formats allow.

Use one of the text formats to save MATLAB numeric values to text files. In this case:

- Each variable must be a two-dimensional `double` array.
- The output includes only the real component of complex numbers.
- MATLAB writes data from each variable sequentially to the file. If you plan to use the `load` function to read the file, all variables must have the same number of columns. The `load` function creates a single variable from the file.

If you specify a text format and any variable is a two-dimensional character array, then MATLAB translates characters to their corresponding internal ASCII codes. For example, `'abc'` appears in a text file as:

```
  9.7000000e+001   9.8000000e+001   9.9000000e+001
```

`version` — MAT-file version

`'-v7.3' | '-v7' | '-v6' | '-v4'`

MAT-file version, specified as one of the following strings. When using the command form of `save`, you do not need to enclose input strings in single quotes.

Value of `version`	Loads in MATLAB Versions	Supported Features	Compression	Maximum Size of Each Variable
`'-v7.3'`	7.3 (R2006b) or later	Saving and loading parts of variables, and all Version 7 features	Yes	≥ 2 GB on 64-bit computers
`'-v7'`	7.0 (R14) or later	Unicode character encoding, which enables file sharing between systems that use different default character encoding schemes, and all Version 6 features.	Yes	2^31 bytes per variable

Value of `version`	Loads in MATLAB Versions	Supported Features	Compression	Maximum Size of Each Variable
`'-v6'`	5 (R8) or later	N-dimensional arrays, cell arrays, structure arrays, variable names longer than 19 characters, and all Version 4 features.	No	2^31 bytes per variable
`'-v4'`	All	Two-dimensional `double`, character, and sparse arrays	No	100,000,000 elements per array, and 2^31 bytes per variable

If any data items require features that the specified version does not support, MATLAB does not save those items and issues a warning. You cannot specify a version later than your current version of MATLAB software.

Note: Version 7.3 MAT-files use an HDF5 based format that requires some overhead storage to describe the contents of the file. For cell arrays, structure arrays, or other containers that can store heterogeneous data types, Version 7.3 MAT-files are sometimes larger than Version 7 MAT-files.

To view or set the default version for MAT-files, select a **MAT-file save format** option in the General Preferences.

More About

Tips

- For more flexibility in creating ASCII files, use `dlmwrite` or `fprintf`.

- "What Is the MATLAB Workspace?"
- "Save, Load, and Delete Workspace Variables"
- "Write to Delimited Data Files"
- "Regular Expressions"

See Also

`clear` | `hgsave` | `load` | `matfile` | `regexp` | `saveas` | `whos`

Introduced before R2006a

save (COM)

Serialize control object to file

Syntax

```
save(h,'filename')
```

Description

`save(h,'filename')` saves the COM control object, `h`, to the file specified in the string, `filename`.

Note The COM `save` function is only supported for controls.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2',[0 0 200 200],f);
save(h,'mwsample')
```

Now, alter the figure by changing its label and the radius of the circle.

```
h.Label = 'Circle';
h.Radius = 50;
Redraw(h);
```

Using the `load` function, you can restore the control to its original state.

```
load(h,'mwsample');
get(h)
```

```
ans =
     Label: 'Label'
    Radius: 20
```

More About

Tips

COM functions are available on Microsoft Windows systems only.

See Also

`load (COM)` | `delete (COM)` | `actxcontrol` | `actxserver` | `release`

Introduced before R2006a

save (serial)

Save serial port objects and variables to file

Syntax

```
save filename
save filename obj1 obj2...
```

Description

`save filename` saves all MATLAB variables to the file `filename`. If an extension is not specified for `filename`, then the `.mat` extension is used.

`save filename obj1 obj2...` saves the serial port objects `obj1` `obj2...` to the file `filename`.

Examples

This example illustrates how to use the command and functional form of `save` on a Windows platform.

```
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

More About

Tips

You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and serial port objects as strings. For example. to save the serial port object `s` to the file `MySerial.mat` on a Windows platform

```
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the serial port object is not automatically stored in the file. For example, suppose there is data in the input buffer for `obj`. To save that data to a file, you must bring it into the MATLAB workspace using one of the synchronous read functions, and then save to the file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

See Also

`load` | `record` | `Status`

Introduced before R2006a

saveas

Save figure to specific file format

Syntax

```
saveas(fig,filename)
saveas(fig,filename,formattype)
```

Description

`saveas(fig,filename)` saves the figure or Simulink block diagram specified by `fig` to file `filename`. Specify the file name as a string that includes a file extension, for example, `'myplot.jpg'`. The file extension defines the file format. If you do not specify an extension, then `saveas` saves the figure to a FIG-file.To save the current figure, specify `fig` as `gcf`.

`saveas(fig,filename,formattype)` creates the file using the specified file format, `formattype`. If you do not specify a file extension in the file name, for example, `'myplot'`, then the standard extension corresponding to the specified format automatically appends to the file name. If you specify a file extension, it does not have to match the format. `saveas` uses `formattype` for the format, but saves the file with the specified extension. Thus, the file extension might not match the actual format used.

Examples

Save Figure as PNG File

Create a bar chart and save it as a PNG file.

```
x = [2 4 7 2 4 5 2 5 1 4];
bar(x);
saveas(gcf,'Barchart.png')
```

Save Figure as EPS File

Create a bar chart and save it as an EPS file. Specify the `'epsc'` driver to save it in color.

```
x = [2 4 7 2 4 5 2 5 1 4];
bar(x);
saveas(gcf,'Barchart','epsc')
```

`saveas` saves the bar chart as `Barchart.eps`.

Save Simulink Block Diagram as BMP File

Save a Simulink block diagram named `'sldemo_tank'` as a BMP file. Use `get_param` to get the handle of the diagram. You must have Simulink installed to run this code.

```
sldemo_tank
fig = get_param('sldemo_tank','Handle');
saveas(fig,'MySimulinkDiagram.bmp');
```

- "Save Figure for Document or Presentation"

Input Arguments

`fig` — Figure to save

figure object | Simulink block diagram

Figure to save, specified as a figure object or a Simulink block diagram. If you specify other types of graphics objects, such as an axes, then `saveas` saves the parent figure to the object. This means that `saveas` cannot save a subplot without also saving all subplots in the parent figure.

Example: `saveas(gcf,'MyFigure.png')`

To save a Simulink block diagram, use `get_param` to get the handle of the diagram. For example, save a block diagram named `'sldemo_tank'`.

```
sldemo_tank
saveas(get_param('sldemo_tank','Handle'),'MySimulinkDiagram.bmp');
```

`filename` — File name

string

File name, specified as a string with or without a file extension.

Example: `'Bar Chart'`

Example: `'Bar Chart.png'`

If you specify a file extension, then `saveas` uses the associated format. If you specify a file extension and additionally specify the `formattype` input argument, then `saveas` uses `formattype` for the format and saves the file with the specified file name. Thus, the file extension might not match the actual format used.

You can specify any extension corresponding to a file format. This table lists some common file extensions.

Extension	Resulting Format
`.fig`	MATLAB FIG-file (invalid for Simulink block diagrams)
`.m`	MATLAB FIG-file and MATLAB code that opens figure (invalid for Simulink block diagrams)
`.jpg`	JPEG image
`.png`	Portable Network Graphics
`.eps`	EPS Level 3 Black and White
`.pdf`	Portable Document Format
`.bmp`	Windows bitmap
`.emf`	Enhanced metafile
`.pbm`	Portable bitmap
`.pcx`	Paintbrush 24-bit
`.pgm`	Portable Graymap
`.ppm`	Portable Pixmap
`.tif`	TIFF image, compressed

`formattype` — File format

`'fig'` | `'m'` | `'mfig'` | bitmap image file format | vector graphics file format

File format, specified as one of these options:

- `'fig'` — Save the figure as a MATLAB figure file with the `.fig` extension. To open figures saved with the `.fig` extension, use the `openfig` function. This format is not value for Simulink block diagrams.

- `'m'` or `'mfig'` — Save the figure as a MATLAB figure file and additionally create a MATLAB file that opens the figure. To open the figure, run the MATLAB file. This option is not valid for Simulink block diagrams.
- Bitmap image file format — Specify the format as one of the bitmap image options in the table, Bitmap Image Formats.
- Vector graphics file format — Specify the format as one of the vector graphics options in the table, Vector Graphics Formats.

Bitmap Image File

Bitmap images contain a pixel-based representation of the figure. The size of the generated file depends on the figure and the format used. Bitmap images are widely used by web browsers and other applications that display graphics. However, they do not scale well and you cannot modify individual graphics objects (such as lines and text) in other graphics applications.

Bitmap Image Formats

Option	Format	Default File Extension
`'jpeg'`	JPEG 24-bit	`.jpg`
`'png'`	PNG 24-bit	`.png`
`'tiff'`	TIFF 24-bit (compressed)	`.tif`
`'tiffn'`	TIFF 24-bit (not compressed)	`.tif`
`'meta'`	Enhanced metafile (Windows only)	`.emf`
`'bmpmono'`	BMP Monochrome	`.bmp`
`'bmp'`	BMP 24-bit	`.bmp`
`'bmp16m'`	BMP 24-bit	`.bmp`
`'bmp256'`	BMP 8-bit (256 color, uses a fixed colormap)	`.bmp`
`'hdf'`	HDF 24-bit	`.hdf`
`'pbm'`	PBM (plain format) 1-bit	`.pbm`
`'pbmraw'`	PBM (raw format) 1-bit	`.pbm`

Option	Format	Default File Extension
`'pcxmono'`	PCX 1-bit	`.pcx`
`'pcx24b'`	PCX 24-bit color (three 8-bit planes)	`.pcx`
`'pcx256'`	PCX 8-bit newer color (256 color)	`.pcx`
`'pcx16'`	PCX older color (EGA/VGA 16-color)	`.pcx`
`'pgm'`	PGM (plain format)	`.pgm`
`'pgmraw'`	PGM (raw format)	`.pgm`
`'ppm'`	PPM (plain format)	`.ppm`
`'ppmraw'`	PPM (raw format)	`.ppm`

Vector Graphics File

Vector graphics files store commands that redraw the figure. This type of format scales well, but can result in a large file. In some cases, a vector graphics format might not produce the correct 3-D arrangement of objects. Some applications support extensive editing of vector graphics formats, but others do not support editing beyond resizing the graphic. The best practice is to make all the necessary changes while your figure is still in MATLAB.

Typically, `saveas` uses the Painters renderer when generating vector graphics files. For some complex figures, `saveas` uses the OpenGL renderer instead. If it uses the OpenGL renderer, then the vector graphics file contains an embedded image, which might limit the extent to which you can edit the image in other applications. To ensure that `saveas` uses the Painters renderer, set the `Renderer` property for the figure to `'painters'`.

If you set the `Renderer` property for the figure, then `saveas` uses that renderer. Otherwise, it chooses the appropriate renderer. However, if `saveas` chooses a renderer that differs from the renderer used for the figure on the display, then some details of the saved figure can differ from the displayed figure. If necessary, you can make the displayed figure and the saved figure use the same renderer by setting the `Renderer` property for the figure.

Vector Graphics Formats

Option	Format	Default File Extension
`'pdf'`	Full page Portable Document Format (PDF) color	`.pdf`
`'eps'`	Encapsulated PostScript (EPS) Level 3 black and white	`.eps`
`'epsc'`	Encapsulated PostScript (EPS) Level 3 color	`.eps`
`'eps2'`	Encapsulated PostScript (EPS) Level 2 black and white	`.eps`
`'epsc2'`	Encapsulated PostScript (EPS) Level 2 color	`.eps`
`'meta'`	Enhanced Metafile (Windows only)	`.emf`
`'svg'`	SVG (scalable vector graphics)	`.svg`
`'ps'`	Full-page PostScript (PS) Level 3 black and white	`.ps`
`'psc'`	Full-page PostScript (PS) Level 3 color	`.ps`
`'ps2'`	Full-page PostScript (PS) Level 2 black and white	`.ps`
`'psc2'`	Full-page PostScript (PS) Level 2 color	`.ps`

Note: Only PDF and PS formats use the `PaperOrientation` property of the figure and the `left` and `bottom` elements of the `PaperPosition` property. Other formats ignore these values.

More About

Tips

- To control the size or resolution when you save a figure, use the `print` function instead.
- The `saveas` function and the **Save As** dialog box (accessed from the **File** menu) do not produce identical results. The **Save As** dialog box produces images at screen resolution and at screen size. The `saveas` function uses a resolution of 150 DPI and uses the `PaperPosition` and `PaperPositionMode` properties of the figure to determine the size of the image.
- Details of saved and printed figures can differ from the figure on the display. To get output that is more consistent with the display, see “Save Figure Preserving Background Color” and “Save Figure at Specific Size and Resolution”.

See Also

`open` | `print` | `savefig`

Introduced before R2006a

savefig

Save figure and contents to FIG-file

Syntax

```
savefig(filename)
savefig(H,filename)
savefig(H,filename,'compact')
```

Description

`savefig(filename)` saves the current figure to a FIG-file named *`filename`*`.fig`.

`savefig(H,filename)` saves the figures identified by the graphics array `H` to a FIG-file named *`filename`*`.fig`.

`savefig(H,filename,'compact')` saves the specified figures in a FIG-file that can be opened only in MATLAB R2014b or later releases. The `'compact'` option reduces the size of the `.fig` file and the time required to create the file.

Examples

Save Current Figure to FIG-File

Create a surface plot of the `peaks` function. Save the figure to the file `PeaksFile.fig`.

```
figure
surf(peaks)
savefig('PeaksFile.fig')
```

To open the saved figure, use the command:

```
openfig('PeaksFile.fig');
```

MATLAB creates a new figure using the saved `.fig` file.

Save Multiple Figures to FIG-File

Create two plots and store the figure handles in array `h`. Save the figures to the file `TwoFiguresFile.fig`. Close the figures after saving them.

```
h(1) = figure;
z = peaks;
surf(z)

h(2) = figure;
plot(z)

savefig(h,'TwoFiguresFile.fig')
close(h)
```

To open the two figures, use the command:

```
figs = openfig('TwoFiguresFile.fig');
```

`figs` contains the handles of the two figures created.

Save Figure Using `'compact'` Option

Save a figure using the compact option:

```
h = figure
surf(peaks)
savefig(h,'PeaksFile.fig','compact')
```

To open the figure, use the command:

```
openfig('PeaksFile.fig');
```

- "Save Figure for Document or Presentation"

Input Arguments

`H` — One or more figures

single figure | array of figures

One or more figures, specified as a single figure or an array of figures.

`filename` — File name

`'Untitled.fig'` (default) | string

File name, specified as a string. If you do not specify a file name, then MATLAB saves the file as `Untitled.fig`, which is the default.

If the specified string does not include a `.fig` file extension, then MATLAB appends the extension. `savefig` does not accept other file extensions.

Example: `'ExampleFile.fig'`

`'compact'` — File format for R2014b or later releases

`'compact'`

File format for R2014b or later releases of MATLAB, specified as the string `'compact'`. This option results in smaller `.fig` files. However, do not use the `'compact'` option if you want to open the `.fig` file in versions of MATLAB before R2014b.

More About

Tips

- You must use MATLAB to open files saved using `savefig`. To open the file, pass the file name to the function `openfig` or `open`. For example,

  ```
  openfig('ExampleFile.fig')
  ```
  opens the file, `ExampleFile.fig`, in MATLAB.
- `savefig` saves the full MATLAB figure. To save only part of a figure, such as an axes, or to save handles in addition to the data, use the `save` function to create a MAT-file.

See Also

`findobj` | `load` | `open` | `openfig` | `save`

Introduced in R2013b

saveobj

Modify save process for object

Syntax

```
b = saveobj(a)
```

Description

`b = saveobj(a)` is called by the `save` function if the class of `a` defines a `saveobj` method. `save` writes the returned value, `b`, to the MAT-file.

Define a `loadobj` method to take the appropriate action when loading the object.

If `A` is an array of objects, MATLAB invokes `saveobj` separately for each object saved.

Examples

Call the superclass `saveobj` method from the subclass implementation of `saveobj` with the following syntax:

```
classdef mySub < super
   methods
      function sobj = saveobj(obj)
         % Call superclass saveobj method
         sobj = saveobj@super(obj);
         % Perform subclass save operations
         ...
      end
   ...
   end
...
end
```

Update object when saved:

```
function b = saveobj(a)
```

```
   % If the object does not have an account number,
   % Add account number to AccountNumber property
   if isempty(a.AccountNumber)
      a.AccountNumber = getAccountNumber(a);
   end
   b = a;
end
```

See Also

`save` | `load` | `loadobj`

Introduced before R2006a

savepath

Save current search path

Syntax

```
savepath
savepath folderName/pathdef.m

status = savepath( ___ )
```

Description

`savepath` saves the current MATLAB search path to an existing `pathdef.m` file in the current folder. If there is no `pathdef.m` file in the current folder, then `savepath` saves the search path to the first `pathdef.m` file on the current path. If there is no such file on the current path, then `savepath` saves the search path to the `pathdef.m` file that MATLAB located at startup.

On a Windows system with User Account Control (UAC) enabled, you might be prompted to allow the update operation because it requires administrator-level permission.

`savepath folderName/pathdef.m` saves the current search path to `pathdef.m` located in the folder specified by `folderName`. If you do not specify `folderName`, then `savepath` saves `pathdef.m` in the current folder.

Use this syntax if you do not have write access to the current `pathdef.m` file.

`status = savepath( ___ )` additionally indicates if the operation is successful, using any of the input arguments in the previous syntaxes. The `status` output is `0` when `savepath` is successful, and `1` otherwise.

Examples

Save Search Path to Specific Folder

Save the current search path to `pathdef.m` located in the folder, `I:/my_matlab_files`.

```
savepath I:/my_matlab_files/pathdef.m
```

Input Arguments

`folderName` — Folder name

string

Folder name, specified as a string. `folderName` can be a relative or absolute path.

Example: `C:\myFolder`

More About

Tips

- To display the paths to all `pathdef.m` files in the current folder and on the current search path, use `which`.

  ```
  which pathdef.m -all
  ```

  The `savepath` command updates the first `pathdef.m` file in this list.
- To save the search path programmatically each time you exit MATLAB, use `savepath` in a `finish.m` file.
- To save a search path and automatically use it in a future session, use the `savepath` *`folderName`*`/pathdef.m` syntax and set the folder specified by `folderName` as the MATLAB startup folder.
- “What Is the MATLAB Search Path?”
- “Running a Script When Exiting”

See Also

`addpath` | `finish` | `path` | `rmpath` | `userpath`

Introduced before R2006a

scatter

Scatter plot

Syntax

```
scatter(x,y)
scatter(x,y,a)
scatter(x,y,a,c)
scatter(___,'filled')
scatter(___,markertype)
scatter(___,Name,Value)

scatter(ax,___)

s = scatter(___)
```

Description

`scatter(x,y)` creates a scatter plot with circles at the locations specified by the vectors `x` and `y`. This type of graph is also known as a bubble plot.

`scatter(x,y,a)` specifies the circle areas. To plot each circle with equal area, specify `a` as a scalar. To plot each circle with a different area, specify `a` as a vector with length equal to the length of x and y.

`scatter(x,y,a,c)` specifies the circle colors. To plot all circles with the same color, specify `c` as a single color string or an RGB triplet. To use varying color, specify `c` as a vector or a three-column matrix of RGB triplets.

`scatter(___,'filled')` fills in the circles. Use the `'filled'` option with any of the input argument combinations in the previous syntaxes.

`scatter(___,markertype)` specifies the marker type.

`scatter(___,Name,Value)` specifies scatter series properties using one or more Name,Value pair arguments. For example, `'LineWidth',2` sets the marker outline width to 2 points.

`scatter(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes. The option `ax` can precede any of the input argument combinations in the previous syntaxes.

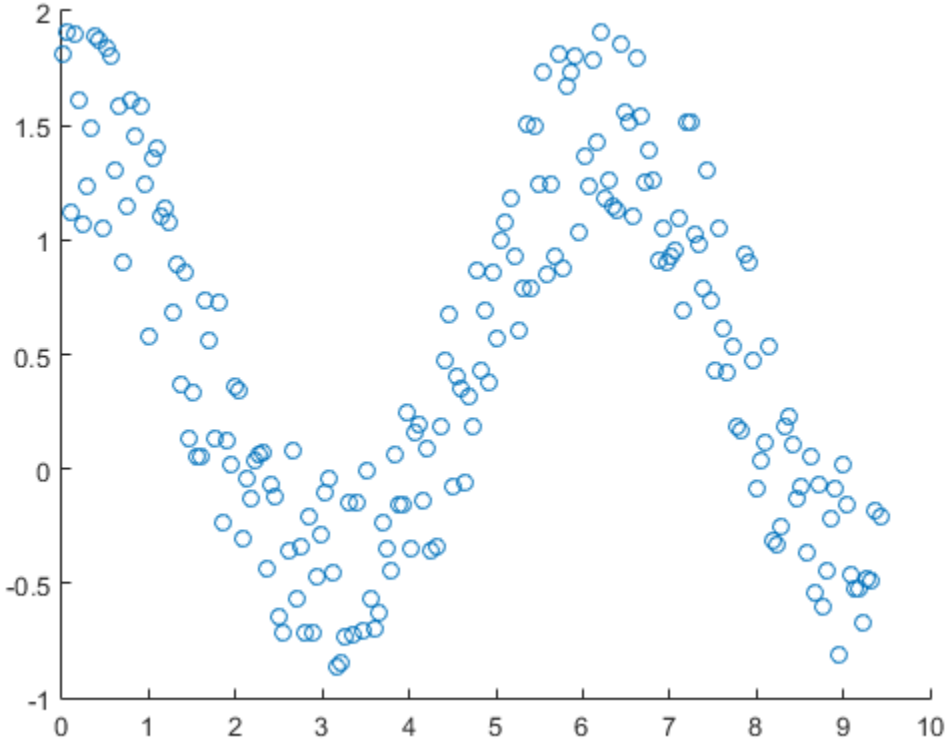
`s = scatter( ___ )` returns the scatter series object. Use `s` to make future modifications to the scatter series after it is created.

Examples

Create Scatter Plot

Create `x` as 200 equally spaced values between 0 and 3π. Create `y` as cosine values with random noise. Then, create a scatter plot.

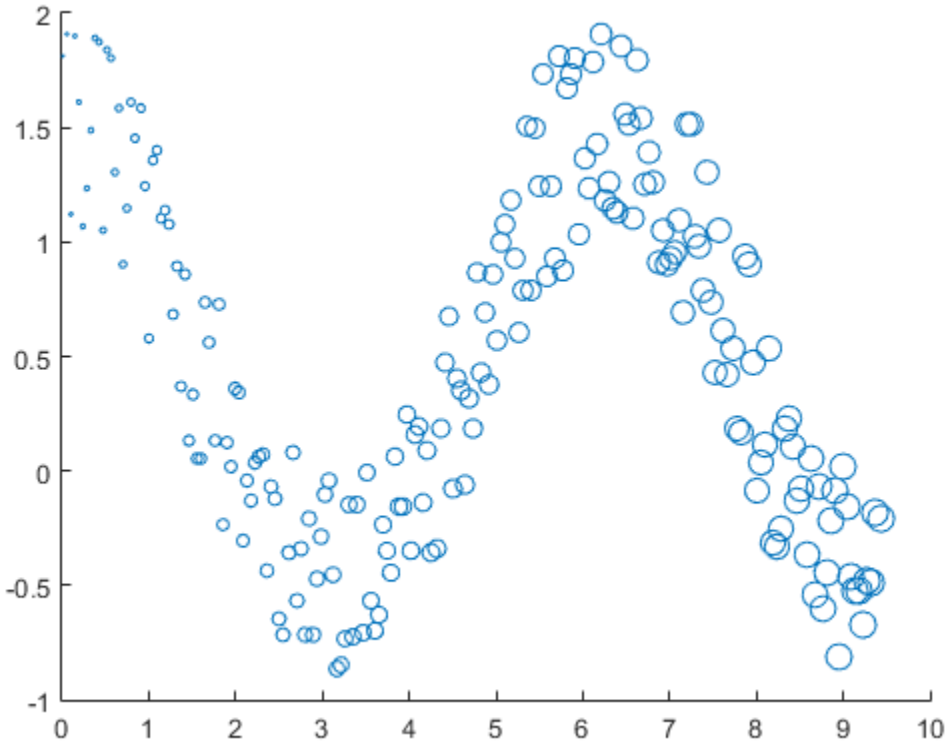
```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
scatter(x,y)
```



Vary Circle Size

Create a scatter plot using circles with different areas.

```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
a = linspace(1,100,200);
scatter(x,y,a)
```

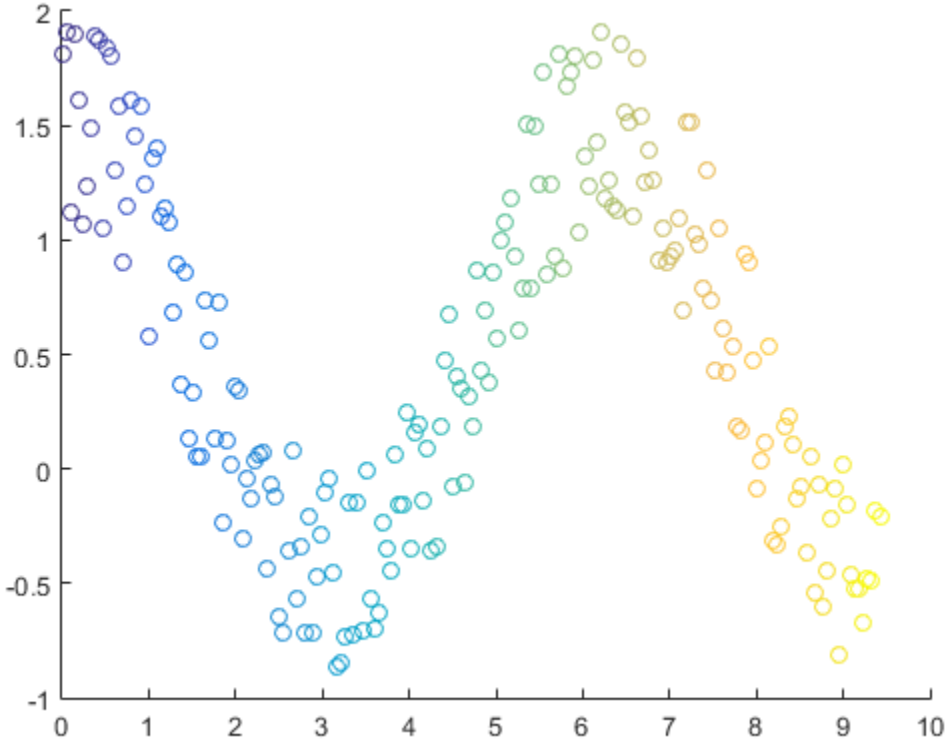


Corresponding elements in `x`, `y`, and `a` determine the location and area of each circle. To plot all circles with the equal area, specify `a` as a numeric scalar.

Vary Circle Color

Create a scatter plot and vary the circle color.

```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
c = linspace(1,10,length(x));
scatter(x,y,[],c)
```

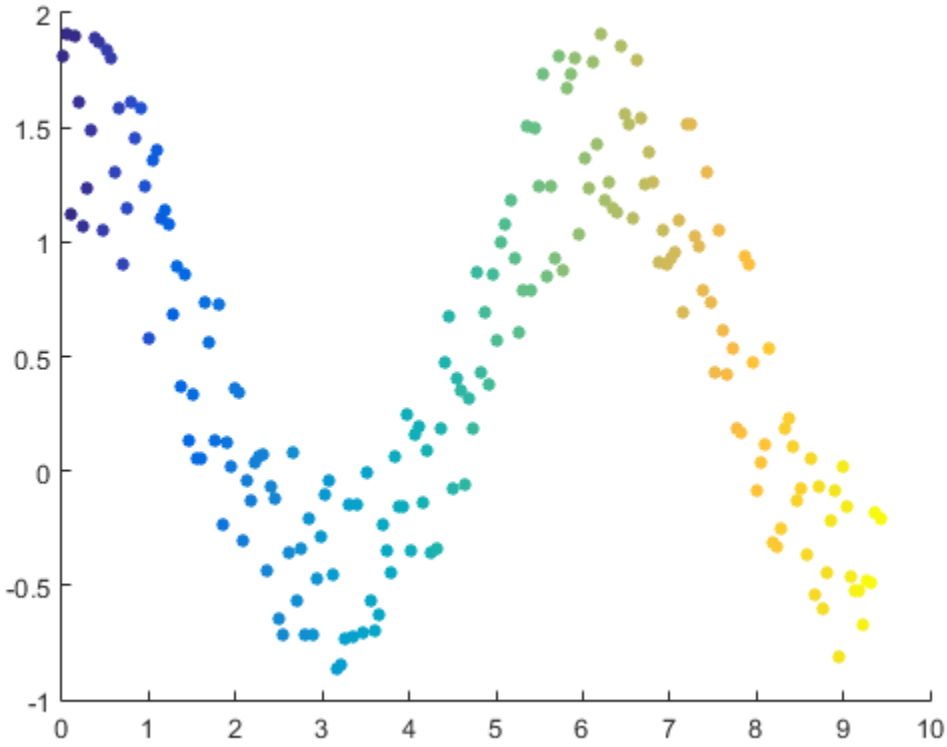


Corresponding elements in `x`, `y`, and `c` determine the location and color of each circle. The `scatter` function maps the elements in `c` to colors in the current colormap.

Fill the Markers

Create a scatter plot and fill in the markers. `scatter` fills each marker using the color of the marker edge.

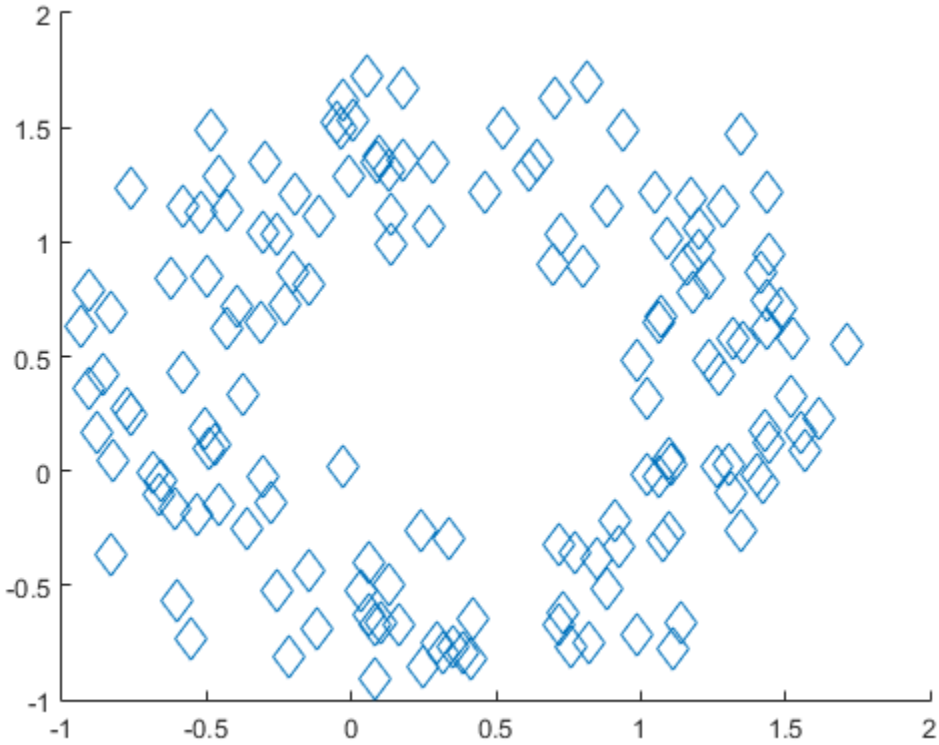
```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
a = 25;
c = linspace(1,10,length(x));
scatter(x,y,a,c,'filled')
```



Specify Marker Symbol

Create vectors `x` and `y` as sine and cosine values with random noise. Then, create a scatter plot and use diamond markers with an area of 140 points squared.

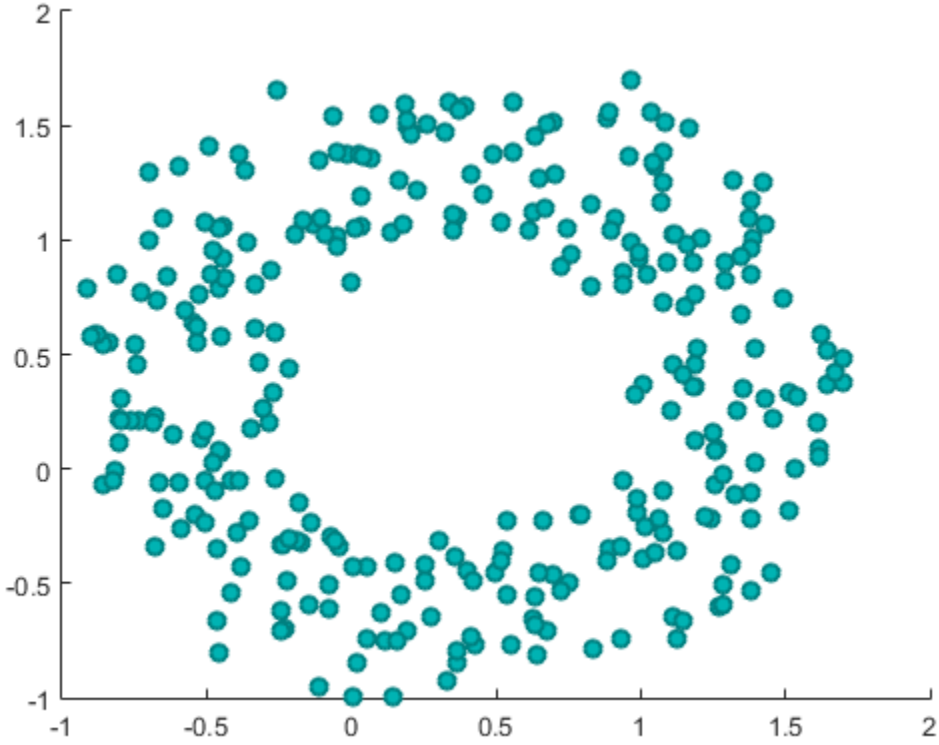
```
theta = linspace(0,2*pi,150);
x = sin(theta) + 0.75*rand(1,150);
y = cos(theta) + 0.75*rand(1,150);
a = 140;
scatter(x,y,a,'d')
```



Change Marker Color and Line Width

Create vectors `x` and `y` as sine and cosine values with random noise. Create a scatter plot and set the marker edge color, marker face color, and line width.

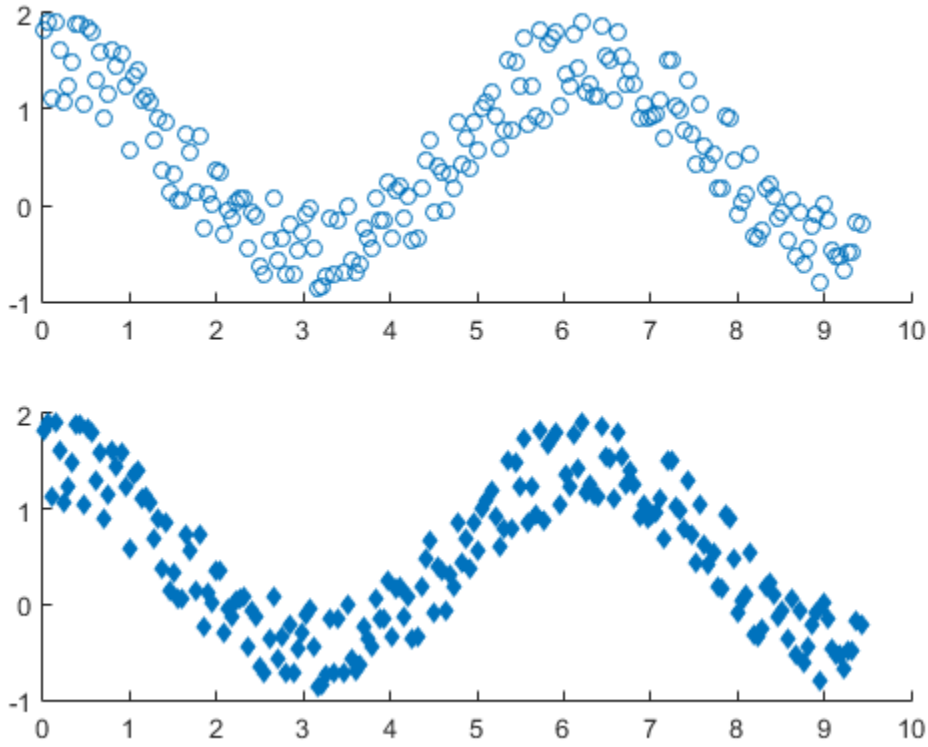
```
theta = linspace(0,2*pi,300);
x = sin(theta) + 0.75*rand(1,300);
y = cos(theta) + 0.75*rand(1,300);
a = 40;
scatter(x,y,a,'MarkerEdgeColor',[0 .5 .5],...
              'MarkerFaceColor',[0 .7 .7],...
              'LineWidth',1.5)
```



Specify Subplot for Scatter Plot

Create a figure with two subplots and add a scatter plot to each subplot. Use filled diamond markers for the scatter plot in the lower subplot.

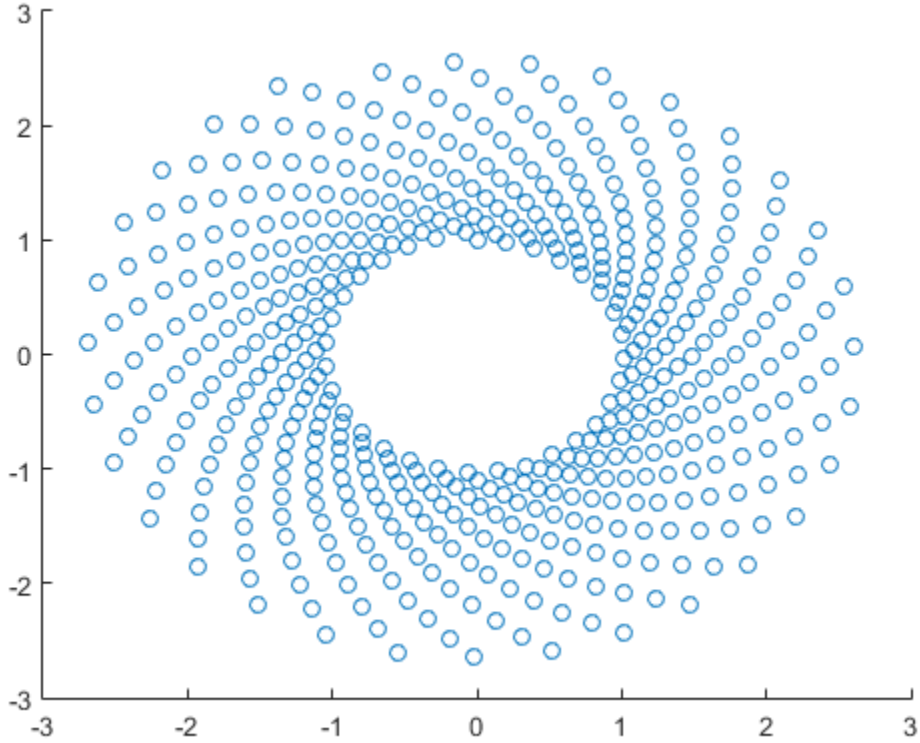
```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
ax1 = subplot(2,1,1);
scatter(ax1,x,y)

ax2 = subplot(2,1,2);
scatter(ax2,x,y,'filled','d')
```



Modify Scatter Series After Creation

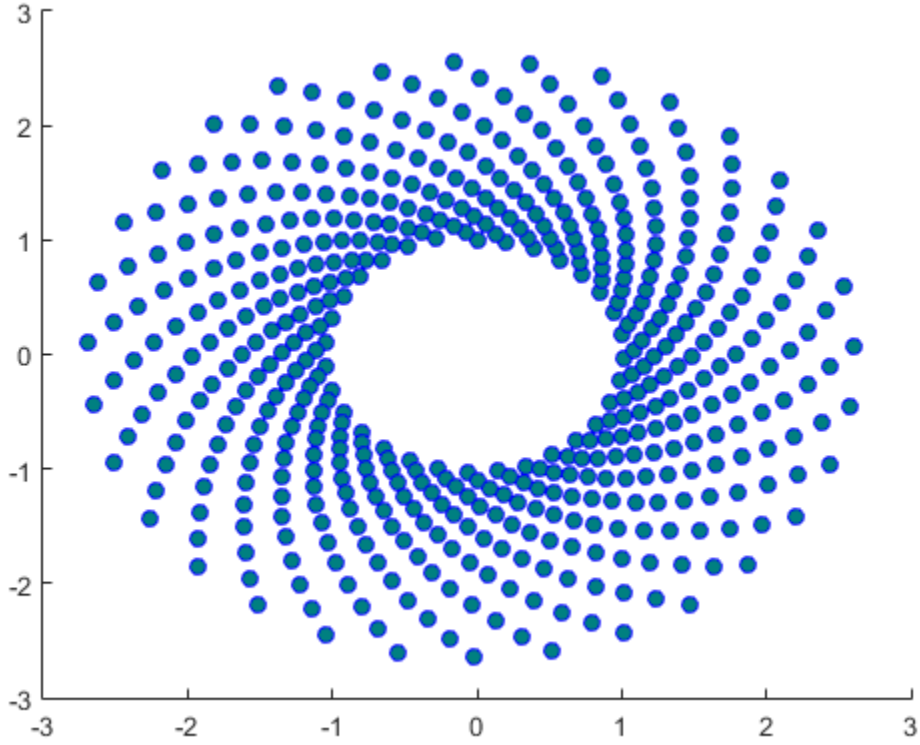
Create a scatter plot and return the scatter series object, `s`.

```
theta = linspace(0,1,500);
x = exp(theta).*sin(100*theta);
y = exp(theta).*cos(100*theta);
s = scatter(x,y);
```



Use `s` to query and set properties of the scatter series after it has been created. Set the line width to `0.6` point. Set the marker edge color to blue. Set the marker face color using an RGB triplet color. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
s.LineWidth = 0.6;
s.MarkerEdgeColor = 'b';
s.MarkerFaceColor = [0 0.5 0.5];
```



Input Arguments

x — x values

vector

x values, specified as a vector. `x` and `y` must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

y — y values

vector

y values, specified as a vector. `x` and `y` must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

a — Marker area

36 (default) | numeric scalar | row or column vector | `[]`

Marker area, specified in one of these forms:

- Numeric scalar — Plot all markers with equal area.
- Row or column vector — Use different areas for each marker. Corresponding elements in `x`, `y`, and `a` determine the location and area of each marker. The length of `a` must equal the length of `x` and `y`.
- `[]` — Use the default area of 36 points squared.

The units for the marker area is points squared.

Example: `50`

Example: `[36 25 25 17 46]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

c — Marker color

`[0 0 1]` (default) | color string | RGB triplet | three-column matrix of RGB triplets | vector

Marker color, specified in one of these forms:

- Color string or RGB triplet — Plot all markers with the same color.
- Three column matrix of RGB triplets — Use different colors for each marker. Each row of the matrix specifies an RGB triplet color for the corresponding marker. The number of rows must equal the length of `x` and `y`.
- Vector — Use different colors for each marker and linearly map values in `c` to the colors in the current colormap. The length of `c` must equal the length of `x` and `y`. To change the colormap for the axes, use the `colormap` function.

  If you have three points in the scatter plot and want the colors to be indices into the colormap, specify `c` as a three-element column vector.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'k'`

Example: `[1 2 3 4]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

`markertype` — Marker type

`'o'` (default) | `'+'` | `'*'` | `'.'` | ...

Marker type, specified as one of the strings listed in this table.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond

String	Marker Symbol
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

ax — Axes object
axes object

Axes object. If you do not specify an axes, then the `scatter` function uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MarkerFaceColor','red'` sets the marker face color to red.

The scatter series properties listed here are only a subset. For a complete list, see Scatter Series Properties.

'MarkerEdgeColor' — Marker outline color
'flat' (default) | 'none' | RGB triplet | color string

Marker outline color, specified as one of these values:

- 'flat' — Colors defined by the CData property.
- 'none' — No color, which makes unfilled markers invisible.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

`[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`'MarkerFaceColor'` — Marker fill color

`'none'` (default) | `'flat'` | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — No color, which makes the interior invisible.
- `'flat'` — Colors defined by the CData property.
- `'auto'` — Same color as the `Color` property for the axes.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.3 0.2 0.1]

Example: 'green'

'LineWidth' — Width of marker edge
0.5 (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: 0.75

Output Arguments

s — Scatter series object
scatter series object

Scatter series object. Use s to access and modify properties of the scatter series after it has been created.

See Also

Functions
hold | plot | scatter3

Properties
Scatter Series Properties

Introduced before R2006a

scatter3

3-D scatter plot

Syntax

```
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(X,Y,Z,S,C)
scatter3(___,'filled')
scatter3(___,markertype)
scatter3(___,Name,Value)

scatter3(ax,___)

h = scatter3(___)
```

Description

`scatter3(X,Y,Z)` displays circles at the locations specified by the vectors `X`, `Y`, and `Z`.

`scatter3(X,Y,Z,S)` draws each circle with the size specified by `S`. To plot each circle with equal size, specify `S` as a scalar. To plot each circle with a specific size, specify `S` as a vector.

`scatter3(X,Y,Z,S,C)` draws each circle with the color specified by `C`.

- If `C` is a color string or an RGB row vector, then all circles are plotted with the specified color.
- If `C` is a three column matrix with the number of rows in `C` equal to the length of `X`, `Y`, and `Z`, then each row of `C` specifies an RGB color value for the corresponding circle.
- If `C` is a vector with length equal to the length of `X`, `Y`, and `Z`, then the values in `C` are linearly mapped to the colors in the current colormap.

`scatter3(___,'filled')` fills in the circles, using any of the input argument combinations in the previous syntaxes.

`scatter3( ___ ,markertype)` specifies the marker type.

`scatter3( ___ ,Name,Value)` specifies scatter series properties using one or more Name,Value pair arguments.

`scatter3(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The `ax` option can precede any of the input argument combinations in the previous syntaxes.

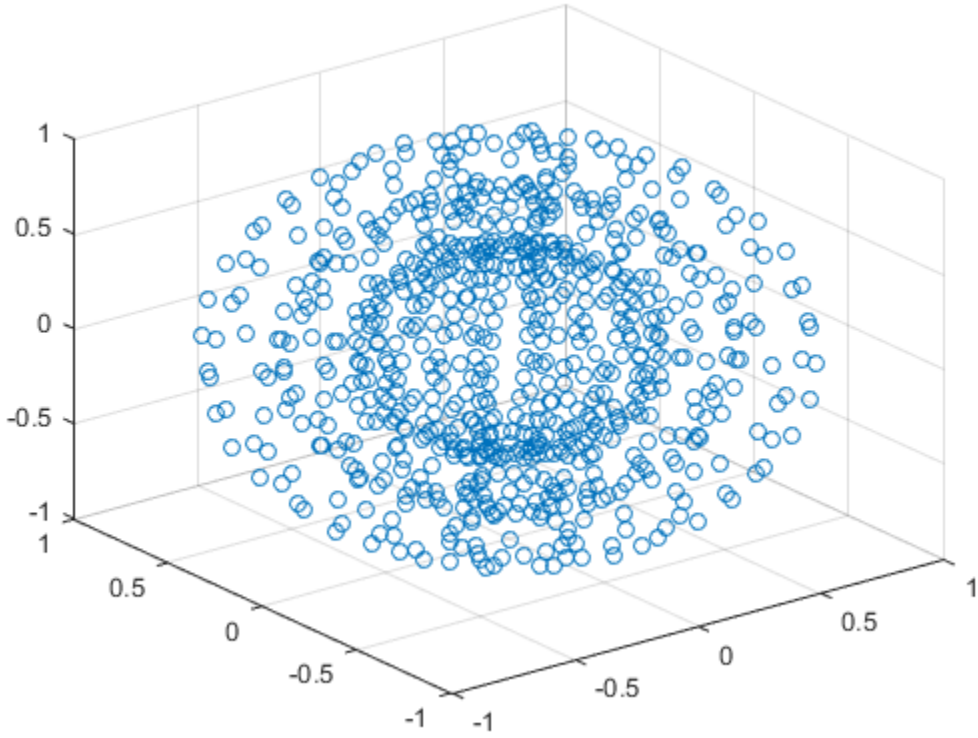
`h = scatter3( ___ )` returns the scatter series object. Use `h` to modify properties of the scatter series after it is created.

Examples

Create 3-D Scatter Plot

Create a 3-D scatter plot. Use `sphere` to define vectors `x`, `y`, and `z`.

```
figure
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
scatter3(x,y,z)
```



Vary Marker Size

Use `sphere` to define vectors `x`, `y`, and `z`.

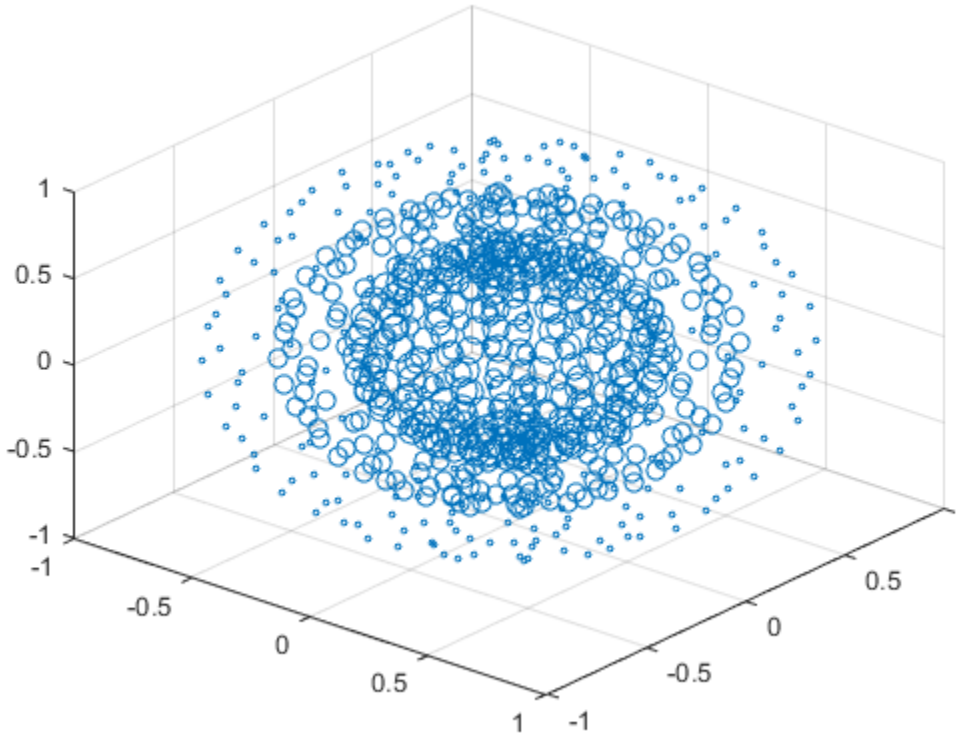
```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Define vector `s` to specify the marker sizes.

```
S = repmat([100,50,5],numel(X),1);
s = S(:);
```

Create a 3-D scatter plot and use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,s)
view(40,35)
```



Corresponding entries in `x`, `y`, `z`, and `s` determine the location and size of each marker.

Vary Marker Color

Use `sphere` to define vectors `x`, `y`, and `z`.

```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
```

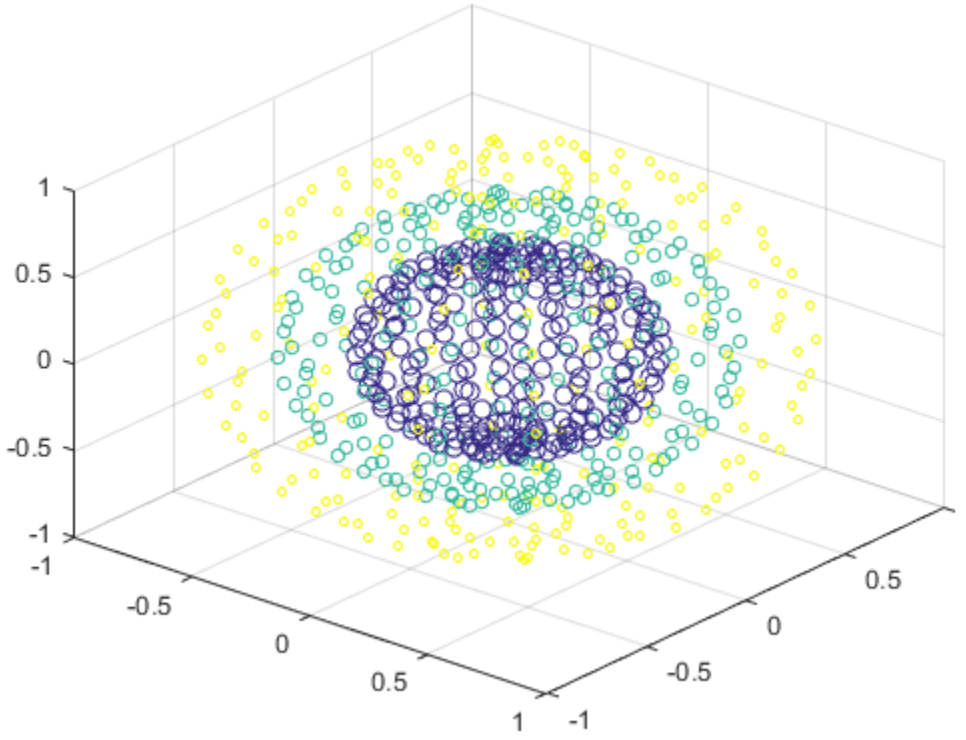
```
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Define vectors `s` and `c` to specify the size and color of each marker.

```
S = repmat([50,25,10],numel(X),1);
C = repmat([1,2,3],numel(X),1);
s = S(:);
c = C(:);
```

Create a 3-D scatter plot and use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,s,c)
view(40,35)
```



Corresponding entries in `x`, `y`, `z`, and `c` determine the location and color of each marker.

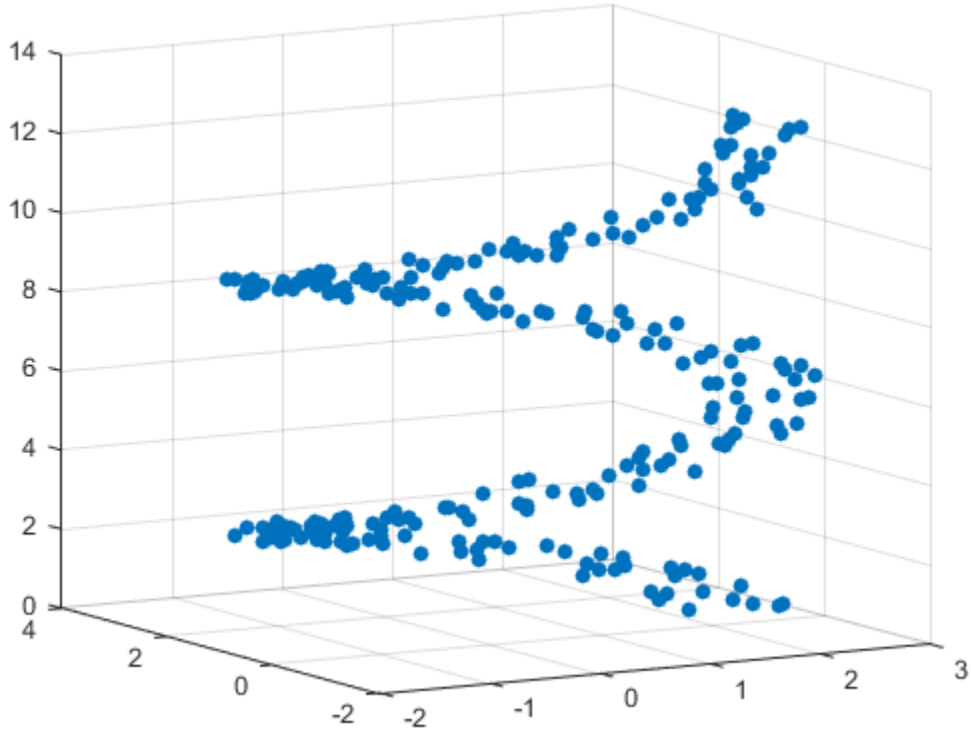
Fill in Markers

Create vectors `x` and `y` as cosine and sine values with random noise.

```
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and fill in the markers. Use `view` to change the angle of the axes in the figure.

```
scatter3(x,y,z,'filled')
view(-30,10)
```



Set Marker Type

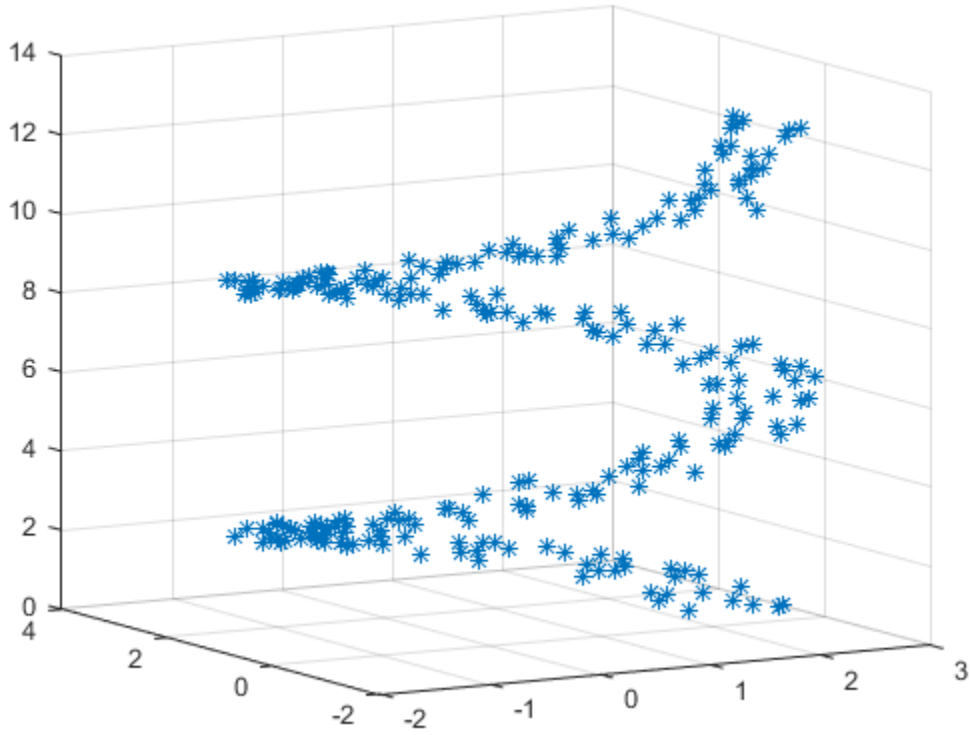
Initialize the random-number generator to make the output of `rand` repeatable. Define vectors `x` and `y` as cosine and sine values with random noise.

```
rng default
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and set the marker type. Use `view` to change the angle of the axes in the figure.

```
figure
```

```
scatter3(x,y,z,'*')
view(-30,10)
```



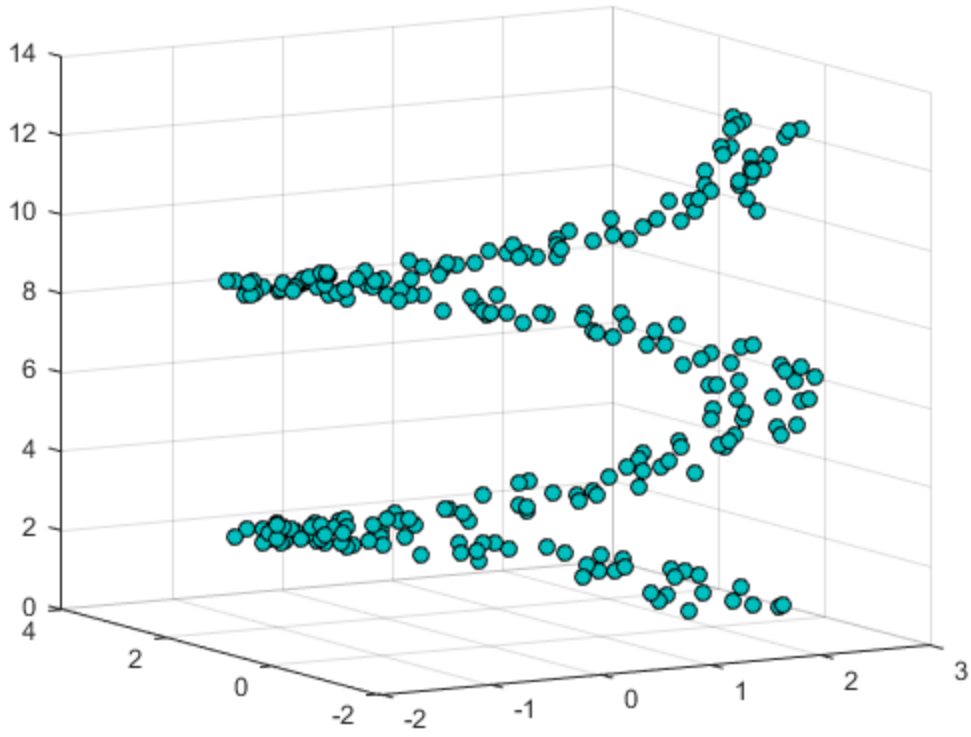
Set Marker Properties

Initialize the random-number generator to make the output of `rand` repeatable. Define vectors `x` and `y` as cosine and sine values with random noise.

```
rng default
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and set the marker edge color and the marker face color. Use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,...
        'MarkerEdgeColor','k',...
        'MarkerFaceColor',[0 .75 .75])
view(-30,10)
```



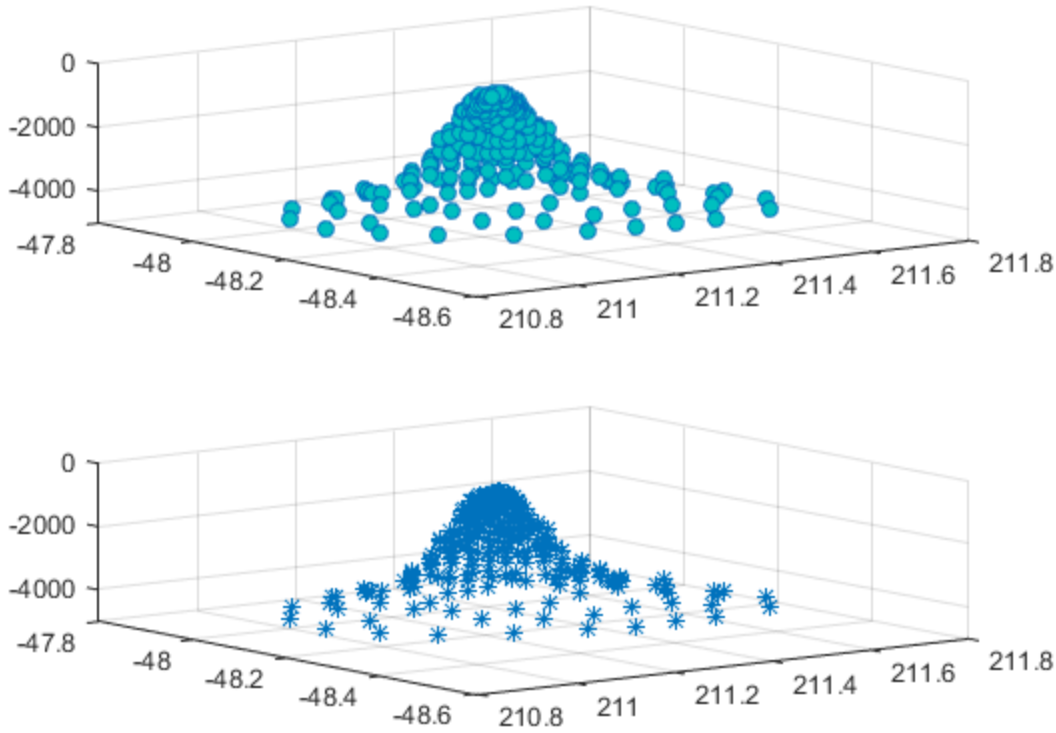
Specify Axes for 3-D Scatter Plot

Load the `seamount` data set to get vectors `x`, `y`, and `z`.

```
load seamount
```

Create a figure with two subplots and return the handles to the two axes in array `hs`. In each subplot, create a 3-D scatter plot. Specify the marker properties for each scatter plot.

```
figure
hs(1) = subplot(2,1,1);
hs(2) = subplot(2,1,2);
scatter3(hs(1),x,y,z,'MarkerFaceColor',[0 .75 .75])
scatter3(hs(2),x,y,z,'*')
```



Set Scatter Series Properties Using Handle

Use the `sphere` function to create vectors `x`, `y`, and `z`.

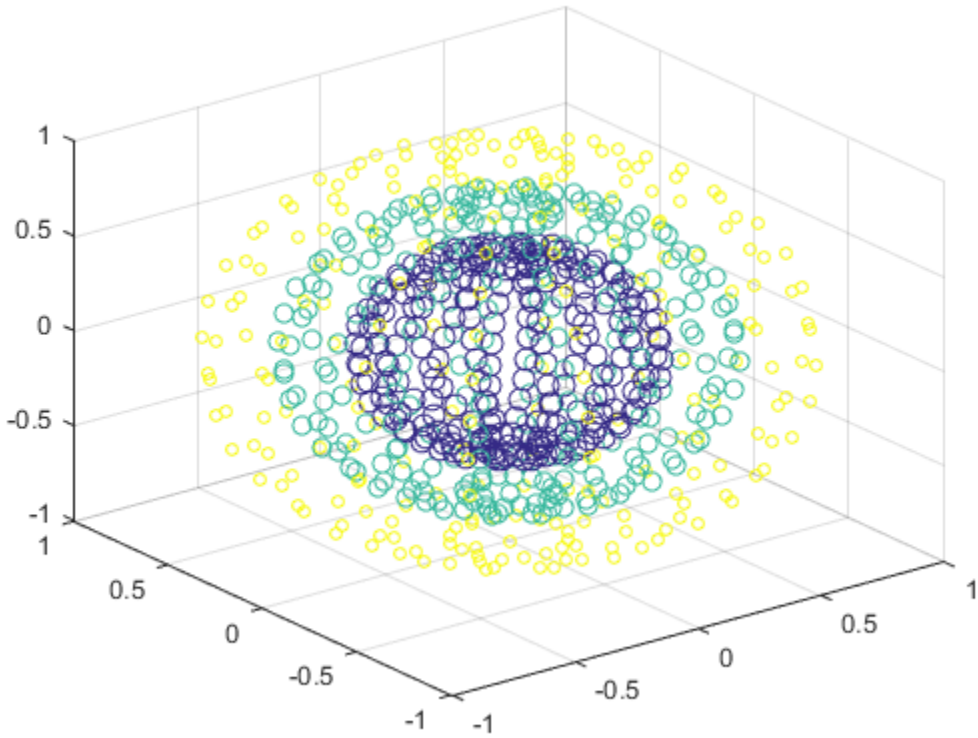
```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Create vectors `s` and `c` to specify the size and color for each marker.

```
S = repmat([70,50,20],numel(X),1);
C = repmat([1,2,3],numel(X),1);
s = S(:);
c = C(:);
```

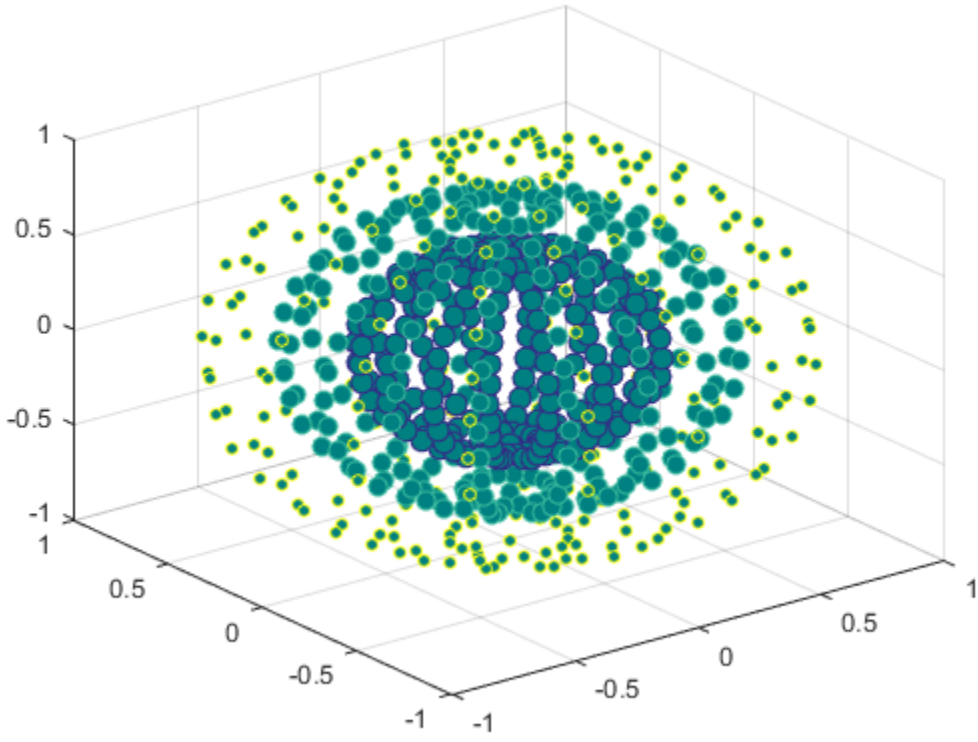
Create a 3-D scatter plot and return the scatter series object.

```
h = scatter3(x,y,z,s,c);
```



Use an RGB triplet color value to set the marker face color. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
h.MarkerFaceColor = [0 0.5 0.5];
```



Input Arguments

X — x values

vector

x values, specified as a vector. `X`, `Y`, and `Z` must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Y — y values

(default) | vector

y values, specified as a vector. `X`, `Y`, and `Z` must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Z — z values

(default) | vector

z values, specified as a vector. `X`, `Y`, and `Z` must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

S — Marker area

36 (default) | scalar | vector | []

Marker area, specified as a scalar, a vector, or `[]`. The values in `S` must be positive. The units for area are points squared.

- If `S` is a scalar, then `scatter3` plots all markers with the specified area.
- If `S` is a row or column vector, then each entry in `S` specifies the area for the corresponding marker. The length of `S` must equal the length of `X`, `Y` and `Z`. Corresponding entries in `X`, `Y`, `Z` and `S` determine the location and area of each marker.
- If `S` is empty, then the default size of 36 points squared is used.

Example: `50`

Example: `[36,25,25,17,46]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

C — Marker color

[0 0 1] (blue) (default) | color string | RGB row vector | three-column matrix of RGB values | vector

Marker color, specified as a color string, an RGB row vector, a three-column matrix of RGB values, or a vector. For an RGB row vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The

intensities must be in the range `[0 1]`. If you have three points in the scatter plot and want the colors to be indices into the colormap, specify `C` as a three-element column vector.

This table lists the predefined colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
`[1 1 0]`	`'y'`	`'yellow'`
`[1 0 1]`	`'m'`	`'magenta'`
`[0 1 1]`	`'c'`	`'cyan'`
`[1 0 0]`	`'r'`	`'red'`
`[0 1 0]`	`'g'`	`'green'`
`[0 0 1]`	`'b'`	`'blue'`
`[1 1 1]`	`'w'`	`'white'`
`[0 0 0]`	`'k'`	`'black'`

Example: `'y'`

Example: `[1,2,3,4]`

Example: `reshape([0,1,0,0,0,1,0.5,1,0.2],3,3)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`markertype` — Marker type

`'o'` (default) | string

Marker type, specified as a string. The table below lists the supported marker types.

Specifier	Marker Type
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square

Specifier	Marker Type
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker

ax — Axes object

axes object

Axes object. If you do not specify an axes, then `scatter3` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MarkerFaceColor','red'` sets the marker face color to red.

The properties listed here are only a subset. For a complete list, see Scatter Series Properties.

'LineWidth' — Width of marker edge

`0.5` (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: `0.75`

'MarkerEdgeColor' — Marker outline color

`'flat'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'flat'` — Colors defined by the CData property.
- `'none'` — No color, which makes unfilled markers invisible.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`'MarkerFaceColor'` — Marker fill color

`'none'` (default) | `'flat'` | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — No color, which makes the interior invisible.
- `'flat'` — Colors defined by the CData property.
- `'auto'` — Same color as the `Color` property for the axes.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Output Arguments

h — Scatter series object

scalar

Scatter series object, returned as a scalar. This is a unique identifier, which you can use to query and modify the properties of the scatter object after it is created.

See Also

Functions

`LineSpec` | `plot3` | `scatter`

Properties

Scatter Series Properties

Introduced before R2006a

Scatter Series Properties

Control scatter series appearance and behavior

Scatter series properties control the appearance and behavior of scatter series object. By changing property values, you can modify certain aspects of the scatter series.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = scatter(1:10,1:10);
m = h.Marker;
h.Marker = '*';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Markers

`Marker` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the strings listed in this table:

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)

String	Marker Symbol
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'flat'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'flat'` — Colors defined by the CData property.
- `'none'` — No color, which makes unfilled markers invisible.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'flat'` | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — No color, which makes the interior invisible.

- `'flat'` — Colors defined by the CData property.
- `'auto'` — Same color as the `Color` property for the axes.
- RGB triplet or color string — Specify a custom color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`LineWidth` — Width of marker edge

`0.5` (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: `0.75`

Data

`XData` — x values

`[ ]` (default) | scalar | vector

x values, specified as a scalar or a vector. The scatter plot displays an individual marker for each value in `XData`.

The input argument `X` to the `scatter` and `scatter3` functions set the *x* values. `XData` and `YData` must have equal lengths.

Example: `[1 2 4 2 6]`

`YData` — y values

[ ] (default) | scalar | vector

y values, specified as a scalar or a vector. The scatter plot displays an individual marker for each value in `YData`.

The input argument `Y` to the `scatter` and `scatter3` functions set the *y* values. `XData` and `YData` must have equal lengths.

Example: `[1 3 3 4 6]`

`ZData` — z values

[ ] (default) | scalar | vector

z values, specified as a scalar or a vector.

- For 2-D scatter plots, `ZData` is empty by default.
- For 3-D scatter plots, the input argument `Z` to the `scatter3` function sets the *z* values. `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1 2 2 1 0]`

`CData` — Marker colors

[ ] (default) | RGB triplet | matrix of RGB triplets | vector

Marker colors, specified as one of these values:

- RGB triplet — Use the same color for all the markers in the plot. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.5 0.6 0.7]`.
- Three-column matrix of RGB triplets — Use a different color for each marker in the plot. Each row of the matrix defines one color. The number of rows must equal the number of markers.
- Vector — Use a different color for each marker in the plot. Specify `CData` as a vector the same length as `XData`. Linearly map the values in the vector to the colors in the current colormap.

Example: `[1 0 0; 0 1 0; 0 0 1]`

`SizeData` — Marker sizes

`[ ]` (default) | scalar | vector

Marker sizes, specified in one of these forms:

- Scalar — Use the same size for all of the markers.
- Vector — Use a different size for each marker. Specify `SizeData` as a vector the same length as `XData`.

Specify the values in point units, where one point equals 1/72 inch. To specify a marker that has an area of one square inch, use a value of 72^2.

Example: `50`

`XDataSource` — Variable linked to `XData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZDataSource` — Variable linked to `ZData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `ZData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

`CDataSource` — Variable linked to `CData`

`''` | string containing MATLAB workspace variable

Variable linked to `CData`, specified as a string containing a MATLAB workspace variable. MATLAB evaluates the variable in the base workspace to generate the `CData`.

By default, there is no linked variable so the value is an empty string. If you link a variable, then MATLAB does not update the `CData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`SizeDataSource` — Variable linked to `SizeData`

`''` | string containing MATLAB workspace variable

Variable linked to `SizeData`, specified as a string containing a MATLAB workspace variable. MATLAB evaluates the variable in the base workspace to generate the `SizeData`.

By default, there is no linked variable so the value is an empty string. If you link a variable, then MATLAB does not update the `SizeData` values. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Visibility

`Visible` — Visibility of scatter series

`'on'` (default) | `'off'`

Visibility of scatter series, specified as one of these values:

- `'on'` — Display the scatter series.
- `'off'` — Hide the scatter series without deleting it. You still can access the properties of an invisible scatter series object.

`Clipping` — Clipping of scatter series to axes limits

`'on'` (default) | `'off'`

Clipping of scatter series to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the scatter series that are outside the axes limits.
- `'off'` — Display the entire scatter series, even if parts of it appear outside the axes limits. Parts of the scatter series might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the scatter series that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'scatter'`

Type of graphics object, returned as `'scatter'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Tag to associate with scatter series

`''` (default) | string

Tag to associate with the scatter series, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with scatter series

`[ ]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the scatter series object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the scatter series.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the scatter series object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the scatter series from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the scatter series object in the legend as one entry (default).
   - `'off'` — Do not include the scatter series object in the legend.
   - `'children'` — Include only children of the scatter series object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of scatter series

axes object | group object | transform object

Parent of scatter series, specified as an axes, group, or transform object.

`Children` — Children of scatter series

empty `GraphicsPlaceholder` array

The scatter series has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of scatter series object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The scatter series object handle is always visible.
- `'off'` — The scatter series object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The scatter series object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the scatter series at the command-line, but allows callback functions to access it.

If the scatter series object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the scatter series. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The scatter series object — You can access properties of the scatter series object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the scatter series. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the scatter series when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the scatter series.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the scatter series that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the scatter series responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the scatter series passes the click to the object below it in the current view of the figure window. The HitTest property of the scatter series has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the scatter series. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the scatter series that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the scatter series object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for scatter series

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for scatter series, specified as one of these values:

- `'off'` — Click the scatter series plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the scatter series plot to select it, that is, anywhere within the rectangle that encloses the scatter series plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the scatter series is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

BusyAction — Callback queuing

'queue' (default) | 'cancel'

Callback queuing specified as 'queue' or 'cancel'. The BusyAction property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the ButtonDownFcn callback of the scatter series tries to interrupt a running callback that cannot be interrupted, then the BusyAction property determines if it is discarded or put in the queue. Specify the BusyAction property as one of these values:

- 'queue' — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- 'cancel' — Discard the interrupting callback.

Creation and Deletion Control

CreateFcn — Creation callback

'' (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the scatter series. Setting the `CreateFcn` property on an existing scatter series has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during scatter series creation. MATLAB executes the callback after creating the scatter series and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The scatter series object — You can access properties of the scatter series object from within the callback function. You also can access the scatter series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the scatter series. MATLAB executes the callback before destroying the scatter series so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The scatter series object — You can access properties of the scatter series object from within the callback function. You also can access the scatter series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.

- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of scatter series

`'off'` (default) | `'on'`

Deletion status of scatter series, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the scatter series begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the scatter series no longer exists.

Check the value of the BeingDeleted property to verify that the scatter series is not about to be deleted before querying or modifying it.

See Also

`scatter` | `scatter3`

More About

- "Access Property Values"
- "Graphics Object Properties"

schur

Schur decomposition

Syntax

```
T = schur(A)
T = schur(A,flag)
[U,T] = schur(A,...)
```

Description

The `schur` command computes the Schur form of a matrix.

`T = schur(A)` returns the Schur matrix `T`.

`T = schur(A,flag)` for real matrix A, returns a Schur matrix `T` in one of two forms depending on the value of `flag`:

`'complex'`	`T` is triangular and is complex if `A` is real and has complex eigenvalues.
`'real'`	`T` has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. `'real'` is the default when `A` is real.

If `A` is complex, `schur` returns the complex Schur form in matrix `T` and `flag` is ignored. The complex Schur form is upper triangular with the eigenvalues of `A` on the diagonal.

The function `rsf2csf` converts the real Schur form to the complex Schur form.

`[U,T] = schur(A,...)` also returns a unitary matrix `U` so that `A = U*T*U'` and `U'*U = eye(size(A))`.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H = [ -149    -50   -154
       537    180    546
       -27     -9    -25 ]
```

Its Schur form is

```
schur(H)

ans =
     1.0000   -7.1119 -815.8706
          0    2.0000  -55.0236
          0         0    3.0000
```

The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

See Also

`eig` | `hess` | `qz` | `rsf2csf`

Introduced before R2006a

script

Sequence of MATLAB statements in file

Description

A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, you can obtain subsequent MATLAB input from the file. Script files have a filename extension of `.m`.

Scripts are the simplest kind of MATLAB program. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, so you can use them in further computations. In addition, scripts can produce graphical output using commands like `plot`.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any MATLAB program, scripts can contain comments. Any text following a percent sign (`%`) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also

`echo` | `function` | `type`

Introduced before R2006a

scatteredInterpolant class

Scattered data interpolation

Description

Use `scatteredInterpolant` to perform interpolation on a 2-D or 3-D "Scattered Data" on page 1-7323 set. For example, you can pass a set of `(x,y)` points and values, `v`, to `scatteredInterpolant`, and it returns a surface of the form $v = F(x, y)$. This surface always passes through the sample values at the point locations. You can evaluate this surface at any query point, `(xq,yq)`, to produce an interpolated value, `vq`.

Use `scatteredInterpolant` to create the "Interpolant" on page 1-7323, `F`. Then, you can evaluate `F` at specific points using any of the following syntaxes:

- `Vq = F(Pq)` specifies the query points in the matrix `Pq`. Each row in `Pq` contains the coordinates of a query point.
- `Vq = F(Xq,Yq)` and `Vq = F(Xq,Yq,Zq)` specify the query points as two or three matrices of equal size.
- `Vq = F({xq,yq})` and `Vq = F({xq,yq,zq})` specify the query points as "Grid Vectors" on page 1-7323. The interpolated values are returned in `Vq`. Use this syntax to conserve memory when you want to query a large grid of points.

Construction

`F = scatteredInterpolant(x,y,v)` creates an interpolant that fits a surface of the form $v = F(x,y)$. Vectors `x` and `y` specify the `(x,y)` coordinates of the sample points. `v` is a vector that contains the sample values associated with the points, `(x,y)`.

`F = scatteredInterpolant(x,y,z,v)` creates a 3-D interpolant of the form $v = F(x,y,z)$.

`F = scatteredInterpolant(P,v)` specifies the coordinates of the sample points as an array. The rows of `P` contain the (x, y) or (x, y, z) coordinates for the values in `v`.

`F = scatteredInterpolant(___,Method)` specifies a string that describes an interpolation method: `'nearest'`, `'linear'`, or `'natural'`. Specify Method as the last input argument in any of the first three syntaxes.

`F = scatteredInterpolant(___,Method,ExtrapolationMethod)` specifies both the interpolation and extrapolation methods as strings. `Method` can be one of three strings: `'nearest'`, `'linear'`, or `'natural'`. Specify `ExtrapolationMethod` as one of the following strings: `'nearest'`, `'linear'`, or `'none'`. Pass Method and ExtrapolationMethod together as the last two input arguments in any of the first three syntaxes.

`F = scatteredInterpolant()` creates an empty scattered data interpolant. Use `F.Points = P` to initialize `F` with the points in matrix `P`. Use `F.Values = v` to initialize `F` with the values in `v`.

Input Arguments

`x`

Sample points *x*-coordinates, specified as a vector of the same size as `v`.

`y`

Sample points *y*-coordinates, specified as a vector of the same size as `v`.

`z`

Sample points *z*-coordinates, specified as a vector of the same size as `v`.

`P`

Sample points array, specified as an `m`-by-`n` matrix, where `m` is the number of points and `n` is the dimension of the space where the points reside. Each row of `P` contains the (*x*, *y*) or (*x*, *y*, *z*) coordinates of a sample point.

`v`

Sample values, specified as a vector that defines the function values, $v = F(x,y)$, at the sample points.

`Method`

Interpolation method, specified as one of the following options.

Method String	Description	Continuity
`'linear'` (default)	Linear interpolation.	C^0

Method String	Description	Continuity
`'nearest'`	Nearest neighbor interpolation.	Discontinuous
`'natural'`	Natural neighbor interpolation.	C^1 except at sample points

`ExtrapolationMethod`

Extrapolation method, specified as one of these strings.

ExtrapolationMethod String	Description
`'linear'`	Linear extrapolation based on boundary gradients. Default when `Method` is `'linear'` or `'natural'`.
`'nearest'`	Nearest neighbor extrapolation. This method evaluates to the value of the nearest neighbor on the boundary. Default when `Method = 'nearest'`.
`'none'`	No extrapolation. Any queries outside the convex hull of `F.Points` return `NaN`.

Properties

`Points`

Array of sample points (locations) for the values in `F.Values`. Each row of `F.Points` contains the (x, y) or (x, y, z) coordinates of a sample point.

`Values`

Vector of values associated with each point in `F.Points`.

`Method`

A string specifying the method used to interpolate the data: `'nearest'`, `'linear'`, or `'natural'`.

`ExtrapolationMethod`

A string specifying the method used to extrapolate the data: `'nearest'`, `'linear'`, or `'none'`. A value of `'none'` indicates that extrapolation is disabled.

Definitions

Interpolant

Interpolating function that you can evaluate at query locations.

Scattered Data

A set of points that have no structure among their relative locations.

Full Grid

A grid represented as a set of arrays. For example, you can create a full grid using `ndgrid`.

Grid Vectors

A set of vectors that serve as a compact representation of a grid in `ndgrid` format. For example, `[X,Y] = ndgrid(xg,yg)` returns a full grid in the matrices `X` and `Y`. You can represent the same grid using the grid vectors, `xg` and `yg`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Indexing

`scatteredInterpolant` supports index-based editing of the properties of F. You can add or remove points in `F.Points` and update the corresponding values in `F.Values`.

For example, `F.Points(5,:) = []` removes the fifth point, and `F.Values(5) = []` removes the corresponding value.

Examples

2-D Interpolation

Define 200 random points.

```
xy = -2.5 + 5*gallery('uniformdata',[200 2],0);
x = xy(:,1);
y = xy(:,2);
```

Sample an exponential function. These are the sample values for the interpolant.

```
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant.

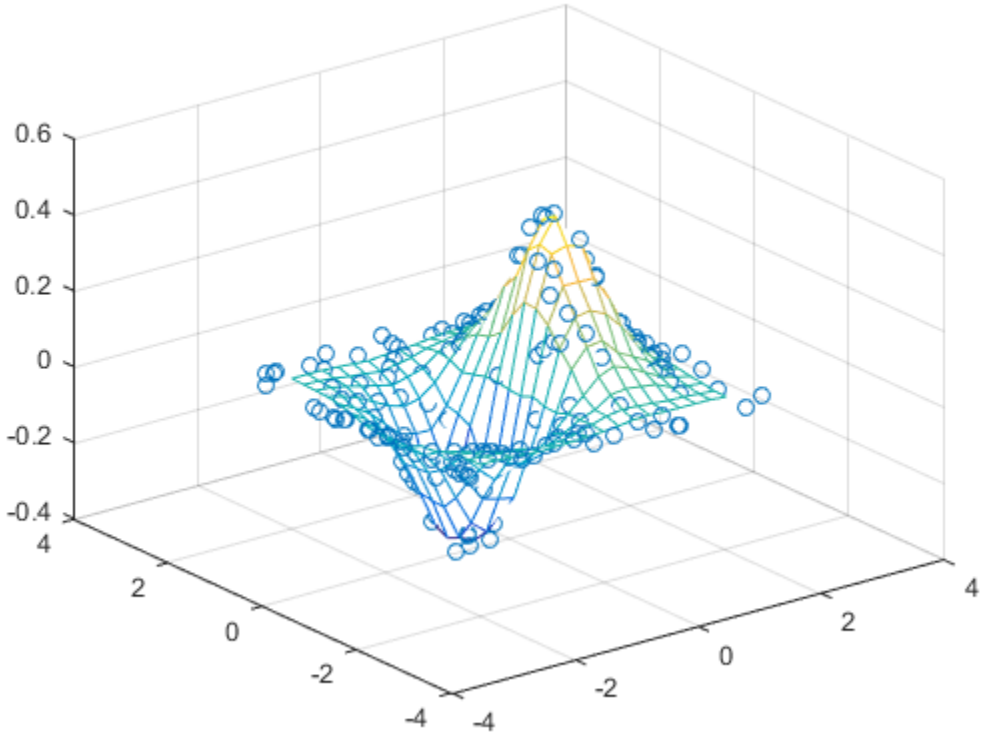
```
F = scatteredInterpolant(x,y,v);
```

Evaluate the interpolant at query locations (xq, yq).

```
ti = -2:.25:2;
[xq,yq] = meshgrid(ti,ti);
vq = F(xq,yq);
```

Plot the result.

```
figure
mesh(xq,yq,vq);
hold on;
plot3(x,y,v,'o');
hold off;
```



2-D Extrapolation

Query the interpolant at a single point outside the convex hull using nearest neighbor extrapolation.

Define a matrix of 200 random points.

```
P = -2.5 + 5*gallery('uniformdata',[200 2],0);
```

Sample an exponential function. These are the sample values for the interpolant.

```
x = P(:,1);
y = P(:,2);
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant, specifying linear interpolation and nearest neighbor extrapolation.

```
F = scatteredInterpolant(P,v,'linear','nearest')
```

```
F =

  scatteredInterpolant with properties:

                Points: [200x2 double]
                Values: [200x1 double]
                Method: 'linear'
    ExtrapolationMethod: 'nearest'
```

Evaluate the interpolant outside the convex hull.

```
vq = F(3.0,-1.5)
```

```
vq =

    0.0031
```

Disable extrapolation and evaluate `F` at the same point.

```
F.ExtrapolationMethod = 'none';
vq = F(3.0,-1.5)
```

```
vq =

   NaN
```

Replacement of Sample Values

Replace the elements in the `Values` property when you want to change the values at the sample points. You get immediate results when you evaluate the new interpolant because the original triangulation has not changed.

Create 50 random points.

```
x = -2.5 + 5*gallery('uniformdata',[50 1],0);
y = -2.5 + 5*gallery('uniformdata',[50 1],1);
```

Sample an exponential function. These are the sample values for the interpolant.

```
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant.

```
F = scatteredInterpolant(x,y,v)
```

```
F =

  scatteredInterpolant with properties:

                Points: [50x2 double]
                Values: [50x1 double]
                Method: 'linear'
    ExtrapolationMethod: 'linear'
```

Evaluate the interpolant at `(1.40,1.90)`.

```
F(1.40,1.90)
```

```
ans =

    0.0029
```

Change the interpolant sample values.

```
vnew = x.^2 + y.^2;
F.Values = vnew;
```

Evaluate the interpolant at `(1.40,1.90)`.

```
F(1.40,1.90)
```

```
ans =

    6.1109
```

See Also

`griddedInterpolant` | `interp1` | `interp2` | `interp3` | `meshgrid` | `ndgrid`

How To

- Class Attributes
- Property Attributes
- “Interpolating Gridded Data”

sec

Secant of angle in radians

Syntax

```
Y = sec(X)
```

Description

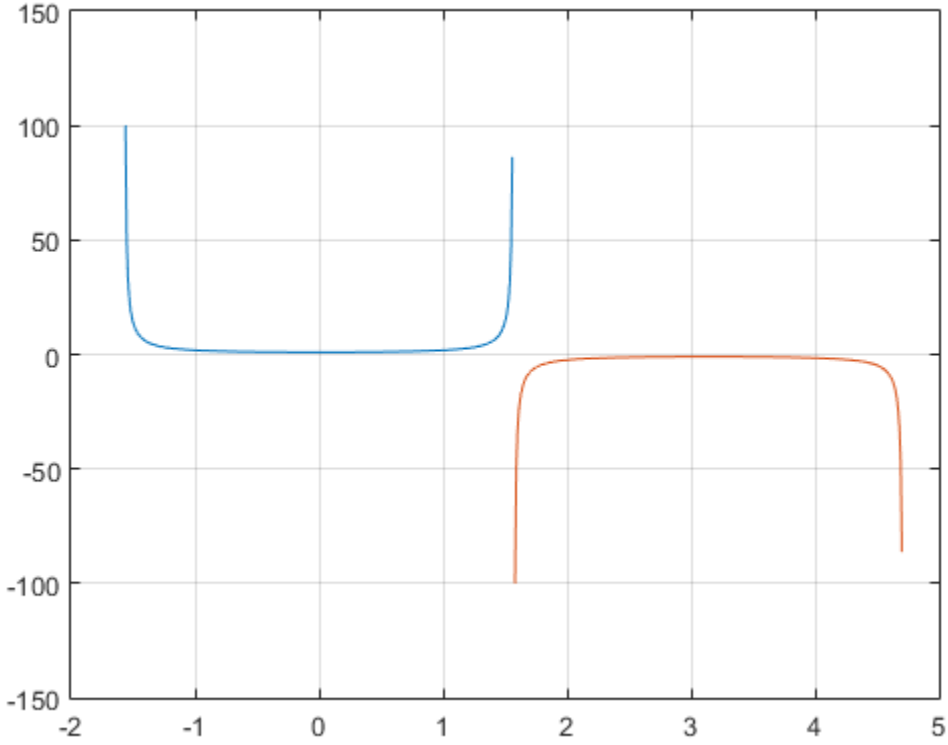
`Y = sec(X)` returns the secant of the elements of `X`. The `sec` function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval [-Inf, Inf], `sec` returns real values in the interval [-Inf ,-1] and [1,Inf]. For complex values of X, `sec` returns complex values. All angles are in radians.

Examples

Plot Secant Function

Plot the secant over the domain $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$.

```
x1 = -pi/2+0.01:0.01:pi/2-0.01;
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;
plot(x1,sec(x1),x2,sec(x2)), grid on
```



Secant of Vector of Complex Angles

Calculate the secant of the complex angles in vector `x`.

```
x = [-i pi+i*pi/2 -1+i*4];
y = sec(x)
```

```
y =

   0.6481 + 0.0000i  -0.3985 + 0.0000i   0.0198 - 0.0308i
```

Input Arguments

X — Input angle in radians

number | vector | matrix | multidimensional array

Input angle in radians, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Secant of input angle

scalar value | vector | matrix | N-D array

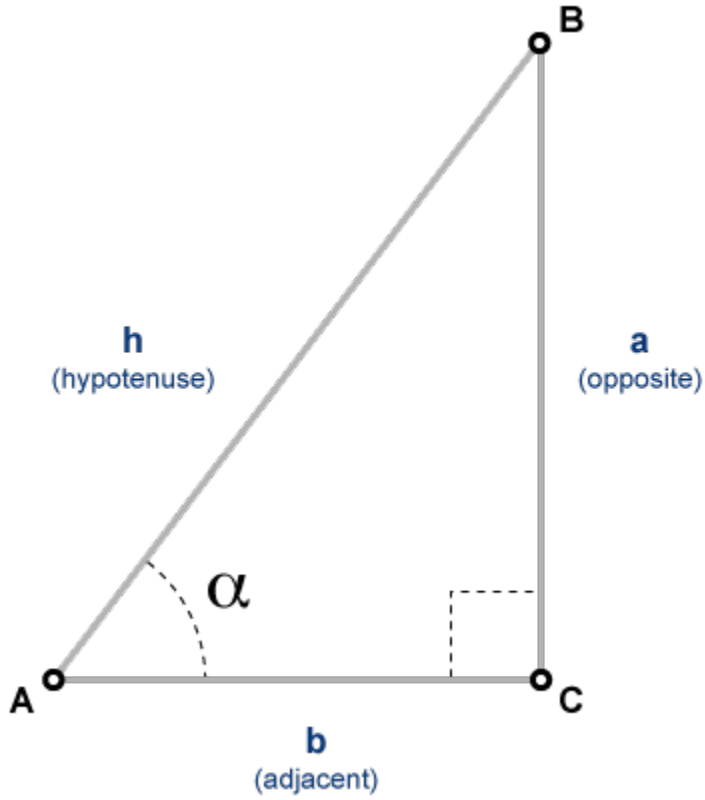
Secant of input angle, returned as real-valued or complex-valued scalar value, vector, matrix or N-D array.

More About

Secant Function

The secant of an angle, α, defined with reference to a right angled triangle is

$$\sec(\alpha) = \frac{1}{\cos(\alpha)} = \frac{\text{hypotenuse}}{\text{adjacent side}} = \frac{h}{b}.$$



The secant of a complex angle, α, is

$$\sec(\alpha) = \frac{2}{e^{i\alpha} + e^{-i\alpha}}.$$

See Also

asec | asecd | asech | secd | sech

Introduced before R2006a

secd

Secant of argument in degrees

Syntax

```
Y = secd(X)
```

Description

`Y = secd(X)` returns the secant of the elements of `X`, which are expressed in degrees.

Examples

Secant of 90 degrees compared to secant of π/2 radians

```
secd(90)
```

```
ans =

   Inf
```

```
sec(pi/2)
```

```
ans =

   1.6331e+16
```

`secd(90)` is infinite, whereas `sec(pi/2)` is large but finite.

Secant of vector of complex angles, specified in degrees

```
z = [35+i 15+2i 10+3i];
y = secd(z)
```

```
y =
```

```
   1.2204 + 0.0149i   1.0346 + 0.0097i   1.0140 + 0.0094i
```

Input Arguments

X — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `secd` operation is element-wise when `X` is nonscalar.

Example:

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Secant of angle

scalar value | vector | matrix | N-D array

Secant of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

See Also

`asec` | `asecd` | `sec`

Introduced before R2006a

sech

Hyperbolic secant

Syntax

```
Y = sech(X)
```

Description

The `sech` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

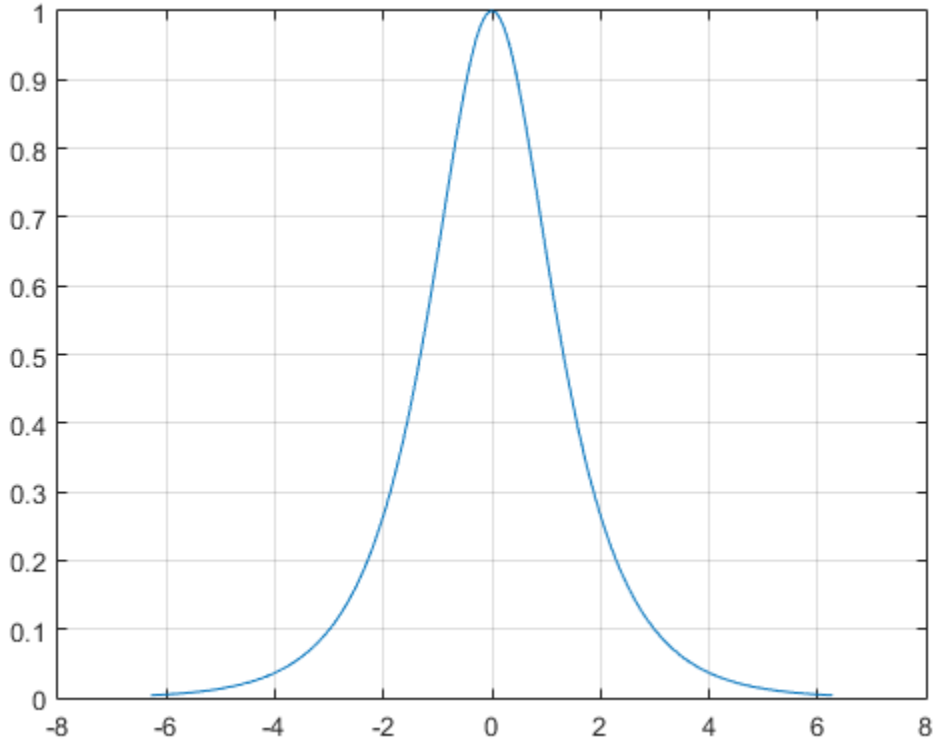
`Y = sech(X)` returns an array the same size as `X` containing the hyperbolic secant of the elements of `X`.

Examples

Graph of Hyperbolic Secant Function

Graph the hyperbolic secant over the domain $-2\pi \le x \le 2\pi$.

```
x = -2*pi:0.01:2*pi;
plot(x,sech(x)), grid on
```



More About

Hyperbolic Secant

The hyperbolic secant of z is

$$\operatorname{sech}(z) = \frac{1}{\cosh(z)}.$$

See Also

`asech` | `sec` | `sinh` | `cosh`

Introduced before R2006a

second

Second number

Syntax

```
s = second(t)
s = second(t,secondType)
```

Description

`s = second(t)` returns the second values, including a fractional part, for the datetime values in `t`. The `s` output is a `double` array the same size as `t` and contains values from 0 to less than 60.

For datetime values whose time zone is `UTCLeapSeconds`, the `s` output can contain a value between 60 and 61 for times that fall during a leap second occurrence.

`s = second(t,secondType)` returns the type of second number specified by `secondType`.

The `second` function returns the second numbers of datetime values. To assign second values to datetime array `t`, use `t.Second` and modify the `Second` property.

Examples

Find Second Number of Datetime Values

```
t1 = datetime('now','Format','dd-MMM-yyyy HH:mm:ss.SSS');
t = t1 + seconds(30:15:60)
```

```
t =

Columns 1 through 2

   23-Feb-2015 09:56:07.528   23-Feb-2015 09:56:22.528
```

```
Columns 3 through 3

   23-Feb-2015 09:56:37.528

s = second(t)

s =

    7.5280   22.5280   37.5280
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`secondType` — Type of second values

`'secondofminute'` (default) | `'secondofday'`

Type of second values, specified as either `'secondofminute'` or `'secondofday'`.

- If `secondType` is `'secondofminute'`, then `second` returns the second of the minute.
- If `secondType` is `'secondofday'`, then `second` returns the second of the day, which (except for leap seconds) is in the range [0, 86400).

See Also

datetime Properties | `hms` | `hour` | `minute` | `timeofday`

Introduced in R2014b

seconds

Duration in seconds

Syntax

```
S = seconds(X)
```

Description

`S = seconds(X)` returns an array of seconds equivalent to the values in `X`.

- If `X` is a numeric array, then `S` is a `duration` array in units of seconds.
- If `X` is a `duration` array, then `S` is a `double` array with each element equal to the number of seconds in the corresponding element of X.

The `seconds` function converts between `duration` and `double` values. To display a duration in units of seconds, set its `Format` property to `'s'`.

Examples

Create Duration Array of Seconds

```
X =  magic(4);
S = seconds(X)
```

```
S =

   16 secs    2 secs    3 secs   13 secs
    5 secs   11 secs   10 secs    8 secs
    9 secs    7 secs    6 secs   12 secs
    4 secs   14 secs   15 secs     1 sec
```

Convert Durations to Numeric Array of Seconds

Create a `duration` array.

```
X = hours(1) + minutes(1:4)

X =

   1.0167 hrs   1.0333 hrs     1.05 hrs   1.0667 hrs
```

Convert each duration in `X` to a number of seconds.

```
S = seconds(X)

S =

        3660         3720         3780         3840
```

`S` is a `double` array.

Find the natural logarithm of `S`. The `log` function accepts inputs of type `double`.

```
Y = log(S)

Y =

    8.2052    8.2215    8.2375    8.2532
```

Input Arguments

X — Input array

numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

See Also

`duration`

Introduced in R2014b

selectmoveresize

Select, move, resize, or copy axes and uicontrol graphics objects

Note: The `selectmoveresize` was removed in R2014b. Use the `plotedit` function to copy, move, resize, and edit objects instead.

Syntax

```
A = selectmoveresize
set(gca,'ButtonDownFcn','selectmoveresize')
```

Description

In releases before R2014b, `selectmoveresize` was useful as the callback routine for axes and uicontrol button down functions. It allowed users to move, resize, and copy axes and uicontrol objects.

`A = selectmoveresize` returns a structure array containing

- `A.Type`: a string containing the action type, which can be `Select`, `Move`, `Resize`, or `Copy`
- `A.Handles`: a list of the selected handles, or, for a `Copy`, an m-by-2 matrix containing the original handles in the first column and the new handles in the second column

`set(gca,'ButtonDownFcn','selectmoveresize')` sets the ButtonDownFcn property of the current axes to `'selectmoveresize'`.

See Also

Axes Properties | `plotedit` | Uicontrol Properties

Introduced before R2006a

matlab.unittest.selectors

Summary of classes in MATLAB Selectors Interface

Description

Use selectors to filter or select elements of a test suite based on their attributes. The `matlab.unittest.selectors` package consists of the following selectors.

matlab.unittest.selectors.HasParameter	Selector for TestSuite elements determined by parameterization
matlab.unittest.selectors.HasSharedTestFixture	Selector for TestSuite elements that use shared test fixture
matlab.unittest.selectors.HasBaseFolder	Selector for TestSuite elements determined by folder
matlab.unittest.selectors.HasName	Selector for TestSuite elements determined by name
matlab.unittest.selectors.HasTag	Selector for TestSuite elements determined by tag

See Also

`matlab.unittest.TestSuite.selectIf`

matlab.unittest.selectors.HasParameter class

Package: matlab.unittest.selectors

Selector for TestSuite elements determined by parameterization

Description

The `HasParameter` selector filters `TestSuite` array elements determined by parameterization.

Construction

`matlab.unittest.selectors.HasParameter` constructs a selector for `TestSuite` elements determined by their parameterization. When you instantiate `HasParameter` without input arguments, the resulting `TestSuite` array only contains elements that have parameterized test methods.

`matlab.unittest.selectors.HasParameter(Name,Value)` constructs a selector with additional options specified by one or more Name,Value pair arguments. The selector filters based on the name of the property that defines a parameter, the name of the parameter, and the value of the parameter. For an element to be selected for the `TestSuite` array, it must have at least one parameter that satisfies all the conditions.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Property'`

Name of the property that defines the parameter specified as a string or as an instance of the `matlab.unittest.constraints.Constraint` class. If the specified property

name is a string, the testing framework creates an `IsEqualTo` constraint with the input string, `Property`, as the expected value.

`'Name'`

Name of the parameter specified as a string or as an instance of the `matlab.unittest.constraints.Constraint` class. If the specified name is a string, the testing framework creates an `IsEqualTo` constraint with the input string, `Name`, as the expected value.

`'Value'`

Value of the parameter specified as any MATLAB data type or as an instance of the `matlab.unittest.constraints.Constraint` class. If the specified property name is not a constraint, the testing framework creates an `IsEqualTo` constraint with the input data, `Value`, as the expected value.

Properties

`PropertyConstraint`

Condition that the test element's parameter property name must satisfy to be included in the test suite, specified as an instance of the `Constraint` in the `Property` input argument.

`NameConstraint`

Condition that the test element's parameter name must satisfy to be included in the test suite, specified as an instance of the `Constraint` in the `Name` input argument.

`ValueConstraint`

Condition that the test element's parameter property value must satisfy to be included in the test suite, specified as an instance of the `Constraint` in the `Value` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Filter Test Suite by Parameterization

In your working folder, create `testZeros.m`. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',3, 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from the file.

```
s = matlab.unittest.TestSuite.fromFile('testZeros.m');
{s.Name}'

ans =

    'testZeros/testClass(type=single,outSize=s2d)'
```

```
    'testZeros/testClass(type=single,outSize=s3d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s3d)'
    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
    'testZeros/testDefaultClass'
    'testZeros/testDefaultSize'
    'testZeros/testDefaultValue'
```

The suite contains 11 test elements: 6 from the parameterized `testClass` method, 2 from the parameterized `testSize` method, and 1 from each of the `testDefaultClass`, `testDefaultSize`, and `testDefaultValue` methods.

Select all of the test elements from parameterized test methods.

```
import matlab.unittest.selectors.HasParameter;

s1 = s.selectIf(HasParameter);
{s1.Name}'

ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s3d)'
    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
```

The suite contains the eight test elements from the two parameterized test methods.

Select all of the test elements from nonparameterized test methods.

```
s2 = s.selectIf(~HasParameter);
{s2.Name}'

ans =

    'testZeros/testDefaultClass'
    'testZeros/testDefaultSize'
    'testZeros/testDefaultValue'
```

Select all test elements that are parameterized and have a property named `'type'` with a parameter name `'double'`.

```
s3 = s.selectIf(HasParameter('Property','type', 'Name','double'));
{s3.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

The resulting suite contains two elements. The `testClass` method is the only method in `testZeros` that uses the `'type'` property. Selecting only `'double'` from the parameters results in two test elements — one for each value of `'outSize'`.

Select all test elements that a have a parameter defined by a property starting with `'t'`.

```
import matlab.unittest.constraints.StartsWithSubstring;

s4 = s.selectIf(HasParameter('Property',StartsWithSubstring('t')));
{s4.Name}'
```

```
ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s3d)'
```

The resulting suite contains the six parameterized test elements from the `testClass` method. The `testSize` method is parameterized, but the elements from the method are not included in the suite because the method does not use a property that starts with `'t'`.

Select all test elements that are parameterized and test the `zeros` function with a `2-D` array. A parameter value representing a `2-D` array has a length of `1` (for example `zeros(3)`) or `2` (for example `zeros(2,3)`).

```
import matlab.unittest.constraints.HasLength;

s5 = s.selectIf(HasParameter('Property','outSize',...
    'Value', HasLength(1)|HasLength(2)));
```

```
{s5.Name}'

ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testSize(outSize=s2d)'
```

Select only the test element that tests that the output is a `double` data type and that it has the correct size for a `2-D` array.

```
s6 = s.selectIf(HasParameter('Property','type','Name','double')...
    & HasParameter('Property','outSize','Name','s2d'))

s6 =

  Test with properties:

                 Name: 'testZeros/testClass(type=double,outSize=s2d)'
      Parameterization: [1x2 matlab.unittest.parameters.TestParameter]
    SharedTestFixtures: []
                 Tags: {0x1 cell}

Tests Include:
   2 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

- “Create Basic Parameterized Test”
- “Create Advanced Parameterized Test”

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.parameters` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2014a

matlab.unittest.selectors.HasSharedTestFixture class

Package: matlab.unittest.selectors

Selector for TestSuite elements that use shared test fixture

Description

The `HasSharedTestFixture` selector filters `TestSuite` array elements based on shared test fixtures.

Construction

`matlab.unittest.selectors.HasSharedTestFixture(f)` constructs a selector for `TestSuite` elements based on their required shared test fixtures. For an element to be selected for the `TestSuite` array, it must use a fixture that is compatible with the specified fixture, `f`.

Input Arguments

`f` — Shared test fixture

`Fixture`

Shared test fixture specified as a `matlab.unittest.fixtures.Fixture` instance. The `TestSuite` array element must use the shared text fixture, `f`, to be selected for the `TestSuite`.

Properties

`ExpectedFixture`

The shared test fixture that a `TestSuite` array element must use to be selected for the `TestSuite`. The `ExpectedFixture` property is specified as a `matlab.unittest.fixtures.Fixture` in the input argument, `f`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Filter Test Suite by Shared Test Fixtures

Create a package folder, `+mytestpackage`, in your current working folder. This package contains two test classes.

In the `+mytestpackage` folder, create `AExampleTest.m`. This class contains two tests that use a suppressed warnings fixture.

```
classdef (SharedTestFixtures={matlab.unittest.fixtures.SuppressedWarningsFixture( ...
        'MATLAB:rmpath:DirNotFound')})...
        AExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testOne(testCase)
            % test code
        end
        function testTwo(testCase)
            % test code
        end
    end
end
```

In the `+mytestpackage` folder, create `BExampleTest.m`. This class contains one test that uses a shared path fixture and a suppressed warnings fixture.

```
classdef (SharedTestFixtures={...
        matlab.unittest.fixtures.PathFixture( ...
        fullfile(matlabroot,'help', 'techdoc', 'matlab_oop', 'examples')),...
        matlab.unittest.fixtures.SuppressedWarningsFixture( ...
        'MATLAB:rmpath:DirNotFound')}) ...
        BExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
    end
```

```
end
```

At the command prompt, define the following fixtures.

```
pf = matlab.unittest.fixtures.PathFixture(...
    fullfile(matlabroot,'help','techdoc','matlab_oop','examples'));
swf = matlab.unittest.fixtures.SuppressedWarningsFixture(...
    'MATLAB:rmpath:DirNotFound');
```

Create a test suite from the package.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasSharedTestFixture;

suite = TestSuite.fromPackage('mytestpackage')
```

```
suite =

  1x3 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 2 Unique Shared Test Fixture Classes, 0 Tags.
```

The test suite has three test elements.

Create a filtered suite that only contains tests that use the path fixture, `pf`.

```
s1 = suite.selectIf(HasSharedTestFixture(pf));
```

The resulting suite, `s1`, contains the test element from `BExampleTest.m`, since the test in that class uses the shared test fixture, `pf`.

Alternatively, pass the selector to the `TestSuite.fromPackage` method instead of generating a full test suite, and then using the `TestSuite.selectIf` method to filter the suite.

```
s1 = TestSuite.fromPackage('mytestpackage', HasSharedTestFixture(pf));
```

Create a filtered test suite that contains tests that use the suppressed warnings fixture, `swf`, but not the path fixture, `pf`.

```
s2 = suite.selectIf(~HasSharedTestFixture(pf) & HasSharedTestFixture(swf));
```

The test suite, `s2`, only contains the two test elements from `AExampleTest.m`. Tests in `BExampleTest.m` are excluded because, in addition to the suppressed warnings fixture, they use the path fixture.

Create a filtered suite that only contains tests that use the path fixture to a different location.

```
pf2 = matlab.unittest.fixtures.PathFixture(fullfile(matlabroot));
s3 =  TestSuite.fromPackage('mytestpackage', HasSharedTestFixture(pf2))
```

```
s3 =

  1x0 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

The test suite does not contain any test elements. The tests in `BExampleTest.m` use a shared path fixture, but the selected path fixture, `pf2`, adds a different folder to the path so its tests are not included in the suite.

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2014a

matlab.unittest.selectors.HasBaseFolder class

Package: matlab.unittest.selectors

Selector for TestSuite elements determined by folder

Description

The `HasBaseFolder` selector filters `TestSuite` array elements determined by the name of the folder that contains the file that defines the test class or function.

Construction

`matlab.unittest.selectors.HasBaseFolder(f)` constructs a selector for `TestSuite` elements determined by the folder, `f`, which contains the file that defines the test class or function. You can specify the base folder as a string or as an instance of the `matlab.unittest.constraints.Constraint` class. If the specified base folder, `f`, is a string instead of a `Constraint`, the testing framework creates an `IsEqualTo` constraint with the input string, `f`, as the expected value.

For a test element to be included in the suite, the file that defines it must be contained in the specified base folder. For test classes defined in packages, the base folder is the parent of the top-level package folder. The base folder never contains any folders that start with `'+'` or `'@'`.

Input Arguments

`f` — Base folder

string | `Constraint`

Base folder specified as a string or a `matlab.unittest.constraints.Constraint` instance. The following conditions must be satisfied for the test element to be selected for the `TestSuite`:

- If `f` is a string, the test element's base folder must exactly match the specified folder.
- If `f` is a constraint, the test element's base folder must satisfy the specified constraint.

Properties

`Constraint`

Condition the base folder must satisfy to be included in the test suite, specified as an instance of the `matlab.unittest.constraints.Constraints` class.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Filter Test Suite by Base Folder

Create a folder, `MyTests`, in your current working folder. In this folder, create two subfolders, `Feature1` and `Feature2`.

In a new file, `Feature1_Test.m`, in the `Feature1` subfolder, create the following test class.

```
classdef Feature1_Test < matlab.unittest.TestCase
    methods (Test)
        function testA1(testCase)
            % test code
        end
        function testB1(testCase)
            % test code
        end
    end
end
```

In a new file, `Feature2_Test.m`, in the `Feature2` subfolder, create the following test class.

```
classdef Feature2_Test < matlab.unittest.TestCase
    methods (Test)
        function testA2(testCase)
```

```
            % test code
        end
        function testB2(testCase)
            % test code
        end
    end
end
```

If necessary, set your current working folder to the folder above `MyTests`. At the command prompt, create a test suite from the `MyTests` folder and examine the contents.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasBaseFolder;
import matlab.unittest.constraints.ContainsSubstring;

suite = TestSuite.fromFolder('MyTests', 'IncludingSubfolders', true);
{suite.Name}

ans =

    'Feature1_Test/testA1'    'Feature1_Test/testB1'    'Feature2_Test/testA2'    'Feat
```

The suite contains the four tests from the two test files.

Select all test suite elements for classes that are defined in the `'Feature1'` folder.

```
 s1 = suite.selectIf(HasBaseFolder(fullfile(pwd,'MyTests','Feature1')));
{s1.Name}

ans =

    'Feature1_Test/testA1'    'Feature1_Test/testB1'
```

The filtered test suite contains only test elements from the `Feature1` folder.

Select all the test suite elements for classes that are defined in folders that do not contain the string `'Feature1'`, and then examine the contents.

```
s1 = suite.selectIf(HasBaseFolder(...
    fullfile(pwd,'MyTests','Feature1')));
{s1.Name}

ans =

    'Feature2_Test/testA2'    'Feature2_Test/testB2'
```

The filtered test suite only contains test elements from the `Feature2` folder.

Alternatively, to generate a filtered suite directly, pass the selector to the `TestSuite.fromFolder` method.

```
s1 = TestSuite.fromFolder('MyTests',...
    ~HasBaseFolder(ContainsSubstring('Feature1')),...
    'IncludingSubfolders',true);
```

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2014a

matlab.unittest.selectors.HasName class

Package: matlab.unittest.selectors

Selector for TestSuite elements determined by name

Description

The `HasName` selector filters `TestSuite` array elements determined by the test element name.

Construction

`matlab.unittest.selectors.HasName(n)` constructs a selector for `TestSuite` elements determined by the test element name, `n`. You can specify the name as a string or as an instance of the `matlab.unittest.constraints.Constraint` class. If the specified name, `n`, is a string, the testing framework creates an `IsEqualTo` constraint with the input string, `n`, as the expected value.

For a test element to be included in the suite, the test element must have the same name as the specified name.

Input Arguments

`n` — Test element name

string | `Constraint`

Test element name specified as a string or `matlab.unittest.constraints.Constraint` instance. The following conditions must be satisfied for the test element to be selected for the `TestSuite`:

- If `n` is a string, the test element's name must exactly match the specified name.
- If `n` is a constraint, the test element's name must satisfy the specified constraint.

Properties

`Constraint`

Condition the test element name must satisfy to be included in the test suite, specified as an instance of the `matlab.unittest.constraints.Constraints` class.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Filter Test Suite by Name

Create the following test class in a file, `ExampleTest.m`, in your current working folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
        function testOne(testCase)
            % test code
        end
         function testTwo(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest.m` file and examine the contents.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasName;
import matlab.unittest.constraints.EndsWithSubstring;

suite = TestSuite.fromFile('ExampleTest.m');
```

```
{suite.Name}

ans =

    'ExampleTest/testPathAdd'    'ExampleTest/testOne'    'ExampleTest/testTwo'
```

The suite contains three tests.

Select all the test suite elements that have the name `'ExampleTest/testPathAdd'`, and examine the contents.

```
s1 = suite.selectIf(HasName('ExampleTest/testPathAdd'));
{s1.Name}

ans =

    'ExampleTest/testPathAdd'
```

The filtered test suite only contains one test element.

Select all the test suite elements that end in either `'One'` or `'Two'`, and examine the contents.

```
s1 =  suite.selectIf(HasName(EndsWithSubstring('One')) | ...
    HasName(EndsWithSubstring('Two')));
{s1.Name}

ans =

    'ExampleTest/testOne'    'ExampleTest/testTwo'
```

At the time of the test suite construction, create a test suite that only contains tests with the substring `'One'`.

```
import matlab.unittest.constraints.ContainsSubstring;
s2 = TestSuite.fromFile('ExampleTest.m',...
    HasName(ContainsSubstring('One')))

s2 =

  Test with properties:

                Name: 'ExampleTest/testOne'
      Parameterization: []
    SharedTestFixtures: []
```

```
                Tags: {0x1 cell}

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2014a

matlab.unittest.selectors.HasTag class

Package: matlab.unittest.selectors

Selector for TestSuite elements determined by tag

Description

The `HasTag` selector filters `TestSuite` array elements determined by the test element tag.

Construction

`matlab.unittest.selectors.HasTag` constructs a selector for `TestSuite` elements determined by the test element tag. When you instantiate `HasTag` without input arguments, the resulting `TestSuite` array contains only elements with one or more tags.

`matlab.unittest.selectors.HasTag(t)` constructs a selector for `TestSuite` elements determined by the test element tag, `t`. You can specify the name as a string or as an instance of the `matlab.unittest.constraints.Constraint` class. If the specified name, `t`, is a string, the testing framework creates an `IsEqualTo` constraint with the input string, `t`, as the expected value.

For a test element to be included in the suite, the test element must be tagged with the specified string or with a value that satisfies the specified constraint.

Input Arguments

`t` — Test element tag

string | `Constraint`

Test element tag, specified as a string or `matlab.unittest.constraints.Constraint` instance. If a test element tag meets the following conditions, the `TestSuite` contains the test:

- If `t` is a string, the test element tag is the specified value.

- If `t` is a constraint, the test element tag is a value that satisfies the specified constraint.

Properties

`Constraint` — Condition test element tag must satisfy

instance of `matlab.unittest.constraints.Constraints` class

Condition the test element tag must satisfy to be included in the test suite, specified as an instance of the `matlab.unittest.constraints.Constraints` class.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Filter Test Suite by Tag

Create the following test class in a file, `ExampleTest.m`, in your current working folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit'})
        function testB (testCase)
            % test code
        end
        function testC (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit','FeatureA'})
        function testD (testCase)
```

```
            % test code
        end
    end
    methods (Test, TestTags = {'System','FeatureA'})
        function testE (testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class and examine the contents.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasTag

suite = TestSuite.fromClass(?ExampleTest)
```

```
suite =

  1x5 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 3 Unique Tags.
```

Click the hyperlink for `3 Unique Tags` to display all the tags in the suite.

```
       Tag
    __________

    'FeatureA'
    'System'
    'Unit'
```

Select all the test suite elements that have the tag `'Unit'`.

```
s1 = suite.selectIf(HasTag('Unit'))
```

```
s1 =
```

```
  1x3 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 2 Unique Tags.
```

Select all the test suite elements that do not contain the tag `'FeatureA'`.

```
s2 =  suite.selectIf(~HasTag('FeatureA'));
{s2.Name}

ans =

    'ExampleTest/testB'    'ExampleTest/testC'    'ExampleTest/testA'
```

Select all the test suite elements that have no tags.

```
s3 =  suite.selectIf(~HasTag)

s3 =

  Test with properties:

                 Name: 'ExampleTest/testA'
      Parameterization: []
    SharedTestFixtures: []
                 Tags: {0x1 cell}

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

- “Tag Unit Tests”

Alternatives

Use the `HasTag` selector for maximum flexibility to create test suites from tags. Alternatively, at the time of test suite construction, you can filter the test suite Using the `'Tag'` name-value pair. For example,

```
s = TestSuite.fromClass(?ExampleTest,'Tag','Unit');
```

You can also select and run tagged tests using the `'Tag'` name-value pair with the `runtests` function. For example,

```
runtests('ExampleTest.m','Tag','Unit')
```

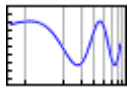
See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2015a

semilogx

Semilogarithmic plot



Syntax

```
semilogx(Y)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)
```

Description

`semilogx` plot data as logarithmic scales for the *x*-axis.

`semilogx(Y)` creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of `Y` versus their index if `Y` contains real numbers. `semilogx(Y)` is equivalent to `semilogx(real(Y),imag(Y))` if `Y` contains complex numbers. `semilogx` ignores the imaginary component in all other uses of this function.

`semilogx(X1,Y1,...)` plots all `Yn` versus `Xn` pairs. If only one of `Xn` or `Yn` is a matrix, `semilogx` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

`semilogx(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'PropertyName',PropertyValue,...)` sets property values for all charting lines created by `semilogx`. For a list of properties, see Chart Line Properties.

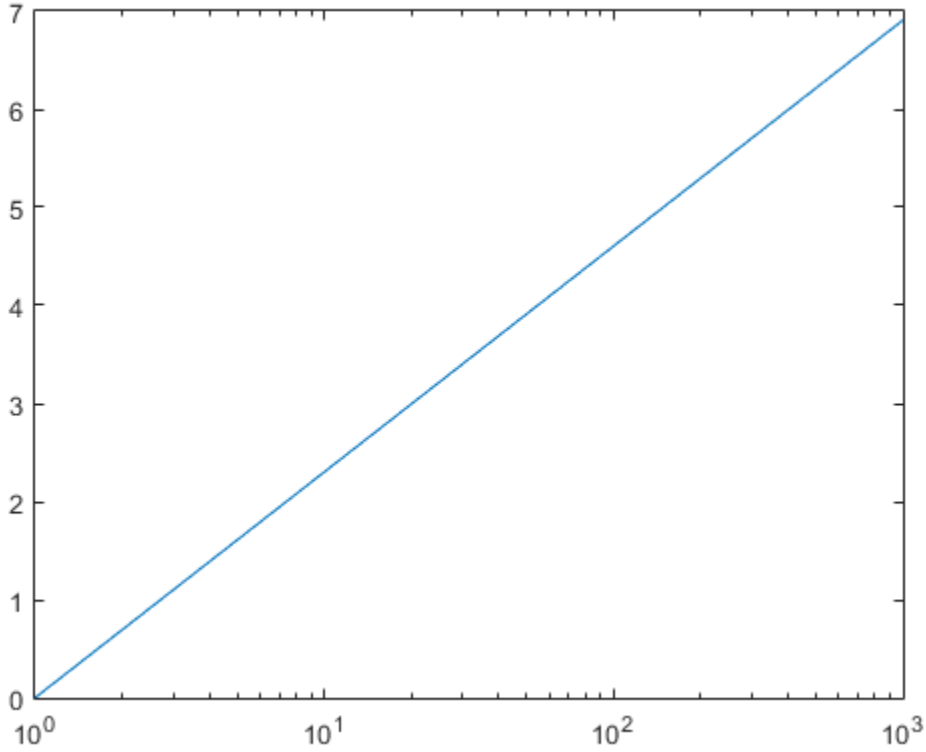
`h = semilogx(...)` return a vector of chart line handles, one handle per line.

Examples

Logarithmic Scale for x-Axis

Create a plot with a logarithmic scale for the x-axis and a linear scale for the y-axis.

```
x = 0:1000;
y = log(x);

figure
semilogx(x,y)
```



More About

Tips

If you do not specify a color when plotting more than one line, `semilogx` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,`*`LineSpec`* triples; for example,

```
semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

See Also

Functions

`LineSpec` | `loglog` | `plot` | `semilogy`

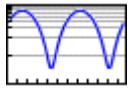
Properties

Chart Line Properties

Introduced before R2006a

semilogy

Semilogarithmic plot



Syntax

```
semilogy(Y)
semilogy(X1,Y1,...)
semilogy(X1,Y1,LineSpec,...)
semilogy(...,'PropertyName',PropertyValue,...)
h = semilogy(...)
```

Description

`semilogy` plots data with logarithmic scale for the *y*-axis.

`semilogy(Y)` creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis. It plots the columns of `Y` versus their index if `Y` contains real numbers. `semilogy(Y)` is equivalent to `semilogy(real(Y),imag(Y))` if `Y` contains complex numbers. `semilogy` ignores the imaginary component in all other uses of this function.

`semilogy(X1,Y1,...)` plots all `Yn` versus `Xn` pairs. If only one of `Xn` or `Yn` is a matrix, `semilogy` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

`semilogy(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogy(...,'PropertyName',PropertyValue,...)` sets property values for all the charting lines created by `semilogy`. For a list of properties, see Chart Line Properties.

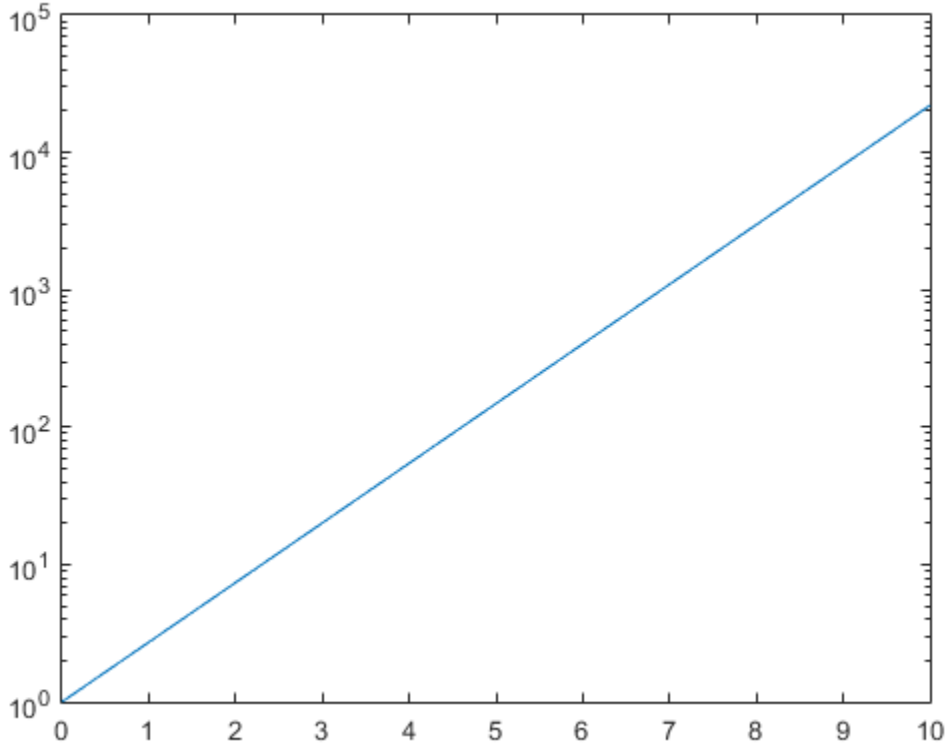
`h = semilogy(...)` returns a vector of chart line handles, one handle per line.

Examples

Logarithmic Scale for y-Axis

Create a plot with a logarithmic scale for the y-axis and a linear scale for the x-axis.

```
x = 0:0.1:10;
y = exp(x);

figure
semilogy(x,y)
```



More About

Tips

If you do not specify a color when plotting more than one line, `semilogy` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,`*`LineSpec`* triples; for example,

```
semilogy(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

See Also

Functions
`LineSpec` | `loglog` | `plot` | `semilogx`

Properties
Chart Line Properties

Introduced before R2006a

sendmail

Send email message to address list

Syntax

```
sendmail(recipients,subject)
sendmail(recipients,subject,message)
sendmail(recipients,subject,message,attachments)
```

Description

`sendmail(recipients,subject)` sends email to `recipients` with the specified `subject`. The `recipients` input is a string for a single address, or a cell array of strings for multiple addresses.

`sendmail(recipients,subject,message)` includes the specified `message`. If `message` is a string, `sendmail` automatically wraps text at 75 characters. To force a line break in the message text, use `10`, as shown in the Examples. If `message` is a cell array of strings, each cell represents a new line of text.

`sendmail(recipients,subject,message,attachments)` attaches the files listed in the string or cell array `attachments`.

Examples

Send a message with two attachments to a hypothetical email address:

```
sendmail('user@otherdomain.com',...
         'Test subject','Test message',...
         {'folder/attach1.html','attach2.doc'});
```

Send a message with forced line breaks (using `10`) to a hypothetical email address:

```
sendmail('user@otherdomain.com','New subject', ...
        ['Line1 of message' 10 'Line2 of message' 10 ...
         'Line3 of message' 10 'Line4 of message']);
```

The resulting message is:

```
Line1 of message
Line2 of message
Line3 of message
Line4 of message
```

Alternatives

On Windows systems with Microsoft Outlook®, you can send email directly through Outlook by accessing the COM server with `actxserver`. For an example, see Solution 1-RTY6J.

More About

Tips

- The `sendmail` function does not support HTML-formatted messages. However, you can send HTML files as attachments.
- If `sendmail` cannot determine your email address or outgoing SMTP mail server from your system registry, specify those settings using the `setpref` function. For example:

  ```
  setpref('Internet','SMTP_Server','my_server.example.com');
  setpref('Internet','E_mail','my_email@example.com');
  ```

  To identify the SMTP server for the call to `setpref`, check the preferences for your electronic mail application, or consult your email system administrator. If you cannot easily determine the server name, try `'mail'`, which is a common default, such as:

  ```
  setpref('Internet','SMTP_Server','mail');
  ```

- By default, the `sendmail` function does not support email servers that require authentication. To support these servers, change your system settings and set preferences for the SMTP user name and password, with commands in the following form:

  ```
  props = java.lang.System.getProperties;
  props.setProperty('mail.smtp.auth','true');
  ```

```
setpref('Internet','SMTP_Username','myaddress@example.com');
setpref('Internet','SMTP_Password','mypassword');
```

- To override the default character encoding, set the preference for email character encoding as follows:

  ```
  setpref('Internet','E_mail_Charset',encoding);
  ```
  where *`encoding`* is a string specifying the character encoding, such as `'SJIS'`.

- "Specify Proxy Server Settings for Connecting to the Internet"

See Also

`getpref` | `setpref`

serial

Create serial port object

Syntax

```
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)
```

Description

`obj = serial('port')` creates a serial port object associated with the serial port specified by *`port`*. If *`port`* does not exist, or if it is in use, you will not be able to connect the serial port object to the device.

`Port` object name will depend upon the platform that the serial port is on. `instrhwinfo` ('serial') provides a list of available serial ports. This list is an example of serial constructors on different platforms:

Platform	Serial Port Constructor
Linux and Linux 64	`serial('/dev/ttyS0');`
Mac OS X 64	`serial('/dev/tty.KeySerial1');`
Windows 32 and Windows 64	`serial('com1');`

`obj = serial('port','PropertyName',PropertyValue,...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

Examples

This example creates the serial port object `s1` associated with the serial port COM1 on a Windows platform.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
get(s1,{'Type','Name','Port'})
ans =
    'serial'    'Serial-COM1'    'COM1'
```

To specify properties during object creation

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

More About

Tips

When you create a serial port object, these property values are automatically configured:

- The `Type` property is given by `serial`.
- The `Name` property is given by concatenating `Serial` with the port specified in the `serial` function.
- The `Port` property is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid on a Windows platform.

```
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Refer to Configuring Property Values for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

See Also

`fclose` | `fopen` | `Name` | `Port` | `Status` | `Type`

Introduced before R2006a

serialbreak

Send break to device connected to serial port

Syntax

```
serialbreak(obj)
serialbreak(obj,time)
```

Description

`serialbreak(obj)` sends a break of 10 milliseconds to the device connected to the serial port object, `obj`.

`serialbreak(obj,time)` sends a break to the device with a duration, in milliseconds, specified by `time`. Note that the duration of the break might be inaccurate under some operating systems.

More About

Tips

For some devices, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the device.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

See Also

`fopen` | `stopasync` | `Status`

Introduced before R2006a

set

Set graphics object properties

Syntax

```
set(H,Name,Value)
set(H,NameArray,ValueArray)
set(H,S)
s = set(H)
values = set(H,Name)
```

Description

Note: Do not use the `set` function on Java objects as it will cause a memory leak. For more information, see "Accessing Private and Public Data"

`set(H,Name,Value)` specifies a value for the property `Name` on the object identified by `H`. Use single quotes around the property name, for example, `set(H,'Color','red')`. If `H` is a vector of objects, then `set` sets the property for all the objects. If `H` is empty (that is, `[]`), `set` does nothing, but does not return an error or warning.

`set(H,NameArray,ValueArray)` specifies multiple property values using the cell arrays `NameArray` and `ValueArray`. To set `n` property values on each of `m` graphics objects, specify `ValueArray` as an `m`-by-`n` cell array, where `m = length(H)` and `n` is equal to the number of property names contained in `NameArray`.

`set(H,S)` specifies multiple property values using `S`, where `S` is a structure whose field names are the object property names and whose field values are the corresponding property values. MATLAB ignores empty structures.

`s = set(H)` returns the user-settable properties and possible values for the object identified by `H`. `s` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output argument, the MATLAB software displays the information on the screen. `H` must be a single object.

`values = set(H,Name)` returns the possible values for the specified property. If the possible values are strings, `set` returns each in a cell of the cell array `values`. For other properties, `set` returns a statement indicating that `Name` does not have a fixed set of property values. If you do not specify an output argument, MATLAB displays the information on the screen. `H` must be a single object.

Note: For more information about properties you can set, see the property pages for each object, for example, Figure Properties, Axes Properties, Chart Line Properties, Text Properties, and so on.

Examples

Change Color of Specific Line

Plot a line and return the chart line object as `p`. Set the `Color` property of the line to `'red'`.

```
p = plot(1:10);
set(p,'Color','red')
```

Change Color for Multiple Lines

Create a plot with four lines using random data and return the four chart line objects as `P`. Set the `Color` property for all of the lines to `'red'`.

```
P = plot(rand(4));
set(P,'Color','red')
```

Set Line Style to Different Value for Multiple Lines

Set the value of the `LineStyle` property for four chart line objects each to a different value. Transpose the value of the cell array so that it has the proper shape.

```
P = plot(rand(4));
NameArray = {'LineStyle'};
ValueArray = {'-','--',':','-.'}';
set(P,NameArray,ValueArray)
```

Set Different Values for Multiple Properties on Multiple Objects

Set the values of the `Marker` and `Tag` properties on three different stem series objects to different values. Each row of the value cell array corresponds to an object in `h` and contains two values, one for the `Marker` property and one for the `Tag` property.

```
x = 0:30;
y = [1.5*cos(x); 4*exp(-.1*x).*cos(x); exp(.05*x).*cos(x)]';
S = stem(x,y);
NameArray = {'Marker','Tag'};
ValueArray = {'o','Decaying Exponential';...
   'square','Growing Exponential';...
   '*','Steady State'};
set(S,NameArray,ValueArray)
```

More About

Tips

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`.

Setting Property Units

Note that if you are setting both the `FontSize` and the `FontUnits` properties in one function call, you must set the `FontUnits` property first so that the MATLAB software can correctly interpret the specified `FontSize`. The same applies to figure and axes units — always set the `Units` property before setting properties whose values you want to be interpreted in those units. For example,

```
f = figure('Units','characters','Position',[30 30 120 35]);
```

- “Default Property Values”
- “Graphics Object Properties”

See Also

`findobj` | `gca` | `gcf` | `gco` | `gcbo` | `get`

Introduced before R2006a

set

Set property values for `audioplayer` object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object `obj`.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array `cellOfNames` to the corresponding values in the cell array `cellOfValues`. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array `structOfProperties` to the values of the associated fields.

`settableProperties = set(obj)` returns the names of the properties that you can set in a structure array. The field names of `settableProperties` are the property names.

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Examples

View the list of properties that you can set for an `audioplayer` object:

```
load handel.mat;
handelObj = audioplayer(y, Fs);
set(handelObj)
```

Set the `Tag` and `UserData` properties of an `audioplayer` object using a structure array:

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data', pi, [1 2 3 4]};

load handel.mat;
handelObj = audioplayer(y, Fs);
set(handelObj, newValues)

% View the values all properties.
get(handelObj)
```

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for an object called `handelObj` (as created in the Examples):

```
handelObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(handelObj, 'Tag', 'This is my tag.');
```

See Also

`audioplayer` | `get`

set

Set property values for `audiorecorder` object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object `obj`.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array `cellOfNames` to the corresponding values in the cell array `cellOfValues`. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array `structOfProperties` to the values of the associated fields.

`settableProperties = set(obj)` returns the names of the properties that you can set in a structure array. The field names of `settableProperties` are the property names.

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Examples

View the list of properties that you can set for an `audiorecorder` object:

```
recorderObj = audiorecorder;
set(recorderObj)
```

Set the `Tag` and `UserData` properties of an `audiorecorder` object using a structure array:

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data', pi, [1 2 3 4]};

recorderObj = audiorecorder;
set(recorderObj, newValues)

% View the values all properties.
get(recorderObj)
```

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for an object called `recorderObj` (as created in the Examples):

```
recorderObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(recorderObj, 'Tag', 'This is my tag.');
```

See Also

`audiorecorder` | `get`

set (COM)

Set object or interface property to specified value

Syntax

```
set(h,'pname',value)
set(h,'pname1',value1,'pname2',value2,...)
```

Description

`set(h,'pname',value)` sets the property specified in the string `pname` to the given `value`.

`set(h,'pname1',value1,'pname2',value2,...)` sets each property specified in the `pname` strings to the given `value`.

For information on how MATLAB converts workspace matrices to COM data types, see "Handling COM Data in MATLAB Software".

COM functions are available on Microsoft Windows systems only.

Examples

Create an `mwsamp` control and use `set` to change the `Label` and `Radius` properties:

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1',[0 0 200 200],f);

set(h,'Label','Click to fire event','Radius',40);
invoke(h,'Redraw');
```

Here is another way to do the same thing, only without `set` and `invoke`:

```
h.Label = 'Click to fire event';
h.Radius = 40;
Redraw(h);
```

See Also

`get (COM)` | `inspect` | `isprop` | `addproperty` | `deleteproperty`

Introduced before R2006a

matlab.mixin.SetGet class

Package: matlab.mixin

Abstract class used to derive handle classes with set and get methods

---

Note: `hgsetget` will be removed in a future release. Use `matlab.mixin.SetGet` instead.

Description

`classdef` *`MySubClass`* `<` `matlab.mixin.SetGet` makes *`MySubClass`* a subclass of the `matlab.mixin.SetGet` class, which is itself a subclass of the `handle` class

Use the `matlab.mixin.SetGet` class to derive classes that inherit `set` and `get` methods that behave like MATLAB graphics `set` and `get` functions.

Methods

The `matlab.mixin.SetGet` class defines these method:

Method	Purpose
`set`	Assigns values to the specified properties or returns a cell array of possible values for writable properties.
`get`	Returns value of specified property or a `struct` with all property values.
`setdisp`	Implicitly called when `set` is called with no output arguments and a handle array, but no property name. Override this method to change what set displays.
`getdisp`	Implicitly called when `get` is called with no output arguments and handle array, but no property name. Override this method to change what get displays.

Attributes

`Abstract`	`true`
`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

To learn about attributes of classes, see Class Attributes in the MATLAB Object-Oriented Programming documentation.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Handle Objects” in the MATLAB documentation.

More About

- Class Attributes

set

Class: matlab.mixin.SetGet
Package: matlab.mixin

Assign specified property name/property value pairs

Syntax

```
set(H,'Name',Value,...)
set(H,pn,pv)
set(H,S)
S = set(h)
```

Description

`set(H,'Name',Value,...)` sets the named property to the specified value for the objects in the handle array `H`.

`set(H,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv` for all objects specified in `H`. The cell array `pn` must be `1`-by-`n` (where `n` is the number of property names), but the cell array `pv` can be `m`-by-`n` where `m` is equal to `length(H)`. `set` updates each object with the associated set of values for the list of property names contained in.

`set(H,S)` sets the properties identified by each field name of `struct` `S` with the values contained in `S`. `S` is a `struct` whose field names are object property names.

`S = set(h)` returns the user-settable properties of scalar `h`. `S` is a `struct` whose field names are the object's property names and values that are either empty cell arrays or cell arrays of possible values for properties that have a finite set of predefined possible values.

Tips

- Override the `matlab.mixin.SetGet` class `setdisp` method to change how MATLAB displays information returned by `set`.

Input Arguments

`H` — Input handle array
handle array

Input handle array, specified as a single handle or an array of handles

`'Name'` — Property name
string

Property name, specified as a quoted string

`Value` — Property value to assign to the named property
property value

Property value to assign to the named property, specified as appropriate for that property.

`pn` — Property names
cell array of strings

Property names, specified as a cell array of strings. The cell array `pn` must be `1`-by-`n` (where `n` is the number of property names).

Data Types: `cell`

`pv` — Property values
cell array

Property values, specified as a cell array. The cell array `pv` can be `m`-by-`n` where `m` is equal to `length(H)` and `n` is the number of property names in `pn`.

Data Types: `cell`

`S` — Property name and value structure
`struct`

Property name and value structure. The fields of `S` correspond to property names and the values of the fields are the property values to set.

Data Types: `struct`

`h` — Scalar object handle
handle

Scalar object handle. To obtain user-settable properties, input argument must be a scalar object handle.

Output Arguments

S — Settable properties

`struct`

Settable properties, returned as a structure with fields corresponding to property names and values that are either empty cell arrays or cell arrays of possible values for properties that have a finite set of predefined possible values.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.SetGet.setdisp` | `matlab.mixin.SetGet` | `matlab.mixin.SetGet.get` | `set`

More About

- "Implementing a Set/Get Interface for Properties"

setdisp

Class: matlab.mixin.SetGet
Package: matlab.mixin

Customize set method display

Syntax

```
setdisp(h)
```

Description

`setdisp(h)` called by `set` when `set` is called with no output arguments and a single input argument that is a scalar handle. Override this `matlab.mixin.SetGet` class method in a subclass to change how property information is displayed in the command window.

Input Arguments

h — Object handle

handle

Object handle whose settable properties and their possible values are to be displayed.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.SetGet.set` | `matlab.mixin.SetGet` | `matlab.mixin.SetGet.getdisp` | `set`

More About

- "Implementing a Set/Get Interface for Properties"

get

Class: matlab.mixin.SetGet
Package: matlab.mixin

Query specified property values

Syntax

```
CV = get(H,'Name')
SV = get(h)
get(h)
```

Description

`CV = get(H,'Name')` returns the value of the named property from the objects in the handle array `H`. If `H` is scalar, `get` returns a single value; if `H` is an array, `get` returns a cell array of property values.

If you specify a cell array of property names as the second argument, then `get` returns a cell array of values, where each row in the cell corresponds to an element in `H` and each column in the cell corresponds to an element in the property name cell array.

If `H` is nonscalar and `'Name'` is the name of a dynamic property, `get` returns a value only if the property exists in all objects referenced in `H`.

`SV = get(h)` returns a structure array in which the field names are the object's property names and the values are the current values of the corresponding properties. If `h` is nonscalar, then `SV` is a `numel(h)`-by-`1` array of structures.

`get(h)` displays all property names and their current values for the MATLAB objects with handle `h`.

Your subclass can override the `matlab.mixin.SetGet` `getdisp` method to control how MATLAB displays this information.

Tips

- Override the `matlab.mixin.SetGet` class `matlab.mixin.SetGet.getdisp` method to change how MATLAB displays information returned by `get`.

Input Arguments

`H` — Input handle array

object array

Input handle array, specified as a single handle or an array of handles

`'Name'` — Property to query

string

Property to query, specified as a case-sensitive string.

Output Arguments

`CV` — Value of queried property

property value

Value of queried property, returned as a single value or a cell array of values.

`SV` — Structure array of property names and values

`struct`

Structure array of property names and values, in which the field names are the object's property names and the values are the current values of the corresponding properties.

Data Types: `struct`

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.SetGet.getdisp` | `matlab.mixin.SetGet` | `get`

More About

- "Implementing a Set/Get Interface for Properties"

getdisp

Class: matlab.mixin.SetGet
Package: matlab.mixin

Customize get method display

Syntax

```
getdisp(h)
```

Description

`getdisp(h)` called by `get` when `get` is called with no output arguments and a single input argument that is a scalar handle. Override this `matlab.mixin.SetGet` class method in a subclass to change how property information is displayed in the command window.

Input Arguments

h — Object handle

object

Object handle whose gettable properties and their current values are to be displayed.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.SetGet.getdisp` | `matlab.mixin.SetGet.getdisp` | `get`

More About

- "Implementing a Set/Get Interface for Properties"

set

Class: VideoReader

Set property values for video reader object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object `obj`.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array `cellOfNames` to the corresponding values in the cell array `cellOfValues`. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array `structOfProperties` to the values of the associated fields.

`settableProperties = set(obj)` returns the names of the properties that you can set in a structure array. The field names of `settableProperties` are the property names.

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Examples

View Settable Properties

View the list of properties that you can set for a `VideoReader` object.

```
xyloObj = VideoReader('xylophone.mp4');
set(xyloObj)
```

```
            Tag: {}
       UserData: {}
    CurrentTime: {}
```

Set Object Properties

Set the `Tag` and `UserData` properties of a `VideoReader` object using a structure array.

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data',pi,[1 2 3 4]};

xyloObj = VideoReader('xylophone.mp4');
set(xyloObj,newValues)
```

View the values all properties.

```
get(xyloObj)
```

```
obj =

  VideoReader with properties:

   General Properties:
           Name: 'xylophone.mp4'
           Path: 'matlabroot\toolbox\matlab\audiovideo'
       Duration: 4.7000
    CurrentTime: 0
            Tag: 'My Tag'
       UserData: {'My User Data'  [3.1416]  [1 2 3 4]}

   Video Properties:
          Width: 320
         Height: 240
      FrameRate: 30
    BitsPerPixel: 24
```

```
    VideoFormat: 'RGB24'
```

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for a reader object called `xyloObj` (as created in the Examples):

```
xyloObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(xyloObj,'Tag','This is my tag.');
```

See Also

`VideoReader` | `get`

set (RandStream)

Set random number stream property

Class

`RandStream`

Syntax

```
set(stream,'PropertyName',Value)
set(stream,'Property1',Value1,'Property2',Value2,...)
set(stream,A)
A = set(stream,'Property')
set(stream,'Property')
A = set(stream)
set(stream)
```

Description

`set(stream,'PropertyName',Value)` sets the property `'PropertyName'` of the random stream `stream` to the value `Value`.

`set(stream,'Property1',Value1,'Property2',Value2,...)` sets multiple random stream property values with a single statement.

`set(stream,A)` where `A` is a structure whose field names are property names of the random stream `stream` sets the properties of `stream` named by each field with the values contained in those fields.

`A = set(stream,'Property')` or `set(stream,'Property')` displays possible values for the specified property of `stream`.

`A = set(stream)` or `set(stream)` displays or returns all writable properties of `stream` and their possible values.

See Also

`RandStream` | `get (RandStream)` | `rand` | `randn` | `randi`

set (serial)

Configure or display serial port object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
```

Description

`set(obj)` displays all configurable properties values for the serial port object, `obj`. If a property has a finite list of possible string values, then these values are also displayed.

`props = set(obj)` returns all configurable properties and their possible values for `obj` to `props`. `props` is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

`set(obj,'PropertyName')` displays the valid values for *`PropertyName`* if it possesses a finite list of string values.

`props = set(obj,'PropertyName')` returns the valid values for *`PropertyName`* to `props`. If *`PropertyName`* does not have a finite list of possible values, `props` is a cell array of possible string values or an empty cell array.

`set(obj,'PropertyName',PropertyValue,...)` configures multiple property values with a single command.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be m-by-n where m is equal to the number of serial port objects in `obj` and n is equal to the length of `PN`.

`set(obj,S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are serial port object properties, and whose field values are the values of the corresponding properties.

Examples

This example illustrates shows how to use the `set` function to configure or return property values, on a Windows platform.

```
s = serial('COM1');
set(s,'BaudRate',9600,'Parity','even')
set(s,{'StopBits','RecordName'},{2,'sydney.txt'})
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

More About

Tips

Refer to Configuring Property Values for a list of serial port object properties that you can configure with `set`.

You can use any combination of property name/property value pairs, structures, and cell arrays in one call to the `set` function. Additionally, you can specify a property name without regard to case, and you can use property name completion. For example, if `s` is a serial port object, then the following commands are all valid.

```
set(s,'BaudRate')
set(s,'baudrate')
set(s,'BAUD')
```

See Also

`get`

Introduced before R2006a

set (tscollection)

Set properties of `tscollection` object

Syntax

```
set(tsc,'Property',Value)
set(tsc,'Property1',Value1,'Property2',Value2,...)
set(tsc,'Property')
```

Description

`set(tsc,'Property',Value)` sets the property `'Property'` of the `tscollection` `tsc` to the value `Value`. The following syntax is equivalent:

```
tsc.Property = Value
```

`set(tsc,'Property1',Value1,'Property2',Value2,...)` sets multiple property values for `tsc` with a single statement.

`set(tsc,'Property')` displays values for the specified property in the time-series collection `tsc`.

`set(tsc)` displays all properties and values of the `tscollection` object `tsc`.

See Also

`get (tscollection)`

Introduced before R2006a

setabstime (tscollection)

Set times of `tscollection` object as date strings

Syntax

```
tsc = setabstime(tsc,Times)
tsc = setabstime(tsc,Times,format)
```

Description

`tsc = setabstime(tsc,Times)` sets the times in tsc using the date strings Times. Times must be either a cell array of strings, or a `char` array containing valid date or time values in the same date format.

`tsc = setabstime(tsc,Times,format)` specifies the date-string format used in Times explicitly.

Examples

1. Create a `tscollection` object.

   ```
   tsc = tscollection(timeseries(rand(3,1)))
   ```

2. Set the absolute time vector.

   ```
   tsc = setabstime(tsc,{'12-DEC-2005 12:34:56',...
         '12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})
   ```

See Also

`datestr` | `getabstime (tscollection)` | `tscollection`

Introduced before R2006a

setappdata

Store application-defined data

Use the `setappdata` function to store data in a UI. You can retrieve the data elsewhere in your code using the `getappdata` function. Both of these functions provide a convenient way to share data between callbacks or between separate UIs.

Syntax

```
setappdata(obj,name,val)
```

Description

`setappdata(obj,name,val)` stores the contents of `val`. The graphics object, obj, and the name identifier, name, uniquely identify the data for later retrieval.

Examples

Store and Retrieve Date Information

Create a figure window. Then, get the current time using the `date` function.

```
f = figure;
val = date
```

```
val =

23-Dec-2014
```

Store the contents of `val` using the `setappdata` function. In this case, `val` is stored in the figure object using the name identifier, `'todaysdate'`.

```
setappdata(f,'todaysdate',val);
```

Retrieve the data and display it.

```
getappdata(f,'todaysdate')
```

```
ans =

23-Dec-2014
```

Input Arguments

`obj` — Graphics object in which to store the value

figure | uipanel | uibuttongroup | uicontrol | ...

Graphics object in which to store the value, specified as any graphics object (except an ActiveX component). The graphics object must be accessible from within the functions you plan to store and retrieve the data.

`name` — Name identifier

`string`

Name identifier, specified as a `string` value. Select a unique name identifier that is easy to remember so that you can easily recall it when you want to retrieve the data.

Data Types: `char`

`val` — Value to store

any MATLAB data type

Value to store, specified as any MATLAB data type.

More About

- "Share Data Among Callbacks"

See Also

`getappdata` | `guidata` | `isappdata` | `rmappdata`

Introduced before R2006a

setcats

Set categories in categorical array

Syntax

```
B = setcats(A,newcats)
```

Description

`B = setcats(A,newcats)` sets categories in output categorical array `B` using categories defined by `newcats` and elements defined by `A`. If an element of `A` is in a category listed in `newcats`, then the corresponding element of `B` has the same value as in `A`.

- If `A` has a category not listed in `newcats`, then `B` does not have that category. The corresponding elements in `B` are undefined.
- If `newcats` lists a category that is not a category of `A`, then `B` has no elements equal to that category.

Examples

Set New Categories

Create a categorical array containing various colors.

```
A = categorical({'blue','black','red';'red','blue','black';'black','red','blue'})
```

```
A =

     blue       black      red
     red        blue       black
     black      red        blue
```

Set new categories.

```
B = setcats(A,{'red','black'})
```

```
B =

     <undefined>      black            red
     red              <undefined>      black
     black            red              <undefined>
```

Elements that were `blue` in `A` are undefined in `B`.

Set New Categories and Assign Elements

Create a categorical array containing various colors.

```
A = categorical({'blue','black','red';'red','blue','black';'black','red','blue'})
```

```
A =

     blue       black      red
     red        blue       black
     black      red        blue
```

Set new categories. Include a category that is not a category of `A`.

```
B = setcats(A,{'red','pink','blue'})
```

```
B =

     blue             <undefined>      red
     red              blue             <undefined>
     <undefined>      red              blue
```

No element of `B` is `pink`, because `pink` is not a category of `A`. Assign an element of `B` to be `pink`.

```
B(1,2) = 'pink';
B
```

```
B =

     blue              pink      red
     red               blue      <undefined>
     <undefined>       red       blue
```

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

newcats — New categories

string | cell array of strings

New categories, specified as a string or a cell array of strings.

More About

Tips

- To change category names in a categorical array, use `renamecats`.

See Also

`addcats` | `categories` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `reordercats`

setdiff

Set difference of two arrays

Syntax

```
C = setdiff(A,B)
C = setdiff(A,B,'rows')
[C,ia] = setdiff(A,B)
[C,ia] = setdiff(A,B,'rows')

[C,ia] = setdiff( ___ ,setOrder)
[C,ia] = setdiff(A,B,'legacy')
[C,ia] = setdiff(A,B,'rows','legacy')
```

Description

`C = setdiff(A,B)` returns the data in `A` that is not in `B`.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `setdiff` returns the values in `A` that are not in `B`. The values of `C` are in sorted order.
- If `A` and `B` are tables, then `setdiff` returns the rows from `A` that are not in `B`, with repetitions removed. The rows of table `C` are in sorted order.

`C = setdiff(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the rows from `A` that are not in `B`. The rows of `C` are in sorted order.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia] = setdiff(A,B)` also returns the index vector `ia`.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `C = A(ia)`.
- If `A` and `B` are tables, then `C = A(ia,:)`.

`[C,ia] = setdiff(A,B,'rows')` also returns the index vector `ia`, such that `C = A(ia,:)`.

`[C,ia] = setdiff( ___ ,setOrder)` returns `C` in a specific order using any of the input arguments in the previous syntaxes. `setOrder='sorted'` returns the values (or rows) of `C` in sorted order. `setOrder='stable'` returns the values (or rows) of `C` in the same order as `A`. If no value is specified, the default is `'sorted'`.

`[C,ia] = setdiff(A,B,'legacy')` and `[C,ia] = setdiff( A,B,'rows','legacy')` preserve the behavior of the `setdiff` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays, tables, datetime arrays, or duration arrays.

Examples

Difference of Two Vectors

Define two vectors with values in common.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
```

Find the values in `A` that are not in `B`.

```
C = setdiff(A,B)
```

```
C =

     1     3     5
```

Difference of Two Tables

Define two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
A =

    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
```

```
    2       B       true
    3       C       false
    4       D       true
    5       E       false


B =

    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    3       C       false
    5       E       false
    7       G       false
    9       I       false
```

Find the rows in `A` that are not in `B`.

```
C = setdiff(A,B)
```

```
C =

    Var1    Var2    Var3
    ----    ----    -----
    2       B       true
    4       D       true
```

Difference of Two Vectors and Indices to Different Values

Define two vectors with values in common.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
```

Find the values in `A` that are not in `B` as well as the index vector `ia`, such that `C = A(ia)`.

```
[C,ia] = setdiff(A,B)
```

```
C =

     1     3     5

ia =

     4
     1
```

```
     5
```

Difference of Two Tables and Indices to Different Rows

Define a table, `A`, of gender, age, and height for five people.

```
A = table(['M';'M';'F';'M';'F'],[27;52;31;46;35],[74;68;64;61;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty' 'Bob' 'Judy'})
```

```
A =

             Gender    Age    Height
             ------    ---    ------
    Ted      M         27     74
    Fred     M         52     68
    Betty    F         31     64
    Bob      M         46     61
    Judy     F         35     64
```

Define a table, `B`, with the same variables as `A`.

```
B = table(['F';'M';'F';'F'],[64;68;62;58],[31;47;35;23],...
'VariableNames',{'Gender' 'Height' 'Age'},...
'RowNames',{'Meg' 'Joe' 'Beth' 'Amy'})
```

```
B =

             Gender    Height    Age
             ------    ------    ---
    Meg      F         64        31
    Joe      M         68        47
    Beth     F         62        35
    Amy      F         58        23
```

Find the rows in `A` that are not in `B`, as well as the index vector `ia`, such that `C = A(ia,:)`.

```
[C,ia] = setdiff(A,B)
```

```
C =

             Gender    Age    Height
             ------    ---    ------
    Judy     F         35     64
    Ted      M         27     74
```

```
    Bob      M          46      61
    Fred     M          52      68


ia =

     5
     1
     4
     2
```

The rows of `C` are in sorted order first by `Gender` and next by `Age`.

Difference of Rows in Two Matrices

Define two matrices with rows in common.

```
A = [7 9 7; 0 0 0; 7 9 7; 5 5 5; 1 4 5];
B = [0 0 0; 5 5 5];
```

Find the rows from `A` that are not in `B` as well as the index vector `ia`, such that `C = A(ia,:)`.

```
[C,ia] = setdiff(A,B,'rows')

C =

     1     4     5
     7     9     7

ia =

     5
     1
```

Difference of Two Vectors with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` or `'sorted'` when the order of the values in `C` are important.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
[C,ia] = setdiff(A,B,'stable')

C =
```

```
     3     1     5

ia =

     1
     4
     5
```

Alternatively, you can specify `'sorted'` order.

```
 [C,ia] = setdiff(A,B,'sorted')

C =

     1     3     5

ia =

     4
     1
     5
```

Difference of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN];
```

Find the set difference of `A` and `B`.

```
C = setdiff(A,B)

C =

   NaN   NaN
```

`setdiff` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the strings in A that are not in B.

```
[C,ia] = setdiff(A,B)
```

```
C =

    'dog'    'fish'


ia =

     1
     3
```

`setdiff` treats trailing white space in cell arrays of strings as distinct characters.

Difference of Char and Cell Array of Strings

Create a character array, A.

```
A = ['cat';'dog';'fox';'pig'];
class(A)
```

```
ans =

char
```

Create a cell array of strings, B.

```
B={'dog','cat','fish','horse'};
class(B)
```

```
ans =

cell
```

Find the strings in A that are not in B.

```
C = setdiff(A,B)
```

```
C =

    'fox'
    'pig'
```

The result, C, is a cell array of strings.

```
class(C)
```

```
ans =

cell
```

Preserve Legacy Behavior of setdiff

Use the `'legacy'` flag to preserve the behavior of `setdiff` from R2012b and prior releases in your code.

Find the difference of A and B with the current behavior.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
[C1,ia1] = setdiff(A,B)
```

```
C1 =

     1     3     5

ia1 =

     4
     1
     5
```

Find the difference of A and B, and preserve the legacy behavior.

```
[C2,ia2] = setdiff(A,B,'legacy')
```

```
C2 =

     1     3     5

ia2 =

     7     1     5
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of strings | tables

Input arrays, specified as numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of strings, or tables.

A and B must belong to the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.
- Categorical arrays can combine with cell arrays of strings or single strings.
- Datetime arrays can combine with cell arrays of date strings or single date strings.

If A and B are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither A nor B are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of C are the sorted union of the categories from A and B.

If you specify the `'rows'` option, A and B must have the same number of columns.

If A and B are tables, they must have the same variable names. Conversely, the row names do not matter. Two rows that have the same values, but different names, are considered equal.

If A and B are datetime arrays, they must be consistent with each other in whether they specify a time zone.

Furthermore, A and B can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `eq`
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class.

`setOrder` — Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in C.

Order Flag	Meaning
`'sorted'`	The values (or rows) in C return in sorted order. For example: `C = setdiff([4 1 3 2],[2 1],'sorted')` returns `C = [3 4]`.

Order Flag	Meaning
`'stable'`	The values (or rows) in `C` return in the same order as in `A`. For example: `C = setdiff([4 1 3 2],[2 1],'stable')` returns `C = [4 3]`.

Output Arguments

`C` — Difference of `A` and `B`

vector | matrix | table

Difference of `A` and `B`, returned as a vector, matrix, or table. If the inputs `A` and `B` are tables, the order of the variables in the resulting table, `C`, is the same as the order of the variables in `A`.

The following describes the shape of `C` when the inputs are vectors or matrices and when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified and `A` is a row vector, then `C` is a row vector.
- If the `'rows'` flag is not specified and `A` is not a row vector, then `C` is a column vector.
- If the`'rows'` flag is specified, then `C` is a matrix containing the rows of `A` that are not in `B`.
- If all the values (or rows) of `A` are also in `B`, then `C` is an empty matrix.

The class of `C` is the same as the class of `A`, unless:

- `A` is a character array and `B` is a cell array of strings, in which case `C` is a cell array of strings.
- `A` is a cell array of strings or single string and `B` is a categorical array, in which case `C` is a categorical array.
- `A` is a cell array of strings or single string and `B` is a datetime array, in which case `C` is a datetime array.

`ia` — Index to `A`

column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in `A` that are not in `B`. If there is a repeated value (or row)

appearing exclusively in `A`, then `ia` contains the index to the first occurrence of the value (or row).

More About

Tips

- To find the set difference with respect to a subset of variables from a table, you can use column subscripting. For example, you can use `setdiff(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.
- "Combine Categorical Arrays"

See Also

`intersect` | `ismember` | `issorted` | `setxor` | `sort` | `union` | `unique`

Introduced before R2006a

setDirectory

Class: Tiff

Make specified IFD current IFD

Syntax

```
setDirectory(tiffobj,dirNum)
```

Description

`setDirectory(tiffobj,dirNum)` sets the image file directory (IFD) specified by `dirNum` as the current IFD. `Tiff` object methods operate on the current IFD. The directory index number is one-based.

Examples

Open a TIFF file and move to an IFD in the file by specifying its index number. The TIFF file should contain multiple images.

```
t = Tiff('example.tif','r');
setDirectory(t,2)
close(t)
```

References

This method corresponds to the `TIFFSetDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.currentDirectory` | `Tiff.nextDirectory`

setenv

Set environment variable

Syntax

```
setenv(name,value)
setenv(name)
```

Description

`setenv(name,value)` sets the value of an environment variable belonging to the underlying operating system. Inputs `name` and `value` are both strings. If `name` already exists as an environment variable, then `setenv` replaces its current value with the string given in `value`. If `name` does not exist, `setenv` creates a new environment variable called `name` and assigns `value` to it.

`setenv(name)` is equivalent to `setenv(name, '')` and assigns a null value to the variable `name`. On the Microsoft Windows platform, this is equivalent to undefining the variable. On most UNIX platforms, it is possible to have an environment variable defined as empty.

The maximum number of characters in `name` is 2^{15} - 2 (or 32766). If `name` contains the character `=`, `setenv` throws an error. The behavior of environment variables with `=` in the name is not well-defined.

On all platforms, `setenv` passes the `name` and `value` strings to the operating system unchanged. Special characters such as `;`, `/`, `:`, `$`, `%`, etc. are left unexpanded and intact in the variable value.

Values assigned to variables using `setenv` are picked up by any process that is spawned using the MATLAB `system`, `unix`, `dos` or `!` functions. You can retrieve any value set with `setenv` by using `getenv(name)`.

Examples

% Set and retrieve a new value for the environment variable `TEMP`:

```
setenv('TEMP', 'C:\TEMP');
getenv('TEMP')
```

% Append the `Perl\bin` folder to your system `PATH` variable:

```
setenv('PATH', [getenv('PATH') ';D:\Perl\bin']);
```

See Also

`getenv` | `system` | `unix` | `dos` | `!`

setfield

Assign values to structure array field

Syntax

```
s = setfield(s,'field',value)
s = setfield(s,{sIndx1,...,sIndxM},'field',
{fIndx1,...,fIndxN},value)
```

Description

`s = setfield(s,'field',value)`, where `s` is a 1-by-1 structure, sets the contents of the specified field, equivalent to `s.field = value`. If `s` does not contain the specified `field`, the `setfield` function creates the field and assigns the specified value. Pass field references as strings.

`s = setfield(s,{sIndx1,...,sIndxM},'field',
{fIndx1,...,fIndxN},value)` sets the contents of the specified field, equivalent to `s(sIndx1,...,sIndxM).field(fIndx1,...,fIndxN) = value`. The `setfield` function supports multiple sets of `field` and `fIndx` inputs. If structure `s` or any of the fields is a nonscalar structure, the `Indx` inputs associated with that input are required. Otherwise, the `Indx` inputs are optional. If you specify a single colon operator for an index input, enclose it in single quotation marks: `':'`.

Examples

Add values to a structure that contains nested fields:

```
grades = [];
level = 5;
semester = 'Fall';
subject = 'Math';
student = 'John_Doe';
fieldnames = {semester subject student}
newGrades_Doe = [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
```

```
grades = setfield(grades, {level}, ...
                  fieldnames{:}, {10, 21:30}, ...
                  newGrades_Doe);

% View the new contents.
grades(level).(semester).(subject).(student)(10, 21:30)
```

Using the structure defined in the previous example, remove the tenth row of the specified field:

```
grades = setfield(grades, {level}, fieldnames{:}, {10,':'}, []);
```

More About

Tips

- For most cases, add data to a structure array by indexing rather than using the `setfield` function. For more information, see "Access Data in a Structure Array" and "Generate Field Names from Variables".
- Call `setfield` to simplify references to structure arrays with nested fields, as shown in the Examples section.

- "Generate Field Names from Variables"
- "Access Data in a Structure Array"

See Also

`getfield` | `fieldnames` | `isfield` | `orderfields` | `rmfield`

Introduced before R2006a

setpixelposition

Set component position in pixels

Syntax

```
setpixelposition(handle,position)
setpixelposition(handle,position,recursive)
```

Description

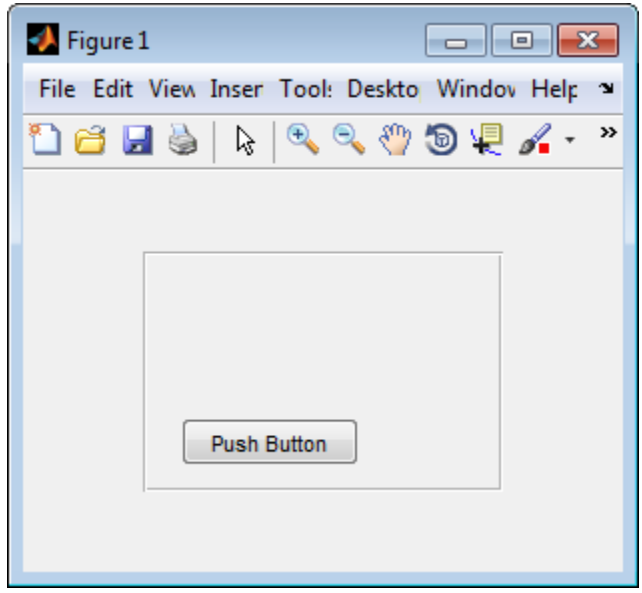
`setpixelposition(handle,position)` sets the position of the component specified by `handle`, to the specified position relative to its parent. `position` is a four-element vector that specifies the location and size of the component: [pixels from left, pixels from bottom, pixels across, pixels high].

`setpixelposition(handle,position,recursive)` sets the position as above. If Boolean `recursive` is true, the position is set relative to the parent figure of `handle`.

Examples

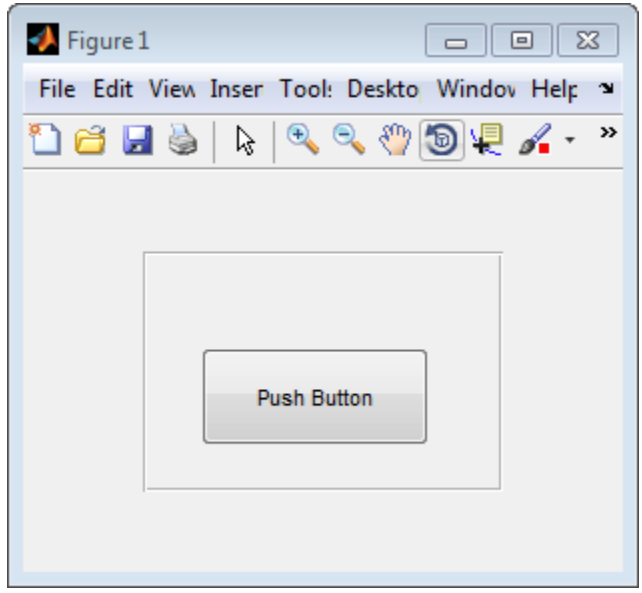
This example first creates a push button within a panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton',...
               'Units','normalized',...
               'String','Push Button',...
               'Position',[.1 .1 .5 .2]);
```



The example then retrieves the position of the push button and changes its position with respect to the panel.

```
pos1 = getpixelposition(h1);
setpixelposition(h1,pos1 + [10 10 25 25]);
```



See Also

`getpixelposition` | `uicontrol` | `uipanel`

setpref

Set preference

Syntax

```
setpref('group','pref',val)
setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,valn})
```

Description

`setpref('group','pref',val)` sets the preference specified by `group` and `pref` to the value `val`. Individual preference values can be any MATLAB data type, including numeric types, strings, cell arrays, structures, and objects. Setting a preference that does not yet exist causes it to be created.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g., `'MathWorks_GUIDE_ApplicationPrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,valn})` sets each preference specified in the cell array of names to the corresponding value.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

Use `addpref` to create a preference group called `mytoolbox` and a preference within it called `version`, and then modify the contents of `version` using `setpref`:

```
addpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')

ans =
```

```
1.0

setpref('mytoolbox','version',{'1.0','beta'})
getpref('mytoolbox','version')

ans =
    '1.0'    'beta'
```

See Also

addpref | ispref | rmpref | getpref | uigetpref | uisetpref

Introduced before R2006a

setstr

Set string flag

Note: `setstr` is not recommended. Use `char` instead.

Description

This MATLAB 4 function has been renamed `char` in MATLAB 5.

Introduced before R2006a

setSubDirectory

Class: Tiff

Make subIFD specified by byte offset current IFD

Syntax

```
setSubDirectory(tiffobj,offset)
```

Description

`setSubDirectory(tiffobj,offset)` sets the subimage file directory (subIFD) specified by `offset` the current IFD. The offset value is given in bytes. Use this method when you want to access subIFDs linked through the `SubIFD` tag.

Examples

Set Subimage File Directory

Open a TIFF file and read the value of the `SubIFD` tag in the current IFD. The `SubIFD` tag contains byte offsets that specify the location of subIFDs in the IFD. The TIFF file should contain subIFDs.

```
t = Tiff('example.tif','r');
```

Read the value of the SubIFD tag to get subdirectory offsets.

```
offsets = getTag(t,'SubIFD');
```

Set one of the subdirectories (if more than one) as the current directory.

```
setSubDirectory(t,offsets(1))
close(t)
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

References

This method corresponds to the `TIFFSetSubDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.setDirectory`

setTag

Class: Tiff

Set value of tag

Syntax

```
setTag(tiffobj,tagId,tagValue)
setTag(tiffobj,tagStruct)
```

Description

`setTag(tiffobj,tagId,tagValue)` sets the value of the TIFF tag specified by `tagId` to the value specified by `tagValue`. You can specify `tagId` as a character string (`'ImageWidth'`) or using the numeric tag identifier defined by the TIFF specification (256). To see a list of all the tags with their numeric identifiers, view the value of the Tiff object `TagID` property. Use the `TagID` property to specify the value of a tag. For example, `Tiff.TagID.ImageWidth` is equivalent to the tag's numeric identifier.

`setTag(tiffobj,tagStruct)` sets the values of all of the tags with name/value fields in `tagStruct`. The names of fields in `tagstruct` must be the name of TIFF tags.

Note: If you are modifying a tag rather than creating it, you must use the `Tiff.rewriteDirectory` method after using the `Tiff.setTag` method.

Examples

Set Tag Values

Write TIFF tags and image data to a new TIFF file.

Read sample data into an array, `imdata`. Create a `Tiff` object associated with a new file, `myfile.tif`, and open the file for writing.

```
imdata = imread('example.tif');
```

```
t = Tiff('myfile.tif','w');
```

Set tag values by specifying the numeric tag identifier. Use the `TagID` property to obtain the tag identifier.

```
setTag(t,Tiff.TagID.ImageLength,size(imdata,1))
setTag(t,Tiff.TagID.ImageWidth,size(imdata,2))
```

Set tag values by specifying the tag name.

```
setTag(t,'Photometric', Tiff.Photometric.RGB)
setTag(t,'PlanarConfiguration', Tiff.PlanarConfiguration.Chunky)
```

Create a structure with fields named after TIFF tags and assign values to the fields. Pass this structure to the `setTag` method to set the values of these tags.

```
tagStruct.BitsPerSample = 8;
tagStruct.SamplesPerPixel = 3;
tagStruct.TileWidth = 128;
tagStruct.TileLength = 128;
tagStruct.Compression = Tiff.Compression.JPEG;
tagStruct.Software = 'MATLAB';
setTag(t,tagStruct)
```

Write the image data to the TIFF file.

```
write(t,imdata)
close(t)
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

References

This method corresponds to the `TIFFSetField` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.getTag`

settimeseriesnames

Change name of `timeseries` object in `tscollection`

Syntax

```
tsc = settimeseriesnames(tsc,old,new)
```

Description

`tsc = settimeseriesnames(tsc,old,new)` replaces the `old` name of `timeseries` object with the new name in tsc.

See Also

`tscollection`

Introduced before R2006a

setxor

Set exclusive OR of two arrays

Syntax

```
C = setxor(A,B)
C = setxor(A,B,'rows')
[C,ia,ib] = setxor(A,B)
[C,ia,ib] = setxor(A,B,'rows')

[C,ia,ib] = setxor( ___ ,setOrder)

[C,ia,ib] = setxor(A,B,'legacy')
[C,ia,ib] = setxor(A,B,'rows','legacy')
```

Description

`C = setxor(A,B)` returns the data of `A` and `B` that are not in their intersection (the symmetric difference).

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `setxor` returns the values that occur in `A` or `B`, but not both. The values of `C` are in sorted order.
- If `A` and `B` are tables, then `setxor` returns the rows that occur in one or the other of the two tables, but not both. The rows of table `C` are in sorted order.

`C = setxor(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the rows of matrices `A` and `B` that are not in their intersection. The rows of `C` are in sorted order.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia,ib] = setxor(A,B)` also returns index vectors `ia` and `ib`.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `C` is a sorted combination of the elements `A(ia)` and `B(ib)`.

- If A and B are tables, then C is a sorted combination of the rows of A(ia,:) and B(ib,:).

[C,ia,ib] = setxor(A,B,'rows') also returns index vectors ia and ib, such that C is a sorted combination of the rows of A(ia,:) and B(ib,:).

[C,ia,ib] = setxor(___ ,setOrder) returns C in a specific order using any of the input arguments in the previous syntaxes. setOrder='sorted' returns the values (or rows) of C in sorted order. setOrder='stable' returns the values (or rows) of C in the same order as A and B. If no value is specified, the default is 'sorted'.

[C,ia,ib] = setxor(A,B,'legacy') and [C,ia,ib] = setxor(A,B,'rows','legacy') preserve the behavior of the setxor function from R2012b and prior releases.

The 'legacy' option does not support categorical arrays, tables, datetime arrays, or duration arrays.

Examples

Symmetric Difference of Two Vectors

Define two vectors with a value in common.

```
A = [5 1 3 3 3]; B = [4 1 2];
```

Find the values of A and B that are not in their intersection.

```
C = setxor(A,B)
```

```
C =

     2     3     4     5
```

Symmetric Difference of Two Tables

Define two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
A =
```

```
    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    2       B       true
    3       C       false
    4       D       true
    5       E       false


B =

    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    3       C       false
    5       E       false
    7       G       false
    9       I       false
```

Find the rows of `A` and `B` that are not in their intersection.

```
C = setxor(A,B)
```

```
C =

    Var1    Var2    Var3
    ----    ----    -----
    2       B       true
    4       D       true
    7       G       false
    9       I       false
```

Symmetric Difference of Two Vectors and Indices to Different Values

Define two vectors with a value in common.

```
A = [5 1 3 3 3]; B = [4 1 2];
```

Find the values of `A` and `B` that are not in their intersection as well as the index vectors `ia` and `ib`.

```
[C,ia,ib] = setxor(A,B)
```

```
C =
```

```
     2     3     4     5


ia =

     3
     1


ib =

     3
     1
```

`C` is a sorted combination of the elements `A(ia)` and `B(ib)`.

Symmetric Difference of Two Tables and Indices to Different Rows

Define a table, `A`, of gender, age, and height for five people.

```
A = table(['M';'M';'F'],[27;52;31],[74;68;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty'})
```

```
A =

              Gender    Age    Height
              ------    ---    ------
    Ted       M         27     74
    Fred      M         52     68
    Betty     F         31     64
```

Define a table, `B`, with the same variables as `A`.

```
B = table(['F';'M'],[64;68],[31;47],...
'VariableNames',{'Gender' 'Height' 'Age'},...
'RowNames',{'Meg' 'Joe'})
```

```
B =

            Gender    Height    Age
            ------    ------    ---
    Meg     F         64        31
    Joe     M         68        47
```

Find the rows of A and B that are not in their intersection, as well as the index vectors ia and ib.

```
[C,ia,ib] = setxor(A,B)
```

```
C =

            Gender    Age    Height
            ------    ---    ------
    Ted     M         27     74
    Joe     M         47     68
    Fred    M         52     68


ia =

     1
     2


ib =

     2
```

C is a sorted combination of the elements A(ia,:) and B(ib,:).

Symmetric Difference of Rows in Two Matrices

Define two matrices with rows in common.

```
A = [7 8 9; 7 7 1; 7 7 1; 1 2 3; 4 5 6];
B = [1 2 3; 4 5 6; 7 7 2];
```

Find the rows of A and B that are not in their intersection as well as the index vectors ia and ib.

```
[C,ia,ib] = setxor(A,B,'rows')
```

```
C =

     7     7     1
     7     7     2
     7     8     9
```

```
ia =

     2
     1


ib =

     3
```

`C` is a sorted combination of the rows of `A(ia,:)` and `B(ib,:)`.

Symmetric Difference of Two Vectors in Specified Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` if you want the values in `C` to have the same order as `A` and `B`.

```
A = [5 1 3 3 3]; B = [4 1 2];
[C,ia,ib] = setxor(A,B,'stable')
```

```
C =

     5     3     4     2


ia =

     1
     3


ib =

     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C,ia,ib] = setxor(A,B,'sorted')
```

```
C =

     2     3     4     5
```

```
ia =

     3
     1


ib =

     3
     1
```

Symmetric Difference of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN NaN];
```

Find the symmetric difference of vectors `A` and `B`.

```
C = setxor(A,B)
```

```
C =

   NaN   NaN   NaN   NaN
```

The `setxor` function treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the strings that are not in the intersection of `A` and `B`.

```
[C,ia,ib] = setxor(A,B)
```

```
C =

    'dog'    'dog '    'fish'    'fish '
```

```
ia =

     1
     3


ib =

     1
     3
```

`setxor` treats trailing white space in cell arrays of strings as distinct characters.

Symmetric Difference of Vectors of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A =

A
B
C


ans =

char
```

Create a row vector containing elements of numeric type `double`.

```
B = [66 67 68], class(B)
```

```
B =

    66    67    68


ans =

double
```

Find the symmetric difference of `A` and `B`.

```
C = setxor(A,B)

C =

A
D
```

The result is a column vector character array.

```
class(C)

ans =

char
```

Symmetric Difference of Char and Cell Array of Strings

Create a character array, A.

```
A = ['cat';'dog';'fox';'pig'];
class(A)

ans =

char
```

Create a cell array of strings, B.

```
B={'dog','cat','fish','horse'};
class(B)

ans =

cell
```

Find the strings that are not in the intersection of A and B.

```
C = setxor(A,B)

C =

    'fish'
    'fox'
    'horse'
    'pig'
```

The result, C, is a cell array of strings.

```
class(C)
```

```
ans =

cell
```

Preserve Legacy Behavior of setxor

Use the 'legacy' flag to preserve the behavior of setxor from R2012b and prior releases in your code.

Find the symmetric difference of A and B with the current behavior.

```
A = [5 1 3 3 3]; B = [4 1 2 2];
[C1,ia1,ib1] = setxor(A,B)
```

```
C1 =

     2     3     4     5


ia1 =

     3
     1


ib1 =

     3
     1
```

Find the symmetric difference and preserve the legacy behavior.

```
[C2,ia2,ib2] = setxor(A,B,'legacy')
```

```
C2 =

     2     3     4     5


ia2 =
```

```
     5     1

ib2 =

     4     1
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of strings | tables

Input arrays, specified as numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of strings, or tables.

`A` and `B` must belong to the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.
- Categorical arrays can combine with cell arrays of strings or single strings.
- Datetime arrays can combine with cell arrays of date strings or single date strings.

If `A` and `B` are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither `A` nor `B` are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of `C` are the sorted union of the categories from `A` and `B`.

If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

If `A` and `B` are tables, they must have the same variable names. Conversely, the row names do not matter. Two rows that have the same values, but different names, are considered equal.

If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

Furthermore, `A` and `B` can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)

- `eq`
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class.

`setOrder` — Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Order Flag	Meaning
`'sorted'`	The values (or rows) in `C` return in sorted order. For example: `C = setxor([5 1 3],[4 1 2],'sorted')` returns `C = [2 3 4 5]`.
`'stable'`	The values (or rows) in `C` return in the same order as in `A` and `B`. For example: `C = setxor([5 1 3],[4 1 2],'stable')` returns `C = [5 3 4 2]`.

Output Arguments

`C` — Symmetric difference array

vector | matrix | table

Symmetric difference array, returned as a vector, matrix, or table. If the inputs `A` and `B` are tables, the order of the variables in the resulting table, `C`, is the same as the order of the variables in `A`.

The following describes the shape of `C` when the inputs are vector or matrices and when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified, then C is a column vector unless both `A` and `B` are row vectors.
- If the `'rows'` flag is not specified and both `A` and `B` are row vectors, then `C` is a row vector.
- If the`'rows'` flag is specified, then `C` is a matrix containing the rows of `A` and `B` that are not in the intersection.

- If all the values (or rows) of A are also in B, then C is an empty matrix.

The class of the inputs A and B determines the class of C:

- If the class of A and B are the same, then C is the same class.
- If you combine a `char` or nondouble numeric class with `double`, then C is the same class as the nondouble input.
- If you combine a `logical` class with `double`, then C is `double`.
- If you combine a cell array of strings with `char`, then C is a cell array of strings.
- If you combine a categorical array with a cell array of strings or single string, then C is a categorical array.
- If you combine a datetime array with a cell array of date strings or single date string, then C is a datetime array.

`ia` — Index to A

column vector

Index to A, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in A that contribute to the symmetric difference. If there is a repeated value (or row) appearing exclusively in A, then `ia` contains the index to the first occurrence of the value (or row).

`ib` — Index to B

column vector

Index to B, returned as a column vector when the `'legacy'` flag is not specified. `ib` identifies the values (or rows) in B that contribute to the symmetric difference. If there is a repeated value (or row) appearing exclusively in B, then `ib` contains the index to the first occurrence of the value (or row).

More About

Tips

- To find the symmetric difference with respect to a subset of variables from a table, you can use column subscripting. For example, you can use `setxor(A(:,`*`vars`*`),B(:,`*`vars`*`))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.

- “Combine Categorical Arrays”

See Also

`intersect` | `ismember` | `issorted` | `setdiff` | `sort` | `union` | `unique`

Introduced before R2006a

shading

Set color shading properties

Syntax

```
shading flat
shading faceted
shading interp
shading(axes_handle,...)
```

Description

The `shading` function controls the color shading of surface and patch graphics objects.

`shading flat` each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.

`shading faceted` flat shading with superimposed black mesh lines. This is the default shading mode.

`shading interp` varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.

`shading(axes_handle,...)` applies the shading type to the objects in the axes specified by `axes_handle`, instead of the current axes. Use quoted strings when using a function form. For example:

```
shading(gca,'interp')
```

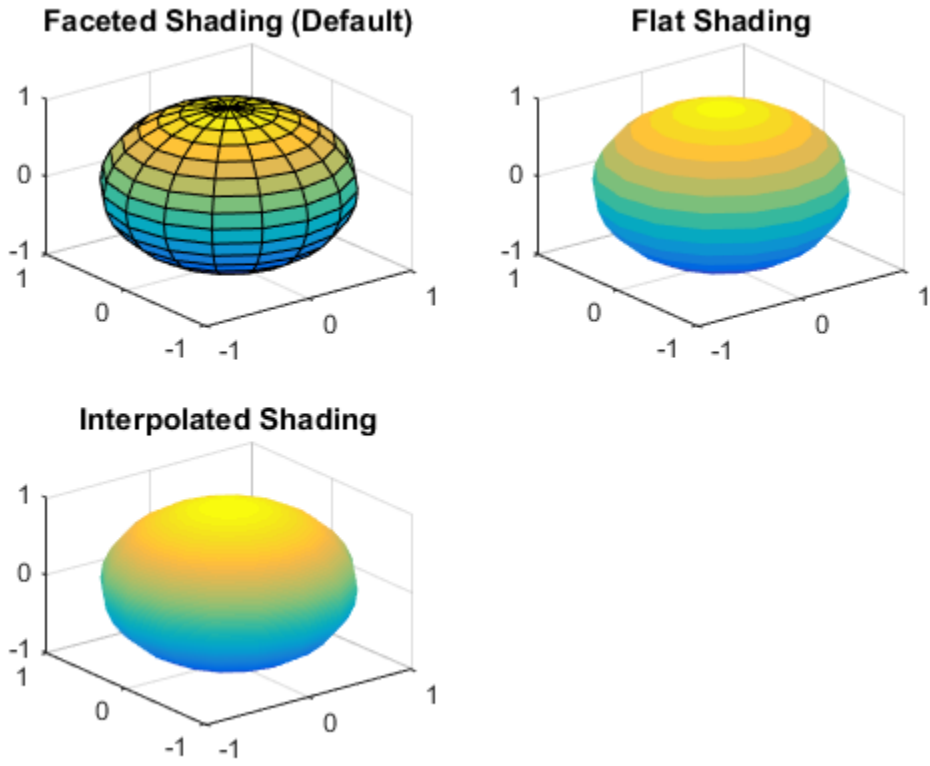
Examples

Display Sphere with Different Types of Shading

Plot the `sphere` function and use different types of shading.

```
figure
subplot(2,2,1)
```

```
sphere(16)
title('Faceted Shading (Default)')

subplot(2,2,2)
sphere(16)
shading flat
title('Flat Shading')

subplot(2,2,3)
sphere(16)
shading interp
title('Interpolated Shading')
```



More About

Algorithms

`shading` sets the `EdgeColor` and `FaceColor` properties of all surface and patch graphics objects in the current axes. `shading` sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

`fill` | `fill3` | `hidden` | `light` | `lighting` | `mesh` | `patch` | `pcolor` | `surf`

Introduced before R2006a

shg

Show most recent graph window

Syntax

```
shg
```

Description

`shg` makes the current figure visible and raises it above all other figures on the screen. This is identical to using the command `figure(gcf)`.

See Also

`figure` | `gca` | `gcf`

shiftdim

Shift dimensions

Syntax

```
B = shiftdim(X,n)
[B,nshifts] = shiftdim(X)
```

Description

`B = shiftdim(X,n)` shifts the dimensions of `X` by `n`. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons.

`[B,nshifts] = shiftdim(X)` returns the array `B` with the same number of elements as `X` but with any leading singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. `nshifts` is the number of dimensions that are removed.

If `X` is a scalar, `shiftdim` has no effect.

Examples

The `shiftdim` command is handy for creating functions that, like `sum` or `diff`, work along the first nonsingleton dimension.

```
a = rand(1,1,3,1,2);
[b,n]  = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
c = shiftdim(b,-n);   % c == a.
d = shiftdim(a,3);    % d is 1-by-2-by-1-by-1-by-3.
```

See Also

`circshift` | `reshape` | `squeeze` | `permute` | `ipermute`

Introduced before R2006a

showplottool

Show or hide figure plot tool

Syntax

```
showplottool('tool')
showplottool('on','tool')
showplottool('off','tool')
showplottool('toggle','tool')
showplottool(figure_handle,...)
```

Description

`showplottool('tool')` shows the specified plot tool on the current figure. *`tool`* can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

`showplottool('on','tool')` shows the specified plot tool on the current figure.

`showplottool('off','tool')` hides the specified plot tool on the current figure.

`showplottool('toggle','tool')` toggles the visibility of the specified plot tool on the current figure.

`showplottool(figure_handle,...)` operates on the specified figure instead of the current figure.

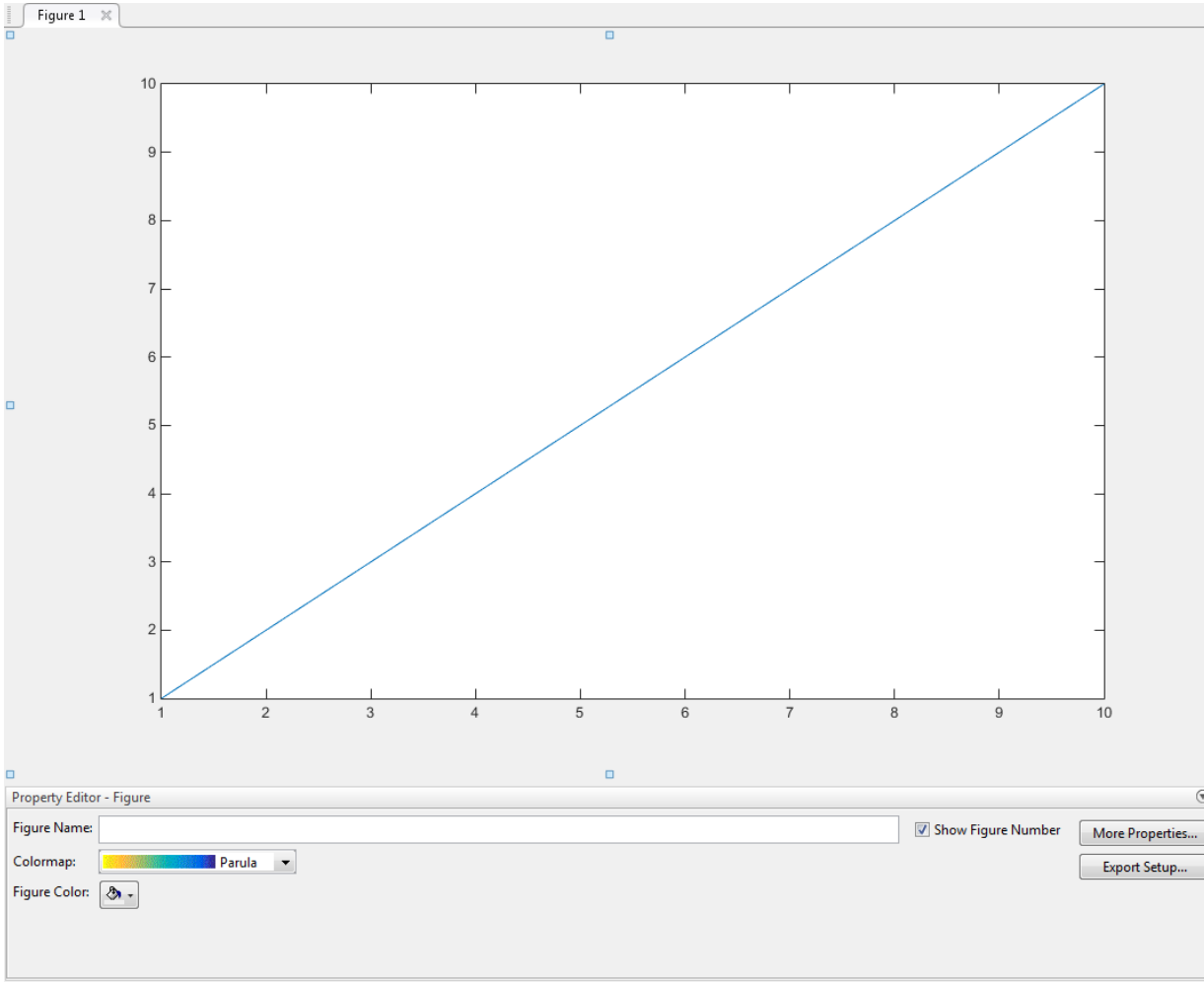
Note: When you dock, undock, resize, or reposition a plotting tool and then close it, it will still be configured as you left it the next time you open it. There is no command to reset plotting tools to their original, default locations.

Examples

Open One of the Plot Tools

Create a simple plot and open the property editor.

```
plot(1:10);
showplottool('propertyeditor')
```



Alternatives

Click the larger Plotting Tools icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Individually select

the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see “Customize Graph Using Plot Tools”.

See Also

`figurepalette` | `plotbrowser` | `plottools` | `propertyeditor`

Introduced before R2006a

shrinkfaces

Reduce size of patch faces

Syntax

```
shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p)
nfv = shrinkfaces(f,v,sf)
[nf,nv] = shrinkfaces(...)
```

Description

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, the MATLAB software creates nonshared vertices before performing the face-area reduction.

`nfv = shrinkfaces(p,sf)` returns the face and vertex data in the struct `nfv`, but does not set the `Faces` and `Vertices` properties of patch `p`.

`nfv = shrinkfaces(fv,sf)` uses the face and vertex data from the struct `fv`.

`shrinkfaces(p)` and `shrinkfaces(fv)` (without specifying a shrink factor) assume a shrink factor of 0.3.

`nfv = shrinkfaces(f,v,sf)` uses the face and vertex data from the arrays `f` and `v`.

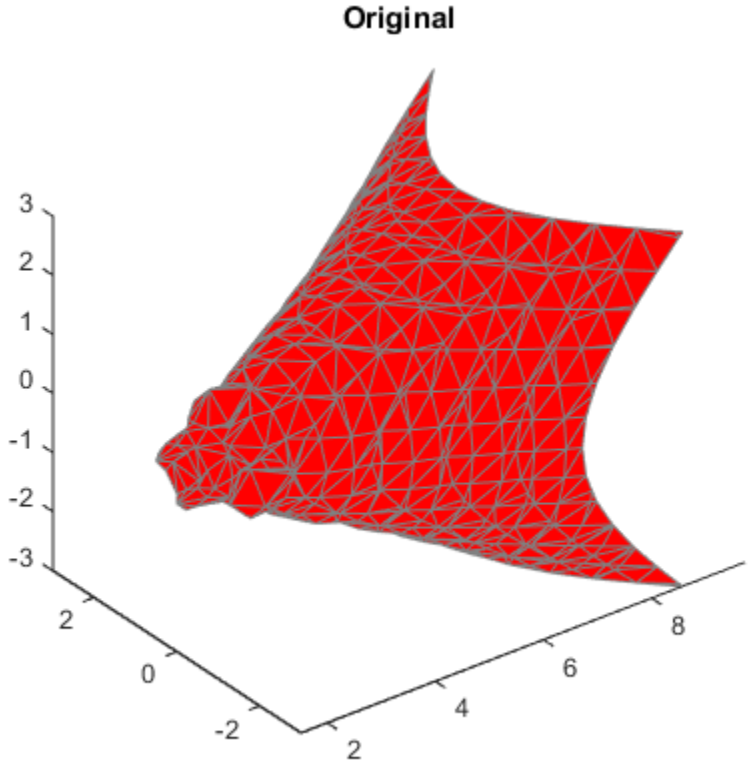
`[nf,nv] = shrinkfaces(...)` returns the face and vertex data in two separate arrays instead of a struct.

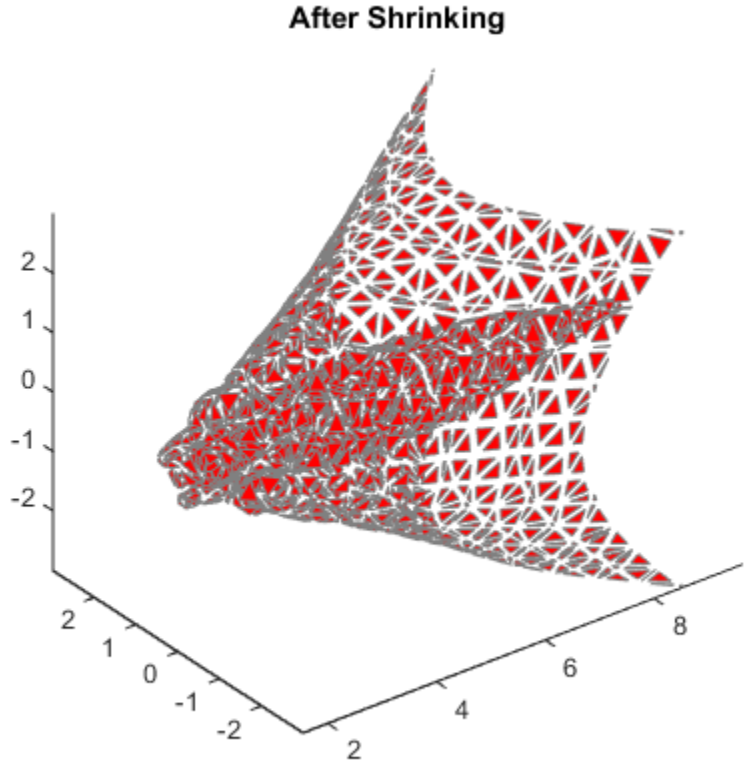
Examples

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). Two isosurfaces provide a before and after view of the effects of shrinking the face size.

- First `reducevolume` samples the flow data at every other point and then `isosurface` generates the faces and vertices data.
- The `patch` command accepts the face/vertex struct and draws the first (`p1`) isosurface.
- Use the `daspect`, `view`, and `axis` commands to set up the view and then add a `title`.
- The `shrinkfaces` command modifies the face/vertex data and passes it directly to `patch`.

```
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
p1.FaceColor = 'red';
p1.EdgeColor = [0.5 0.5 0.5];
daspect([1 1 1]);
view(3);
axis tight
title('Original')

figure
p2 = patch(shrinkfaces(fv,.3));
p2.FaceColor = 'red';
p2.EdgeColor = [0.5 0.5 0.5];
daspect([1 1 1]);
view(3);
axis tight
title('After Shrinking')
```


Original
3
2
1
0
-1
-2
-3
2
0
-2
2
4
6
8



See Also

`isosurface` | `patch` | `reducevolume` | `daspect` | `view` | `axis`

Introduced before R2006a

sign

Signum function

Syntax

```
Y = sign(X)
```

Description

`Y = sign(X)` returns an array `Y` the same size as `X`, where each element of `Y` is:

- `1` if the corresponding element of `X` is greater than zero
- `0` if the corresponding element of `X` equals zero
- `-1` if the corresponding element of `X` is less than zero

For nonzero complex `X`, `sign(X) = X./abs(X)`.

See Also

`abs` | `conj` | `imag` | `real`

Introduced before R2006a

sin

Sine of argument in radians

Syntax

```
Y = sin(X)
```

Description

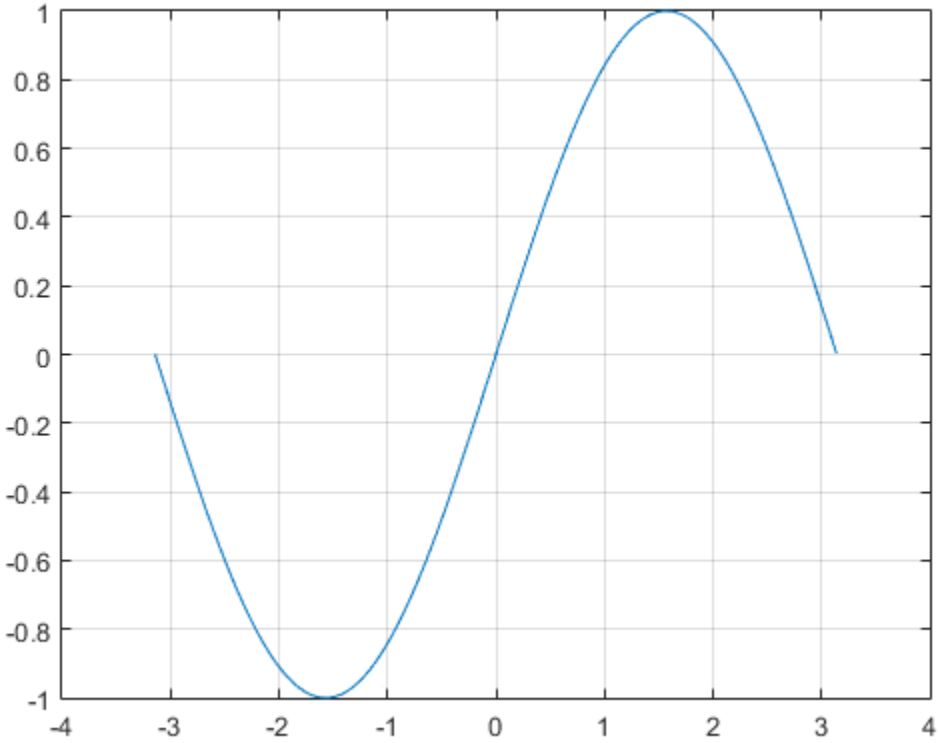
`Y = sin(X)` returns the sine of the elements of `X`. The `sin` function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval [-Inf, Inf], `sin` returns real values in the interval [-1 ,1]. For complex values of X, `sin` returns complex values. All angles are in radians.

Examples

Plot Sine Function

Plot the sine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,sin(x)), grid on
```



Sine of Vector of Complex Angles

Calculate the sine of the complex angles in vector `x`.

```
x = [-i pi+i*pi/2 -1+i*4];
y = sin(x)
```

```
y =

   0.0000 - 1.1752i   0.0000 - 2.3013i -22.9791 +14.7448i
```

Input Arguments

X — Input angle in radians

number | vector | matrix | multidimensional array

Input angle in radians, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Sine of input angle

scalar value | vector | matrix | N-D array

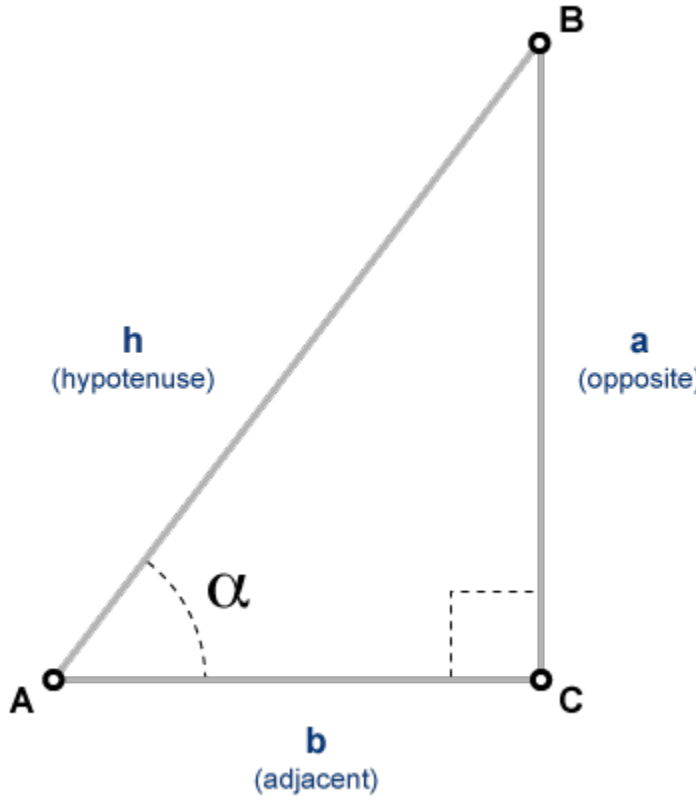
Sine of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or N-D array.

More About

Sine Function

The sine of an angle, α, defined with reference to a right angled triangle is

$$\sin(\alpha) = \frac{\text{opposite side}}{\text{hypotenuse}} = \frac{a}{h}.$$



The sine of a complex angle, α, is

$$\sin(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{2i}.$$

See Also

`asin` | `asind` | `sind` | `sinh`

Introduced before R2006a

sind

Sine of argument in degrees

Syntax

```
Y = sind(X)
```

Description

`Y = sind(X)` returns the sine of the elements in `X`, which are expressed in degrees.

Examples

Sine of 180 degrees compared to sine of π radians

```
sind(180)
```

```
ans =

     0
```

```
sin(pi)
```

```
ans =

   1.2246e-16
```

Sine of vector of complex angles, specified in degrees

```
z = [90+i 15+2i 10+3i];
y = sind(z)
```

```
y =
```

```
   1.0002        0.2590 + 0.0337i   0.1739 + 0.0516i
```

Input Arguments

X — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `sind` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Sine of angle

scalar value | vector | matrix | N-D array

Sine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

See Also

`asin` | `asind` | `sin`

Introduced before R2006a

single

Convert to single precision

Syntax

```
B = single(A)
```

Description

`B = single(A)` converts the matrix `A` to single precision, returning that value in `B`. `A` can be any numeric object (such as a `double`). If `A` is already single precision, `single` has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The `single` class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are `reshape`, `size`, the relational operators, subscripted assignment, and subscripted reference.

You can define your own methods for the `single` class by placing the appropriately named method in an `@single` folder within a folder on your path.

Examples

```
a = magic(4);
b = single(a);

whos
  Name      Size         Bytes  Class

  a         4x4            128  double array
  b         4x4             64  single array
```

More About

- "Floating-Point Numbers"

See Also

`cast` | `double` | `typecast`

Introduced before R2006a

sinh

Hyperbolic sine of argument in radians

Syntax

```
Y = sinh(X)
```

Description

The `sinh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

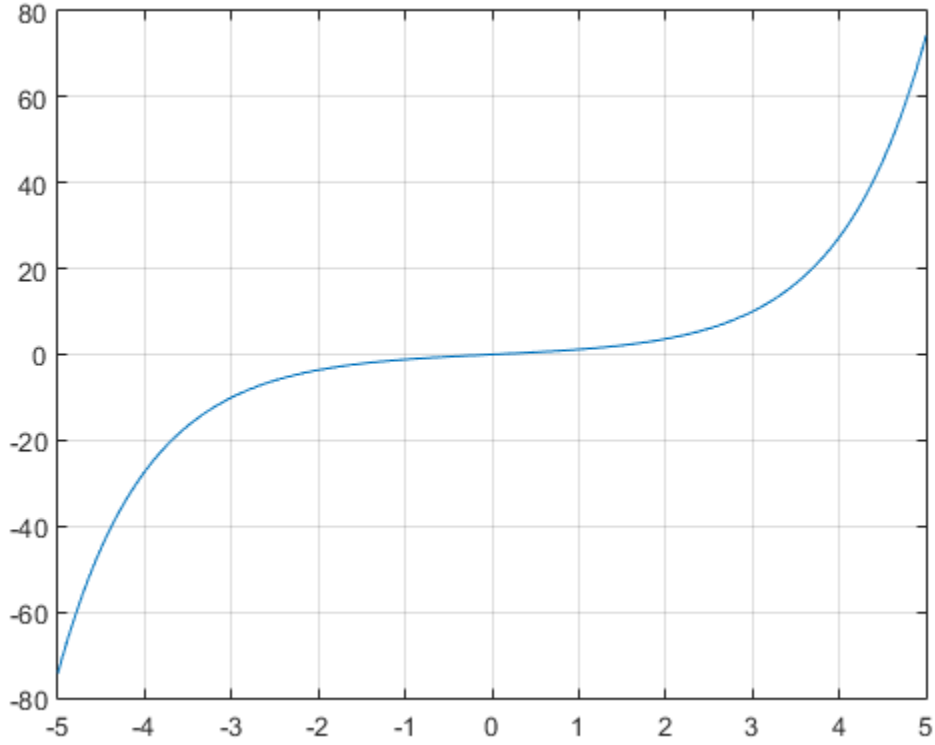
`Y = sinh(X)` returns the hyperbolic sine of the elements of `X`.

Examples

Graph of Hyperbolic Sine Function

Graph the hyperbolic sine over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,sinh(x)), grid on
```



More About

Hyperbolic Sine

The hyperbolic sine of z is

$$\sinh(z) = \frac{e^z - e^{-z}}{2}.$$

See Also

`sin` | `asinh` | `cosh`

Introduced before R2006a

size

Array dimensions

Syntax

```
d = size(X)
[m,n] = size(X)
m = size(X,dim)
[d1,d2,d3,...,dn] = size(X),
```

Description

`d = size(X)` returns the sizes of each dimension of array `X` in a vector, `d`, with `ndims(X)` elements.

- If `X` is a scalar, then `size(X)` returns the vector `[1 1]`. Scalars are regarded as a 1-by-1 arrays in MATLAB.
- If `X` is a table, `size(X)` returns a two-element row vector consisting of the number of rows and the number of variables in the table. Variables in the table can have multiple columns, but `size` only counts the variables and rows.

`[m,n] = size(X)` returns the size of matrix `X` in separate variables `m` and `n`.

`m = size(X,dim)` returns the size of the dimension of `X` specified by scalar `dim`.

`[d1,d2,d3,...,dn] = size(X),` for `n` > 1, returns the sizes of the dimensions of the array `X` in the variables d1,d2,d3,...,dn, provided the number of output arguments `n` equals `ndims(X)`. If `n` does not equal `ndims(X)`, the following exceptions hold:

`n < ndims(X)`	`di` equals the size of the `i`th dimension of `X` for `0<i<n`, but `dn` equals the product of the sizes of the remaining dimensions of `X`, that is, dimensions `n` through `ndims(X)`.
`n > ndims(X)`	`size` returns ones in the "extra" variables, that is, those corresponding to `ndims(X)+1` through `n`.

Note For a Java array, `size` returns the length of the Java array as the number of rows. The number of columns is always 1. For a Java array of arrays, the result describes only the top level array.

Examples

Size of Dimensions in 3-D Array

Find the size of `rand(2,3,4)`.

```
d = size(rand(2,3,4))
```

```
d =

     2     3     4
```

The size is output as a single vector.

Find the size of the second dimension of `rand(2,3,4)`.

```
m = size(rand(2,3,4),2)
```

```
m =

     3
```

The size is output as a scalar.

Assign the size of each dimension to a separate variable.

```
[m,n,p] = size(rand(2,3,4))
```

```
m =

     2


n =

     3


p =
```

```
     4
```

Size of Table

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

X = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
X =

               Age    Height    Weight     BloodPressure
               ---    ------    ------    ---------------
    Smith      38     71        176       124          93
    Johnson    43     69        163       109          77
    Williams   38     64        131       125          83
    Jones      40     67        133       117          75
    Brown      49     64        119       122          80
```

Find the size of the table.

```
d = size(X)
```

```
d =

     5     4
```

`size` counts four variables, even though the variable `BloodPressure` contains two columns.

Specify Different Number of Outputs than `ndims(X)`

Assign the size of each dimension to a separate variable where the number of outputs matches the number of dimensions.

```
X = ones(3,4,5);
[d1,d2,d3] = size(X)
```

```
d1 =
```

```
     3

d2 =

     4

d3 =

     5
```

There is one output for each dimension of `X`.

Specify fewer output variables than `ndims(X)`.

```
[d1,d2] = size(X)
d1 =

     3

d2 =

    20
```

The “extra” dimensions are collapsed into a single product.

Specify more output variables than `ndims(X)`.

```
[d1,d2,d3,d4,d5,d6] = size(X)
d1 =

     3

d2 =

     4

d3 =
```

```
     5

d4 =

     1

d5 =

     1

d6 =

     1
```

The “extra” variables all represent singleton dimensions.

See Also

`exist` | `length` | `ndims` | `numel` | `whos`

Introduced before R2006a

size

Class: containers.Map
Package: containers

Size of `containers.Map` object

Syntax

```
dim = size(mapObj,1)
dimVector = size(mapObj)
[dim1,dim2,...,dimN] = size(mapObj)
```

Description

`dim = size(mapObj,1)` returns a scalar numeric value that indicates the number of key-value pairs in `mapObj`. If you call `size` with a numeric second input argument other than `1`, the `size` method returns the scalar numeric value `1`.

`dimVector = size(mapObj)` returns a two-element vector `[k,1]`, where `k` is the number of key-value pairs in `mapObj`.

`[dim1,dim2,...,dimN] = size(mapObj)` returns `[k,1,...,1]`.

Input Arguments

`mapObj`

Object of class `containers.Map`.

Output Arguments

`dim`

Scalar numeric value that indicates the number of key-value pairs in `mapObj`.

`dimVector`

Two-element numeric vector `[k,1]`, where `k` is the number of key-value pairs in `mapObj`.

`[dim1,dim2,...,dimN]`

Numeric scalar values. Variable `dim1` equals `k`, where `k` is the number of key-value pairs in `mapObj`. All other outputs equal `1`.

Examples

Determine the Size of a Map

Construct a map and find the number of key-value pairs:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
mapObj = containers.Map(myKeys,myValues);
dim = size(mapObj,1)
```

This code returns a scalar numeric value:

```
dim =
     3
```

If you do not specify a second input argument,

```
dimVector = size(mapObj)
```

then the `size` method returns a vector:

```
dimVector =
     3     1
```

See Also

`isKey` | `keys` | `values` | `containers.Map` | `length`

size

Class: matlab.io.MatFile
Package: matlab.io

Array dimensions

Syntax

```
allDims = size(matObj,variable)
[dim1,...,dimN] = size(matObj,variable)
selectedDim = size(matObj,variable,dim)
```

Description

`allDims = size(matObj,variable)` returns the size of each dimension of the specified variable in the file corresponding to `matObj`. Output `allDims` is a `1`-by-`m` vector, where `m = ndims(variable)`.

`[dim1,...,dimN] = size(matObj,variable)` returns the sizes of each dimension in separate output variables `dim1,...,dimN`.

`selectedDim = size(matObj,variable,dim)` returns the size of the specified dimension.

Tips

- Do not call `size` with the syntax `size(matObj.variable)`. This syntax loads the entire contents of the variable into memory. For very large variables, this load operation results in `Out of Memory` errors.

Input Arguments

`matObj`

Object created by the `matfile` function.

`variable`

String enclosed in single quotation marks that specifies the name of a variable in the MAT-file corresponding to `matObj`.

`dim`

Nonzero positive scalar integer that specifies a dimension of the variable.

Output Arguments

`allDims`

1-by-`m` vector of sizes of the dimensions of the specified variable, where `m = ndims(variable)`.

`dim1,...,dimN`

Scalar numeric values, where `dimK` contains the size of the `K`th dimension of `variable`:

- If `N < ndims(variable)`, then `dimN`, equals the product of the sizes of dimensions `N` through `ndims(variable)`.
- If `N > ndims(variable)`, the `size` method returns ones in the output variables corresponding to dimensions `ndims(variable)+1` through `N`.

`selectedDim`

Scalar numeric value that contains the size of the selected dimension for the specified variable.

Examples

Find the size of the matrix `topo` in `topography.mat` without loading any data:

```
matObj = matfile('topography.mat');
[nrows,ncols] = size(matObj,'topo');
```

Determine the dimensions of a variable, and process one part of the variable at a time. In this case, calculate and store the average of each column of variable `stocks` in the example file `stocks.mat`:

```
filename = 'stocks.mat';
matObj = matfile(filename);
[nrows, ncols] = size(matObj,'stocks');

avgs = zeros(1,ncols);
for idx = 1:ncols
    avgs(idx) = mean(matObj.stocks(:,idx));
end
```

Create a three-dimensional array, and call the `size` method with different numbers of output arguments:

```
matObj = matfile('temp.mat','Writable',true);
matObj.X = rand(2,3,4);

d = size(matObj,'X')
d2 = size(matObj,'X',2)
[m,n] = size(matObj,'X')
[m1,m2,m3,m4] = size(matObj,'X')
```

This code returns

```
d =
     2     3     4

d2 =
     3

m =
     2
n =
    12

m1 =
     2
m2 =
     3
m3 =
     4
m4 =
     1
```

See Also

`whos` | `matfile`

size (serial)

Size of serial port object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in the serial port object, `obj`.

`[m,n] = size(obj)` returns the number of rows, `m` and columns, `n` in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also

`length`

Introduced before R2006a

size

Class: TriRep

(Will be removed) Size of triangulation matrix

Note: `size(TriRep)` will be removed in a future release. Use `size(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
size(TR)
```

Description

`size(TR)` provides size information for a triangulation matrix. The matrix is of size `mtri`-by-`nv`, where `mtri` is the number of simplices and `nv` is the number of vertices per simplex (triangle/tetrahedron, etc).

Input Arguments

`TR`	Triangulation matrix

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

See Also

`size` | `triangulation` | `delaunayTriangulation`

size (tscollection)

Size of `tscollection` object

Syntax

```
size(tsc)
```

Description

`size(tsc)` returns `[n m]`, where `n` is the length of the time vector and `m` is the number of `tscollection` members.

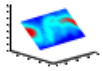
See Also

`length (tscollection)` | `isempty (tscollection)` | `tscollection`

Introduced before R2006a

slice

Volumetric slice plot



Syntax

```
slice(V,sx,sy,sz)
slice(X,Y,Z,V,sx,sy,sz)
slice(V,XI,YI,ZI)
slice(X,Y,Z,V,XI,YI,ZI)
slice(...,'method')
slice(axes_handle,...)
h = slice(...)
```

Description

`slice` displays orthogonal slice planes through volumetric data.

`slice(V,sx,sy,sz)` draws slices along the *x*, *y*, *z* directions in the volume `V` at the points in the vectors `sx`, `sy`, and `sz`. `V` is an *m*-by-*n*-by-*p* volume array containing data values at the default location `X = 1:n`, `Y = 1:m`, `Z = 1:p`. Each element in the vectors `sx`, `sy`, and `sz` defines a slice plane in the *x*-, *y*-, or *z*-axis direction.

`slice(X,Y,Z,V,sx,sy,sz)` draws slices of the volume `V`. `X`, `Y`, and `Z` are three-dimensional arrays specifying the coordinates for `V`. `X`, `Y`, and `Z` must be monotonic and orthogonally spaced (as if produced by the function `meshgrid`). The color at each point is determined by 3-D interpolation into the volume `V`.

`slice(V,XI,YI,ZI)` draws data in the volume `V` for the slices defined by `XI`, `YI`, and `ZI`. `XI`, `YI`, and `ZI` are matrices that define a surface, and the volume is evaluated at the surface points. `XI`, `YI`, and `ZI` must all be the same size.

`slice(X,Y,Z,V,XI,YI,ZI)` draws slices through the volume `V` along the surface defined by the arrays `XI`, `YI`, `ZI`.

`slice(...,'method')` specifies the interpolation method. `'method'` is `'linear'`, `'cubic'`, or `'nearest'`.

- `linear` specifies trilinear interpolation (the default).
- `cubic` specifies tricubic interpolation.
- `nearest` specifies nearest-neighbor interpolation.

`slice(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes object (`gca`). The axes `clim` property is set to span the finite values of `V`.

`h = slice(...)` returns a vector of handles to surface graphics objects.

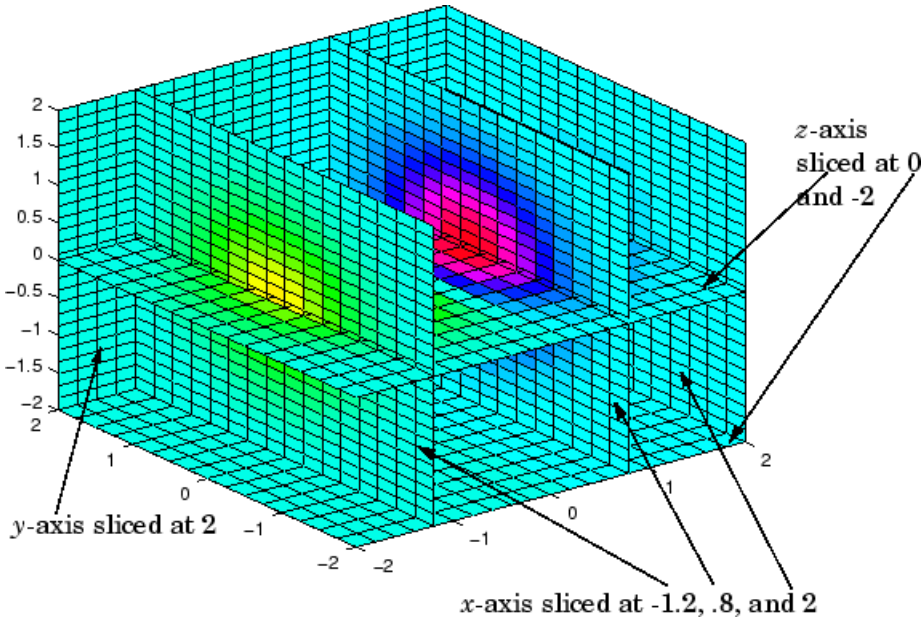
Examples

Visualize the function

$$v = xe^{\left(-x^2-y^2-z^2\right)}$$

over the range $-2 \le x \le 2$, $-2 \le y \le 2$, $-2 \le z \le 2$:

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
xslice = [-1.2,.8,2];
yslice = 2;
zslice = [-2,0];
slice(x,y,z,v,xslice,yslice,zslice)
colormap hsv
```



Slicing At Arbitrary Angles

You can also create slices that are oriented in arbitrary planes. To do this,

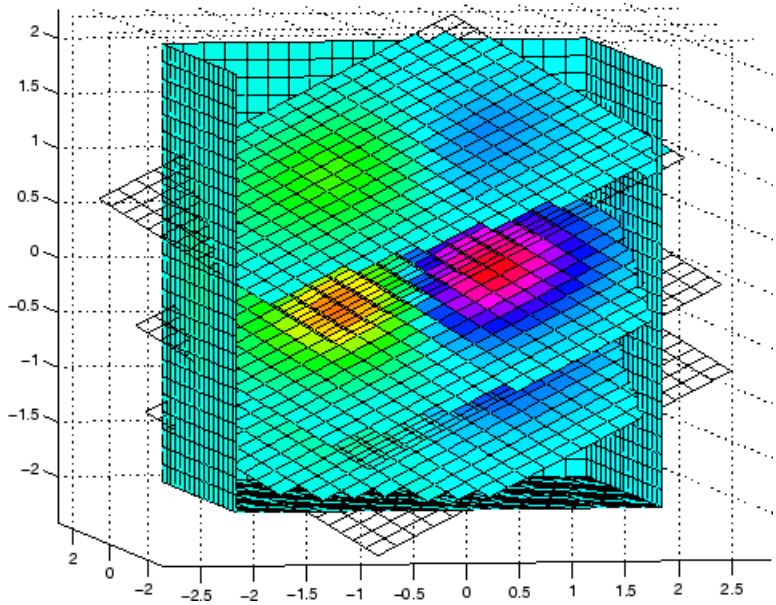
- Create a slice surface in the domain of the volume (`surf`, `linspace`).
- Orient this surface with respect to the axes (`rotate`).
- Get the `XData`, `YData`, and `ZData` of the surface.
- Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a `for` loop "passes" the plane through the volume along the *z*-axis.

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
figure
colormap hsv
for k = -2:.05:2
   hsp = surf(linspace(-2,2,20),linspace(-2,2,20),...
      zeros(20) + k);
```

```
    rotate(hsp,[1,-1,1],30)
    xd = hsp.XData;
    yd = hsp.YData;
    zd = hsp.ZData;
    delete(hsp)

    slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries
    hold on
    slice(x,y,z,v,xd,yd,zd)
    hold off
    view(-5,10)
    axis([-2.5 2.5 -2 2 -2 4])
    drawnow
end
```

The following picture illustrates three positions of the same slice surface as it passes through the volume.

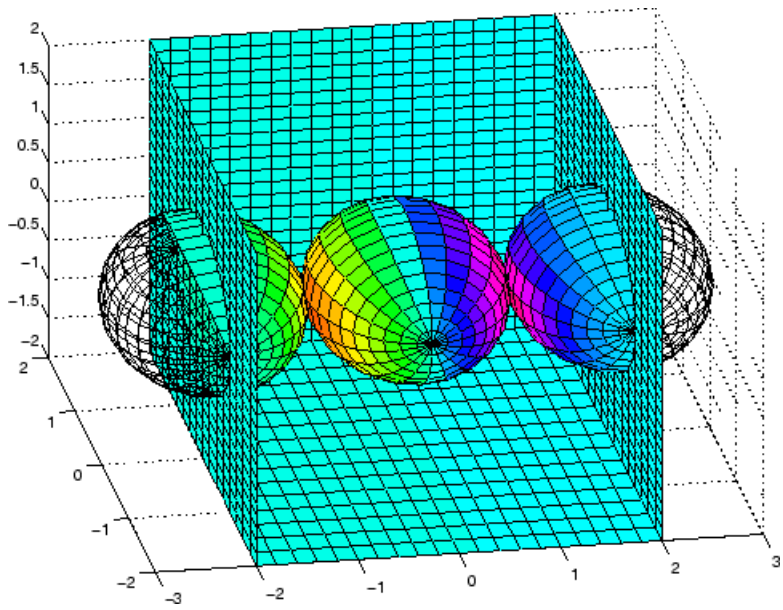


Slicing with a Nonplanar Surface

You can slice the volume with any surface. This example probes the volume created in the previous example by passing a spherical slice surface through the volume.

```
[xsp,ysp,zsp] = sphere;
slice(x,y,z,v,[-2,2],2,-2)
colormap hsv
for i = -3:.2:3
   hsp = surface(xsp+i,ysp,zsp);
   rotate(hsp,[1 0 0],90)
   xd = hsp.XData;
   yd = hsp.YData;
   zd = hsp.ZData;
   delete(hsp)
   hold on
   hslicer = slice(x,y,z,v,xd,yd,zd);
   axis tight
   xlim([-3,3])
   view(-10,35)
   drawnow
   delete(hslicer)
   hold off
end
```

The following picture illustrates three positions of the spherical slice surface as it passes through the volume.



More About

Tips

The color drawn at each point is determined by interpolation into the volume `V`.

- Exploring Volumes with Slice Planes

See Also

`interp3` | `meshgrid`

Introduced before R2006a

smooth3

Smooth 3-D data

Syntax

Description

`W = smooth3(V)` smooths the input data `V` and returns the smoothed data in `W`.

`W = smooth3(V,'filter')` *`filter`* determines the convolution kernel and can be the strings

- `'gaussian'`
- `'box'` (default)

`W = smooth3(V,'filter',size)` sets the size of the convolution kernel (default is `[3 3 3]`). If `size` is scalar, then `size` is interpreted as [`size`, `size`, `size`].

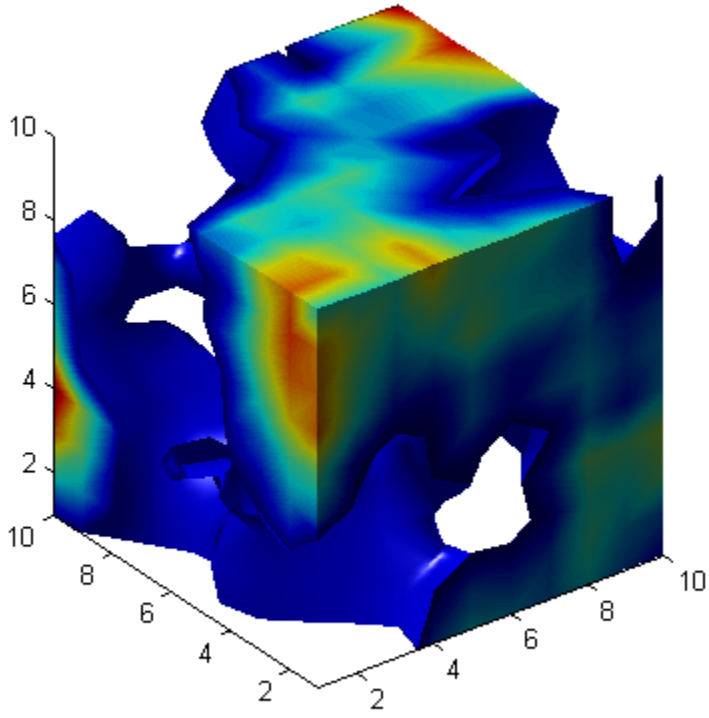
`W = smooth3(V,'filter',size,sd)` sets an attribute of the convolution kernel. When *`filter`* is `gaussian`, `sd` is the standard deviation (default is .65).

Examples

This example smooths some random 3-D data and then creates an isosurface with end caps.

```
rng(9,'twister')
data = rand(10,10,10);
data = smooth3(data,'box',5);
patch(isocaps(data,.5),...
   'FaceColor','interp','EdgeColor','none');
p1 = patch(isosurface(data,.5),...
   'FaceColor','blue','EdgeColor','none');
isonormals(data,p1)
view(3);
axis vis3d tight
camlight left;
```

```
colormap jet
lighting gouraud
```



More About

- Displaying an Isosurface

See Also

`isocaps` | `isonormals` | `isosurface` | `patch`

Introduced before R2006a

snapnow

Force snapshot of image for inclusion in published document

Syntax

```
snapnow
```

Description

The `snapnow` command forces a snapshot of the image or plot that the code has most recently generated for presentation in a published document. The output appears in the published document at the end of the cell that contains the `snapnow` command. When used outside the context of publishing a file, `snapnow` has the same behavior as `drawnow`. That is, if you run a file that contains the `snapnow` command, the MATLAB software interprets it as though it were a `drawnow` command.

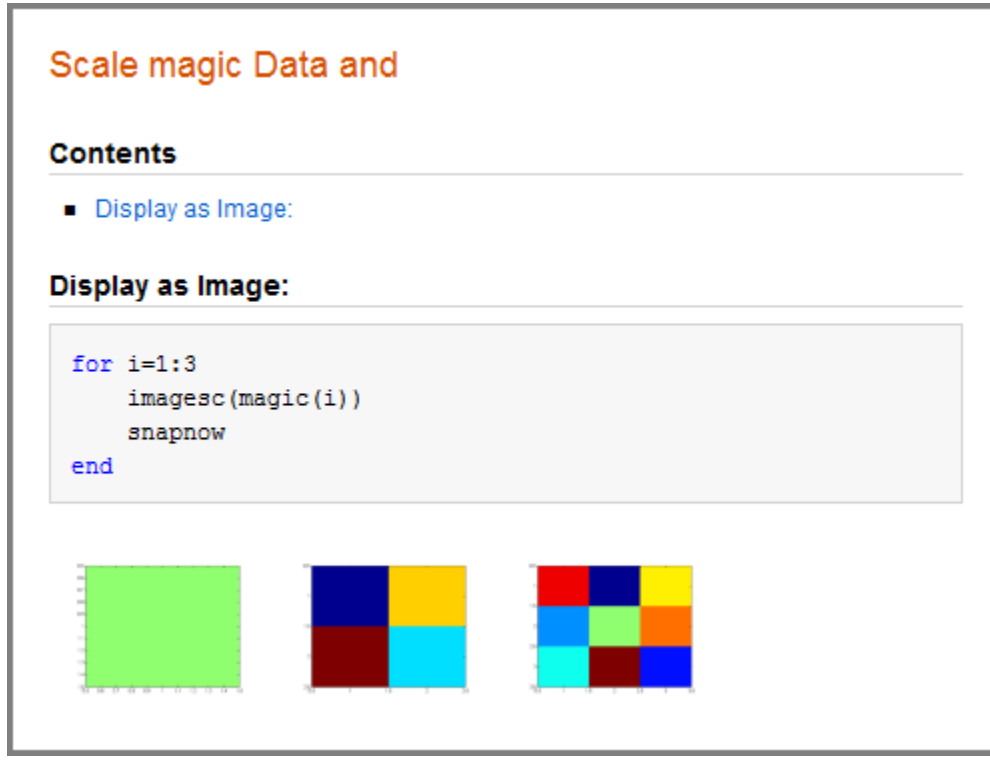
Examples

This example demonstrates the difference between publishing code that contains the `snapnow` command and running that code. The first image shows the results of publishing the code and the second image shows the results of running the code.

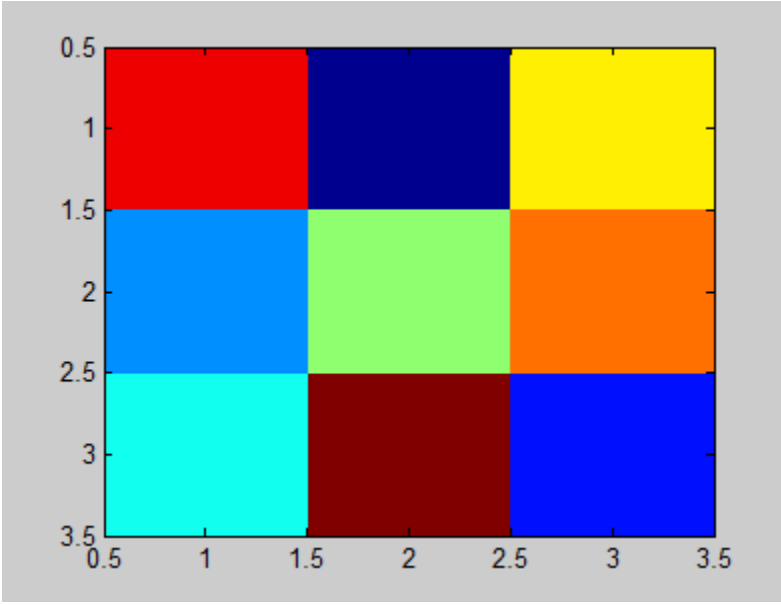
Suppose you have a file that contains the following code:

```
%% Scale magic Data and
%% Display as Image:

for i=1:3
    imagesc(magic(i))
    snapnow
end
```

When you publish the code to HTML, the published document contains a title, a table of contents, the commented text, the code, and each of the three images produced by the `for` loop. (In the published document shown, the size of the images have been reduced.)



When you run the code, a single Figure window opens and MATLAB updates the image within this window as it evaluates each iteration of the `for` loop. Each successive image replaces the one that preceded it, so that the Figure window appears as follows when the code evaluation completes.



More About

- “Image Snapshot”

See Also

`drawnow`

sort

Sort array elements

Syntax

```
B = sort(A)
B = sort(A,dim)
B = sort( ___ ,mode)
[B,I] = sort( ___ )
```

Description

`B = sort(A)` sorts the elements of `A` in ascending order along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `sort(A)` sorts the vector elements.
- If `A` is a matrix, then `sort(A)` treats the columns of `A` as vectors and sorts each column.
- If `A` is a multidimensional array, then `sort(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors.

`B = sort(A,dim)` returns the sorted elements of `A` along dimension `dim`. For example, if `A` is a matrix, then `sort(A,2)` sorts the elements in each row.

`B = sort( ___ ,mode)` returns sorted elements of `A` in the order specified by `mode` using any of the previous syntaxes. The single string, `'ascend'`, indicates ascending order (default) and `'descend'` indicates descending order.

`[B,I] = sort( ___ )` additionally returns a collection of index vectors for any of the previous syntaxes. `I` is the same size as `A` and describes the arrangement of the elements of `A` into `B` along the sorted dimension. For example, if `A` is a numeric vector, `B = A(I)`.

Examples

Sort Vector in Ascending Order

Create a row vector and sort its elements in ascending order.

```
A = [9 0 -7 5 3 8 -10 4 2];
B = sort(A)

B =

   -10    -7     0     2     3     4     5     8     9
```

Sort Matrix Rows in Ascending Order

Create a matrix and sort each of its rows in ascending order.

```
A = [3 6 5; 7 -2 4; 1 0 -9]

A =

     3     6     5
     7    -2     4
     1     0    -9

B = sort(A,2)

B =

     3     5     6
    -2     4     7
    -9     0     1
```

Sort Matrix Columns in Descending Order

Create a matrix and sort its columns in descending order.

```
A = [10 -12 4 8; 6 -9 8 0; 2 3 11 -2; 1 1 9 3]

A =

    10   -12     4     8
     6    -9     8     0
     2     3    11    -2
     1     1     9     3
```

```
B = sort(A,'descend')
```

```
B =

    10     3    11     8
     6     1     9     3
     2    -9     8     0
     1   -12     4    -2
```

Sort and Index `datetime` Array

Create an array of `datetime` values and sort them in ascending order, that is, from the earliest to the latest calendar date.

```
ds = {'2012-12-22';'2063-04-05';'1992-01-12'};
A = datetime(ds,'Format','yyyy-MM-dd')
```

```
A =

   2012-12-22
   2063-04-05
   1992-01-12
```

```
[B,I] = sort(A)
```

```
B =

   1992-01-12
   2012-12-22
   2063-04-05


I =

     3
     1
     2
```

`B` lists the sorted dates and `I` contains the corresponding indices of `A`.

Access the sorted elements from the original array directly by using the index array `I`.

```
A(I)
```

```
ans =

   1992-01-12
   2012-12-22
   2063-04-05
```

Sort 3-D Array

Create a 2-by-2-by-2 array and sort its elements in ascending order along the third dimension.

```
A(:,:,1) = [2 3; 1 6];
A(:,:,2) = [-1 9; 0 12];
A
```

```
A(:,:,1) =

     2     3
     1     6


A(:,:,2) =

    -1     9
     0    12
```

```
B = sort(A,3)
```

```
B(:,:,1) =

    -1     3
     0     6


B(:,:,2) =
```

```
     2     9
     1    12
```

Use `A(:)`, the column representation of `A`, to sort all of the elements of `A`.

```
B = sort(A(:))
```

```
B =

    -1
     0
     1
     2
     3
     6
     9
    12
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If `A` is a scalar, then `sort(A)` returns `A`.
- If `A` is an empty 0-by-0 matrix, then `sort(A)` returns an empty 0-by-0 matrix.
- If `A` contains `NaN` values or undefined `categorical` or `datetime` elements, then `sort(A)` places them on the high end of the sort (as if they were large numbers).
- If `A` is complex, then `sort(A)` sorts the elements by magnitude. If more than one element has equal magnitude, the elements are sorted by phase angle on the interval [−π, π].
- If `A` is a string, then `sort(A)` sorts according to the ASCII dictionary order. The sort is case sensitive with uppercase letters appearing in the output before the lowercase letters.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `cell` | `categorical` | `datetime` | `duration`

Complex Number Support: Yes

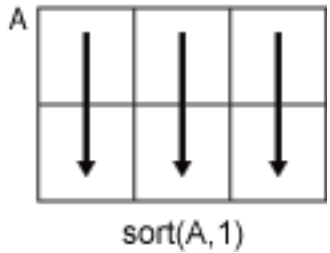
`dim` — Dimension to operate along

positive integer scalar

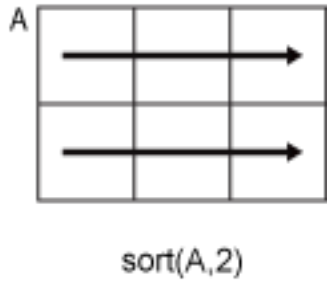
Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `sort(A,1)` sorts the elements in the columns of `A`.



- `sort(A,2)` sorts the elements in the rows of `A`.



`sort` returns `A` if `dim` is greater than `ndims(A)`. If `A` is a cell array, then `dim` is not supported.

`mode` — Sorting mode

`'ascend'` (default) | `'descend'`

Sorting mode, specified as `'ascend'` or `'descend'`. If `A` is a cell array, then `mode` is not supported.

Example: `sort(A,2,'descend')` sorts the rows of `A` in descending order.

Data Types: `char`

Output Arguments

`B` — Sorted array

vector | matrix | multidimensional array

Sorted array, returned as a vector, matrix, or multidimensional array the same size and type as `A`. The ordering of the elements in `B` preserves the order of any equal elements in `A`.

`I` — Sort index

vector | matrix | multidimensional array

Sort index, returned as a vector, matrix, or multidimensional array the same size as `A`. The index vectors are oriented along the same dimension that `sort` operates on. For example, if `A` is a 2-by-3 matrix, then `[B,I] = sort(A,2)` sorts the elements in each row of `A`. `I` is a collection of 1-by-3 row index vectors describing the rearrangement. If you do not specify `dim`, then `[B,I] = sort(A)` sorts the columns of `A`, and `I` is a collection of 2-by-1 column index vectors.

More About

Tips

- The `sortrows` function provides additional flexibility for subsorting over multiple columns of nonvector input arrays.
- The `sort` function and the relational operators use different orderings for complex numbers. For more information, see “Relational Operations”.
- “Shifting and Sorting Matrices”

See Also

`issorted` | `max` | `mean` | `median` | `min` | `sortrows` | `unique`

sortrows

Sort array rows

Syntax

```
B = sortrows(A)
B = sortrows(A,column)
[B,index] = sortrows( ___ )

tblB = sortrows(tblA)
tblB = sortrows(tblA,'RowNames')
tblB = sortrows(tblA,vars)

tblB = sortrows(tblA,mode)
tblB = sortrows(tblA,'RowNames',mode)
tblB = sortrows(tblA,vars,mode)

[tblB,index] = sortrows(tblA, ___ )
```

Description

`B = sortrows(A)` sorts the rows of `A` in ascending order. For strings, this is the familiar dictionary sort.

`B = sortrows(A,column)` sorts matrix `A` based on the columns specified in the vector, `column`. This input is used to perform multiple column sorts in succession.

`[B,index] = sortrows( ___ )` also returns an index vector using any of the previous syntaxes. The index vector satisfies `B = A(index,:)`.

`tblB = sortrows(tblA)` sorts the rows of table `tblA` in ascending order by the first variable, then by the second variable, and so on.

`tblB = sortrows(tblA,'RowNames')` sorts by the row names.

`tblB = sortrows(tblA,vars)` sorts by the variables specified by `vars`.

`tblB = sortrows(tblA,mode)` and `tblB = sortrows(tblA,'RowNames',mode)` sorts `tblA` in the order specified by `mode`. The single string, `'ascend'`, indicates ascending order (default) and `'descend'` indicates descending order.

`tblB = sortrows(tblA,vars,mode)` uses `mode` to specify the sort order. `mode` can be a single string or a cell array of strings containing `'ascend'` for ascending order (default) or `'descend'` for descending order.

- When `mode` is a single string, `sortrows` sorts in the specified direction for all variables in `vars`.
- When `mode` is a cell array of strings, `sortrows` sorts in the specified direction for each variable in `vars`.

`[tblB,index] = sortrows(tblA, ___ )` also returns an index vector, `index`, such that `tblB = tblA(index,:)`.

Examples

Sort Rows of Matrix

Start with an arbitrary matrix, `A`.

```
A = floor(gallery('uniformdata',[6 7],0)*100);
A(1:4,1) = 95;  A(5:6,1) = 76;  A(2:4,2) = 7;  A(3,3) = 73
```

```
A =
    95    45    92    41    13     1    84
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

Sort the rows of `A`.

```
B = sortrows(A)
```

```
B =
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
```

```
    95     7    73    89    20    74    52
    95    45    92    41    13     1    84
```

When called with only a single input argument, `sortrows` bases the sort on the first column of the matrix. For any rows that have equal elements in a particular column, (e.g., `A(1:4,1)` for this matrix), sorting is based on the column immediately to the right, (`A(1:4,2)` in this case).

When called with two input arguments, `sortrows` bases the sort entirely on the column specified in the second argument.

Sort the rows of `A` based on the values in the second column.

```
C = sortrows(A,2)
```

```
C =
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

Rows that have equal elements in the specified column, (e.g., `A(2:4,:)`, if sorting matrix `A` by column 2) remain in their original order.

Specify two columns to sort by: columns 1 and 7.

```
D = sortrows(A,[1 7])
```

```
D =
    76    79    91     0    19    41     1
    76    61    93    81    27    46    83
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
```

`sortrows` sorts by column 1 first, and then for any rows with equal values in column 1, sorts by column 7.

Sort the rows in descending order using the values in column 4.

```
[E,index] = sortrows(A, -4)
```

```
E =

    95     7    73    89    20    74    52
    76    61    93    81    27    46    83
    95    45    92    41    13     1    84
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    76    79    91     0    19    41     1


index =

     2
     5
     1
     4
     3
     6
```

The index vector, `index`, describes the rearrangement of the rows, such that `E = A(index,:)`.

Sort Rows of Cell Array

Create a 6-by-2 cell array of strings.

```
A = {'Germany' 'Lukas'; 'USA' 'William'; 'USA' 'Andrew'; ...
'Germany' 'Andreas'; 'USA' 'Olivia'; 'Germany' 'Julia'}
```

```
A =

    'Germany'    'Lukas'
    'USA'        'William'
    'USA'        'Andrew'
    'Germany'    'Andreas'
    'USA'        'Olivia'
    'Germany'    'Julia'
```

The result is a list of countries and names.

Sort the rows of `A`.

```
B = sortrows(A)
```

```
B =
```

```
    'Germany'    'Andreas'
    'Germany'    'Julia'
    'Germany'    'Lukas'
    'USA'        'Andrew'
    'USA'        'Olivia'
    'USA'        'William'
```

The result is an alphabetized list sorted by both country and name.

Sort the names in the second column in descending order.

```
[C,index] = sortrows(A,[1 -2])
```

```
C =

    'Germany'    'Lukas'
    'Germany'    'Julia'
    'Germany'    'Andreas'
    'USA'        'William'
    'USA'        'Olivia'
    'USA'        'Andrew'


index =

     1
     6
     4
     2
     5
     3
```

The index vector, `index`, describes the rearrangement of the rows, such that `C = A(index,:)`.

Sort Rows of Table

Sort the rows of a table by the variable values in ascending order.

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
```

```
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA =

               Age    Height    Weight     BloodPressure
               ---    ------    ------    ---------------
    Smith      38     71        176       124          93
    Johnson    43     69        163       109          77
    Williams   38     64        131       125          83
    Jones      40     67        133       117          75
    Brown      49     64        119       122          80
```

Sort the rows of the table.

```
tblB = sortrows(tblA)
```

```
tblB =

               Age    Height    Weight     BloodPressure
               ---    ------    ------    ---------------
    Williams   38     64        131       125          83
    Smith      38     71        176       124          93
    Jones      40     67        133       117          75
    Johnson    43     69        163       109          77
    Brown      49     64        119       122          80
```

The `sortrows` function sorts the rows in ascending order first by the variable `Age`, and then it sorts by the variable `Height`.

Sort Rows of Table by Row Names

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA =
```

```
              Age    Height    Weight     BloodPressure
              ---    ------    ------    ---------------
    Smith     38     71        176       124          93
    Johnson   43     69        163       109          77
    Williams  38     64        131       125          83
    Jones     40     67        133       117          75
    Brown     49     64        119       122          80
```

Sort the rows of the table by the row names and return an index vector, such that `tblB = tblA(index,:)`.

```
[tblB,index] = sortrows(tblA,'RowNames')
```

```
tblB =

              Age    Height    Weight     BloodPressure
              ---    ------    ------    ---------------
    Brown     49     64        119       122          80
    Johnson   43     69        163       109          77
    Jones     40     67        133       117          75
    Smith     38     71        176       124          93
    Williams  38     64        131       125          83


index =

     5
     2
     4
     1
     3
```

The `sortrows` function sorts the rows in ascending order by the row names.

Sort Rows of Table by Variables

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA =

               Age    Height    Weight     BloodPressure
               ---    ------    ------    ---------------
    Smith      38     71        176       124           93
    Johnson    43     69        163       109           77
    Williams   38     64        131       125           83
    Jones      40     67        133       117           75
    Brown      49     64        119       122           80
```

Sort the rows of the table in ascending order by `Height`, and then sort in descending order by `Weight`. Also, return an index vector, such that `tblB = tblA(index,:)`.

```
[tblB,index] = sortrows(tblA,{'Height','Weight'},{'ascend','descend'})
```

```
tblB =

               Age    Height    Weight     BloodPressure
               ---    ------    ------    ---------------
    Williams   38     64        131       125           83
    Brown      49     64        119       122           80
    Jones      40     67        133       117           75
    Johnson    43     69        163       109           77
    Smith      38     71        176       124           93


index =

     3
     5
     4
     2
     1
```

Input Arguments

A — Input array

column vector | matrix

Input array, specified as a column vector or matrix. The data type of `A` can be numeric, `logical`, `char`, `cell`, categorical, datetime, or duration. If `A` contains `NaN` values or undefined categorical or datetime elements, `sortrows(A)` places them on the high end of the sort (as if they are large numbers).

When A is complex, sortrows sorts the elements by magnitude, and, where magnitudes are equal, further sorts by phase angle on the interval [−π, π].
Complex Number Support: Yes

column — Column sorting vector
vector of integers

Column sorting vector, specified as a vector of integers. Each integer value indicates a column to sort by. The sign of the integer indicates ascending (positive) or descending (negative) sort order.

Example: sortrows(A,[2 -3]) sorts the rows of A first in ascending order for the second column, and then it sorts by descending order for the third column.

tblA — Input table
table

Input table, specified as a table. Each variable in tblA must be a valid input to sort or sortrows.

Data Types: table

'RowNames' — Row sort input
string

Row sort input, specified as the string 'RowNames'. Specify the 'RowNames' option to sort a table by row names rather than by variables. If tblA.Properties.RowNames is empty, sortrows(tblA,'RowNames') returns tblA.

vars — Sorting variables
integer | vector of integers | variable name | cell array of variable names | logical vector

Sorting variables, specified as an integer, a vector of integers, a variable name, a cell array of variable names, or a logical vector. vars indicates the table variables to sort by. You also can use the mode input to indicate ascending or descending order for each sorting variable.

If an element of vars is a positive integer, sortrows sorts the corresponding variable in tblA in ascending order. If an element of vars is a negative integer, sortrows sorts the corresponding variable in tblA in descending order. If you provide the mode input argument, MATLAB ignores the sign of the integers.

Example: sortrows(tblA,{'Height','Weight'}) sorts the rows of tblA in ascending order, first by the variable Height, and then it sorts by the variable Weight.

Example: `sortrows(tblA,[1 4],{'descend' 'ascend'})` sorts the first variable of `tblA` in descending order, then it sorts the fourth variable in ascending order.

`mode` — Sorting mode

single string | cell array of strings

Sorting mode, specified as a single string or cell array of strings composed of the options `'ascend'` (default), or `'descend'`. If `mode` is a cell array of strings, the number of required entries depends on whether you are sorting by variables or by row names.

- If `tblA` is being sorted by variables, the cell array must have an entry for each variable.
- If `tblA` is being sorted by row names, the cell array must have one entry.

Data Types: `char | cell`

Output Arguments

`B` — Sorted array

array

Sorted array, returned as an array of the same size and class as `A`.

`tblB` — Sorted table

table

Sorted table, returned as a table with the same variables as `tblA`.

`index` — Sort index

index vector

Sort index, returned as an index vector. The sort index describes the rearrangement of elements or rows in the input.

See Also

`issorted | sort`

Introduced before R2006a

sound

Convert matrix of signal data to sound

Syntax

```
sound(y)
sound(y,Fs)
sound(y,Fs,nBits)
```

Description

`sound(y)` sends audio signal `y` to the speaker at the default sample rate of 8192 hertz.

`sound(y,Fs)` sends audio signal `y` to the speaker at sample rate `Fs`.

`sound(y,Fs,nBits)` uses `nBits` bits per sample for audio signal `y`.

Examples

Play Sample Data at Default Sample Rate

Load the example file `gong.mat`, which contains sample data `y` and rate `Fs`, and listen to the audio.

```
load gong.mat;
sound(y);
```

Play Sample Data at Specific Sample Rate

Play an excerpt from Handel's “Hallelujah Chorus” at twice the recorded sample rate.

```
load handel.mat;
sound(y, 2*Fs);
```

Play Sample Data with Specific Bit Depth

```
load handel.mat;
nBits = 16;
```

```
sound(y,Fs,nBits);
```

MATLAB plays the audio with a bit depth of 16 bits per sample, if this is supported on your system.

Input Arguments

y — Audio data

column vector | m-by-2 matrix

Audio data, specified as an m-by-1 column vector for single-channel (mono) audio, or an m-by-2 matrix for stereo playback, where m is the number of audio samples. If y is an m-by-2 matrix, then the first column corresponds to the left channel, and the second column corresponds to the right channel. Stereo playback is available only if your system supports it.

Data Types: double

Fs — Sample rate

8192 (default) | positive number

Sample rate, in hertz, of audio data y, specified as a positive number between 80 and 1000000.

Data Types: single | double

nBits — Bit depth of sample values

8 (default) | 16 | 24

Bit depth of the sample values, specified as an integer. Valid values depend on the audio hardware installed. Most platforms support bit depths of 8 bits or 16 bits.

More About

Tips

- The sound function supports sound devices on all Windows and most UNIX platforms.
- Most sound cards support sample rates between 5 and 48 kilohertz. Specifying a sample rate outside this range might produce unexpected results.

- “Characteristics of Audio Files”
- “Play Audio”

See Also

`audioplayer` | `audioread` | `audiowrite` | `soundsc`

Introduced before R2006a

soundsc

Scale data and play as sound

Syntax

```
soundsc(y)
soundsc(y,Fs)
soundsc(y,Fs,nBits)

soundsc( ___ ,yRange)
```

Description

`soundsc(y)` scales the values of audio signal `y` to fit in the range from –1.0 to 1.0, and then sends the data to the speaker at the default sample rate of 8192 hertz. By first scaling the data, `soundsc` plays the audio as loudly as possible without clipping. The mean of the dynamic range of the data is set to zero.

`soundsc(y,Fs)` sends audio signal `y` to the speaker at sample rate `Fs`.

`soundsc(y,Fs,nBits)` uses `nBits` bits per sample for audio signal `y`.

`soundsc( ___ ,yRange)`, where `yRange` is a vector of the form `[low,high]`, linearly scales the values in `y` between `low` and `high` to the full sound range `[-1.0,1.0]`. Values outside `[low,high]` scale beyond `[-1.0,1.0]`. You can use `yRange` with any of the input arguments in the previous syntaxes.

Examples

Play Sample Data at Default Sample Rate

Load the example file `gong.mat`, which contains sample data `y` and rate `Fs`, and listen to the audio.

```
load gong.mat;
```

```
soundsc(y);
```

Play Sample Data at Specific Sample Rate

Play an excerpt from Handel's "Hallelujah Chorus" at twice the recorded sample rate.

```
load handel.mat;
soundsc(y, 2*Fs);
```

Play Sample Data with Specific Bit Depth

```
load handel.mat;
nBits = 16;
soundsc(y,Fs,nBits);
```

MATLAB plays the scaled audio with a bit depth of 16 bits per sample.

Scale Selected Audio Data

```
load handel.mat;
yRange = [-0.7,0.7];
soundsc(y,yRange);
```

Input Arguments

y — Audio data

column vector | m-by-2 matrix

Audio data, specified as an `m`-by-1 column vector for single-channel (mono) audio, or an `m`-by-2 matrix for stereo playback, where `m` is the number of audio samples. If `y` is an `m`-by-2 matrix, then the first column corresponds to the left channel, and the second column corresponds to the right channel. Stereo playback is available only if your system supports it.

Data Types: `double`

Fs — Sample rate

8192 (default) | positive number

Sample rate, in hertz, of audio data `y`, specified as a positive number between 80 and 1000000.

Data Types: `single` | `double`

`nBits` — Bit depth of sample values

8 (default) | 16 | 24

Bit depth of the sample values, specified as an integer. Valid values depend on the audio hardware installed. Most platforms support bit depths of 8 bits or 16 bits.

`yRange` — Range of audio data to scale

`[-max(abs(y)),max(abs(y))]` (default) | two-element vector

Range of audio data to scale, specified as a two-element vector of the form `[low,high]`, where `low` and `high` are the lower and upper limits of the range. Values in `y` that are scaled beyond `[-1.0, 1.0]` are clipped when played back on a sound device.

Example: `[-0.8,0.8]`

Data Types: `double`

More About

Tips

- The `sound` function supports sound devices on all Windows and most UNIX platforms.
- Most sound cards support sample rates between 5 and 48 kilohertz. Specifying a sample rate outside this range might produce unexpected results.

See Also

`audioplayer` | `audioread` | `audiowrite` | `sound`

Introduced before R2006a

spalloc

Allocate space for sparse matrix

Syntax

```
S = spalloc(m,n,nzmax)
```

Description

`S = spalloc(m,n,nzmax)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nzmax` nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nzmax)` is shorthand for

```
sparse([],[],[],m,n,nzmax)
```

Examples

To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

```
S = spalloc(n,n,3*n);
for j = 1:n
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';end
```

Introduced before R2006a

sparse

Create sparse matrix

The `sparse` function generates matrices in the MATLAB sparse storage organization.

Syntax

```
S = sparse(A)

S = sparse(m,n)

S = sparse(i,j,v)
S = sparse(i,j,v,m,n)
S = sparse(i,j,v,m,n,nz)
```

Description

`S = sparse(A)` converts a full matrix into sparse form by squeezing out any zero elements. If a matrix contains many zeros, converting the matrix to sparse storage saves memory.

`S = sparse(m,n)` generates an `m`-by-`n` all zero sparse matrix.

`S = sparse(i,j,v)` generates a sparse matrix `S` from the triplets `i`, `j`, and `v` such that `S(i(k),j(k)) = v(k)`. The `max(i)`-by-`max(j)` output matrix has space allotted for `length(v)` nonzero elements. `sparse` adds together elements in `v` that have duplicate subscripts in `i` and `j`.

If the inputs `i`, `j`, and `v` are vectors or matrices, they must have the same number of elements. Alternatively, the argument `v` and/or one of the arguments `i` or `j` can be scalars.

`S = sparse(i,j,v,m,n)` specifies the size of `S` as `m`-by-`n`.

`S = sparse(i,j,v,m,n,nz)` allocates space for `nz` nonzero elements. Use this syntax to allocate extra space for nonzero values to be filled in after construction.

Examples

Save Memory Using Sparse Storage

Create a 10,000-by-10,000 full storage identity matrix.

```
A = eye(10000);
whos A
```

```
  Name          Size                    Bytes  Class     Attributes

  A         10000x10000             800000000  double
```

This matrix uses 800-megabytes of storage.

Convert the matrix to sparse storage.

```
S = sparse(A);
whos S
```

```
  Name          Size                 Bytes  Class     Attributes

  S         10000x10000             240008  double    sparse
```

In sparse form, the same matrix uses roughly 0.25-megabytes of storage. In this case, you can avoid full storage completely by using the `speye` function, which creates sparse identity matrices directly.

Sparse Matrix of All Zeros

```
S = sparse(10000,5000)
```

```
S =

   All zero sparse: 10000-by-5000
```

Sparse Matrix of Nonzeros with Specified Size

Create a 1500-by-1500 sparse matrix from the triplets `i`, `j`, and `v`.

```
i = [900 1000];
j = [900 1000];
v = [10 100];
S = sparse(i,j,v,1500,1500)


S =

 (900,900)      10
(1000,1000)    100
```

When you specify a size larger than `max(i)` -by- `max(j)`, the `sparse` function pads the output with extra rows and columns of zeros.

```
size(S)


ans =

        1500         1500
```

Preallocate Storage in Sparse Matrix

Create a sparse matrix with 10 nonzero values, but which has space allocated for 100 nonzero values.

```
S = sparse(1:10,1:10,5,20,20,100);
N = nnz(S)


N =

    10


N_alloc = nzmax(S)


N_alloc =

   100
```

The `spalloc` function is a shorthand way to create a sparse matrix with *no* nonzero elements but which has space allotted for some number of nonzeros.

Accumulate Values into Sparse Matrix

Use repeated subscripts to accumulate values into a single sparse matrix that would otherwise require one or more loops.

Create a column vector of data and two column vectors of subscripts.

```
i = [6 6 6 5 10 10 9 9]';
j = [1 1 1 2 3 3 10 10]';
v = [100 202 173 305 410 550 323 121]';
```

Visualize the subscripts and values side-by-side.

```
[i,j,v]
```

```
ans =

     6     1   100
     6     1   202
     6     1   173
     5     2   305
    10     3   410
    10     3   550
     9    10   323
     9    10   121
```

Use the `sparse` function to accumulate the values that have identical subscripts.

```
S = sparse(i,j,v)
```

```
S =

   (6,1)       475
   (5,2)       305
  (10,3)       960
   (9,10)      444
```

Input Arguments

A — Input matrix

full matrix | sparse matrix

Input matrix, specified as a full or sparse matrix. If A is already sparse, then `sparse(A)` returns A.

Data Types: `double` | `logical`
Complex Number Support: Yes

i,j — Subscript pairs (as separate arguments)

scalars | vectors | matrices

Subscript pairs, specified as separate arguments of scalars, vectors, or matrices. Corresponding elements in `i` and `j` specify `S(i,j)` subscript pairs, which determine the placement of the values in `v` into the output. If either `i` or `j` is a vector or matrix, then the other input can be a scalar or can be a vector or matrix with the same number of elements. In that case, `sparse` uses `i(:)` and `j(:)` as the subscripts. If `i` and `j` have identical values for several elements in `v`, then those elements are added together.

Note: If any value in `i` or `j` is larger than `2^31-1` for 32-bit platforms, or `2^48-1` on 64-bit platforms, then the sparse matrix cannot be constructed.

Data Types: `double` | `logical`

v — Values

scalar | vector | matrix

Values, specified as a scalar, vector, or matrix. If `v` is a vector or matrix, then one of the inputs `i` or `j` must also be a vector or matrix with the same number of elements.

Any elements in `v` that are zero are ignored, as are the corresponding subscripts in `i` and `j`. However, if you do not specify the dimension sizes of the output, `m` and `n`, then `sparse` calculates the maxima `m = max(i)` and `n = max(j)` before ignoring any zero elements in `v`.

Data Types: `double` | `logical`

Complex Number Support: Yes

`m,n` — Size of each dimension (as separate arguments)

integer values

Size of each dimension, specified as separate arguments of integer values. If you specify `m` (the row size), you also must specify `n` (the column size).

If you do not specify `m` and `n`, then `sparse` uses the default values `m = max(i)` and `n = max(j)`. These maxima are computed before any zeros in `v` are removed.

Data Types: `double`

`nz` — Storage allocation for nonzero elements

positive integer scalar

Storage allocation for nonzero elements, specified as a positive integer scalar. `nz` must be greater than or equal to `max([numel(i), numel(j), numel(v)])`.

For a sparse matrix, `S`, the `nnz` function returns the number of nonzero elements in the matrix, and the `nzmax` function returns the amount of storage allocated for nonzero matrix elements. If `nnz(S)` and `nzmax(S)` return different results, then more storage might be allocated than is actually required. For this reason, set `nz` only in anticipation of later fill-in.

If you do not specify `nz`, the `sparse` function uses a default value of `max([numel(i), numel(j), numel(v)])`.

Data Types: `double`

More About

Tips

- The `accumarray` function has similar accumulation behavior to that of `sparse`.
  - `accumarray` groups data into bins using *n*-dimensional subscripts, but `sparse` groups data into bins using 2-D subscripts.
  - `accumarray` adds elements that have identical subscripts into the output by default, but can optionally apply any function to the bins. `sparse` only adds elements that have identical subscripts into the output.

- " Constructing Sparse Matrices"
- "Sparse Matrix Operations"
- "Accessing Sparse Matrices"

See Also

`accumarray` | `diag` | `find` | `full` | `issparse` | `nnz` | `nonzeros` | `nzmax` | `spalloc` | `speye` | `spones` | `sprandn` | `sprandsym` | `spy`

Introduced before R2006a

spaugment

Form least squares augmented system

Syntax

```
S = spaugment(A,c)
S = spaugment(A)
```

Description

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix `S = [c*I A; A' 0]`. The matrix `S` is related to the least squares problem

```
min norm(b - A*x)
```

by

```
r = b - A*x
S * [r/c; x] = [b; 0]
```

The optimum value of the residual scaling factor `c`, involves `min(svd(A))` and `norm(r)`, which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses `max(max(abs(A)))/1000`.

Note In previous versions of MATLAB product, the augmented matrix was used by sparse linear equation solvers, \ and /, for nonsquare problems. Now, MATLAB software performs a least squares solve using the `qr` factorization of `A` instead.

See Also

`spparms`

Introduced before R2006a

spconvert

Import from sparse matrix external format

`spconvert` creates sparse matrices from a simple sparse format produced by sparse programs outside of MATLAB.

Syntax

```
S = spconvert(D)
```

Description

`S = spconvert(D)` constructs sparse matrix `S` from the columns of `D` in a manner similar to the `sparse` function.

- If `D` is of size `N`-by-`3`, then `spconvert` uses the columns `[i,j,re]` of `D` to construct `S`, such that `S(i(k), j(k)) = re(k)`.
- If `D` is of size `N`-by-`4`, then `spconvert` uses the columns `[i,j,re,im]` of `D` to construct `S`, such that `S(i(k), j(k)) = re(k) + 1i*im(k)`.

Examples

Convert Data File to Sparse Matrix

Create an ASCII file, `uphill.dat`, which contains the following values. Save the file in your current directory.

```
1     1     1.000000000000000
1     2     0.500000000000000
2     2     0.333333333333333
1     3     0.333333333333333
2     3     0.250000000000000
3     3     0.200000000000000
1     4     0.250000000000000
2     4     0.200000000000000
```

```
3     4     0.166666666666667
4     4     0.142857142857143
4     4     0.000000000000000
```

It is common to purposefully make the last line of the file include the desired size of the matrix with a value of 0. This practice ensures that the converted sparse matrix has that size.

Load the data into MATLAB and convert it into a sparse matrix.

```
load uphill.dat
H = spconvert(uphill)
```

```
H =

   (1,1)        1.0000
   (1,2)        0.5000
   (2,2)        0.3333
   (1,3)        0.3333
   (2,3)        0.2500
   (3,3)        0.2000
   (1,4)        0.2500
   (2,4)        0.2000
   (3,4)        0.1667
   (4,4)        0.1429
```

In this case, the last line in the file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.

Input Arguments

`D` — Input matrix

matrix

Input matrix, specified as a matrix with either three or four columns. In both cases, the first two columns of `D` are subscripts and the third column is composed of data values. A four column matrix specifies the real (third column) and imaginary (fourth column) parts of complex numbers.

If `D` is already a sparse matrix, then `spconvert` returns `D`.

Data Types: `single` | `double`

See Also

`full` | `sparse`

Introduced before R2006a

spdiags

Extract and create sparse band and diagonal matrices

Syntax

```
B = spdiags(A)
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)
```

Description

The `spdiags` function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible.

`B = spdiags(A)` extracts all nonzero diagonals from the `m`-by-`n` matrix `A`. `B` is a `min(m,n)`-by-`p` matrix whose columns are the `p` nonzero diagonals of `A`.

`[B,d] = spdiags(A)` returns a vector `d` of length `p`, whose integer components specify the diagonals in `A`.

`B = spdiags(A,d)` extracts the diagonals specified by `d`.

`A = spdiags(B,d,A)` replaces the diagonals specified by `d` with the columns of `B`. The output is sparse.

`A = spdiags(B,d,m,n)` creates an `m`-by-`n` sparse matrix by taking the columns of `B` and placing them along the diagonals specified by `d`.

Note In this syntax, if a column of `B` is longer than the diagonal it is replacing, and `m >= n`, `spdiags` takes elements of super-diagonals from the lower part of the column of `B`, and elements of sub-diagonals from the upper part of the column of `B`. However, if `m < n`, then super-diagonals are from the upper part of the column of `B`, and sub-diagonals from the lower part. (See “Example 5A” on page 1-7550 and “Example 5B” on page 1-7551, below).

Arguments

The `spdiags` function deals with three matrices, in various combinations, as both input and output.

A	An `m`-by-`n` matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on `p` diagonals.
B	A `min(m,n)`-by-`p` matrix, usually (but not necessarily) full, whose columns are the diagonals of `A`.
d	A vector of length `p` whose integer components specify the diagonals in `A`.

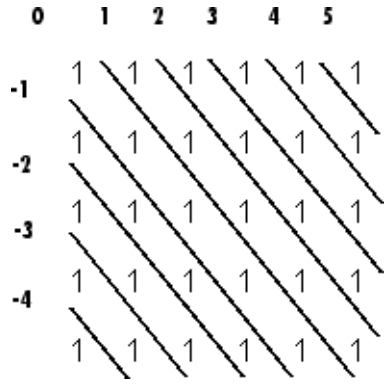
Roughly, `A`, `B`, and `d` are related by

```
for k = 1:p
    B(:,k) = diag(A,d(k))
end
```

Some elements of `B`, corresponding to positions outside of `A`, are not defined by these loops. They are not referenced when `B` is input and are set to zero when `B` is output.

How the Diagonals of A are Listed in the Vector d

An m-by-n matrix `A` has m+n-1diagonals. These are specified in the vector `d` using indices from -m+1 to n-1. For example, if `A` is 5-by-6, it has 10 diagonals, which are specified in the vector `d` using the indices -4, -3 , ... 4, 5. The following diagram illustrates this for a vector of all ones.



Examples

Example 1

For the following matrix,

```
A=[0 5 0 10 0 0;...
0 0 6 0 11 0;...
3 0 0 7 0 12;...
1 4 0 0 8 0;...
0 2 5 0 0 9]
```

```
A =

     0     5     0    10     0     0
     0     0     6     0    11     0
     3     0     0     7     0    12
     1     4     0     0     8     0
     0     2     5     0     0     9
```

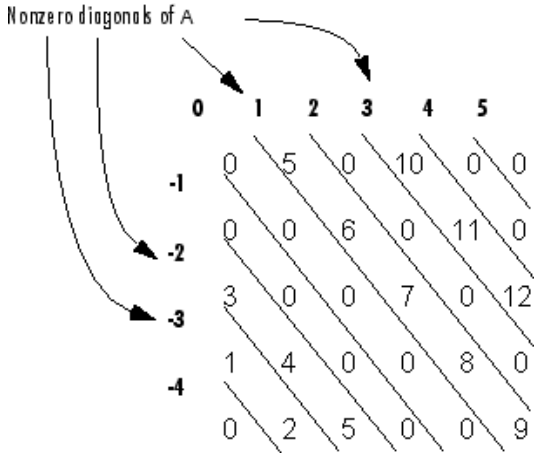
the command

```
[B, d]  =spdiags(A)
```

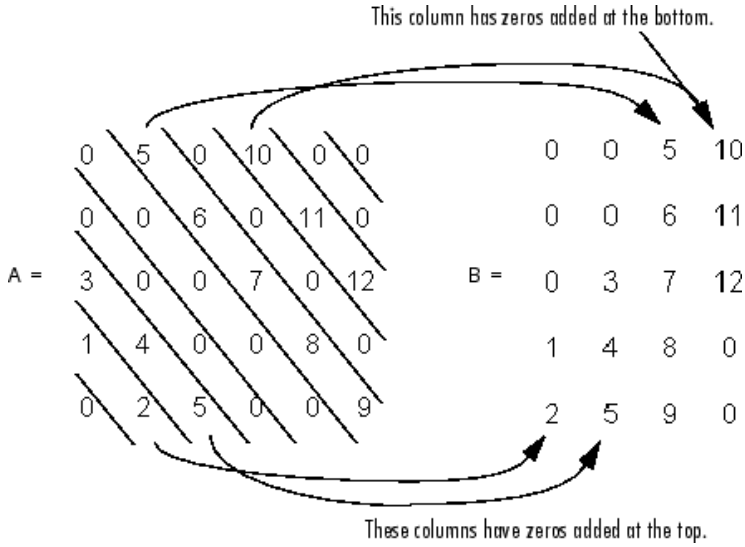
returns

```
B =

     0     0     5    10
     0     0     6    11
     0     3     7    12
     1     4     8     0
     2     5     9     0


d =


    -3
    -2
     1
     3
```

The columns of the first output B contain the nonzero diagonals of A. The second output d lists the indices of the nonzero diagonals of A, as shown in the following diagram. See “How the Diagonals of A are Listed in the Vector d” on page 1-7545.



Note that the longest nonzero diagonal in A is contained in column 3 of B. The other nonzero diagonals of A have extra zeros added to their corresponding columns in B, to give all columns of B the same length. For the nonzero diagonals below the main diagonal of A, extra zeros are added at the tops of columns. For the nonzero diagonals above the main diagonal of A, extra zeros are added at the bottoms of columns. This is illustrated by the following diagram.



Example 2

This example generates a sparse tridiagonal representation of the classic second difference operator on `n` points.

```
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson's test matrix (see `gallery`):

```
A = spdiags(abs(-(n-1)/2:(n-1)/2)',0,A)
```

Finally, recover the three diagonals:

```
B = spdiags(A)
```

Example 3

The second example is not square.

```
A = [11     0    13     0
       0    22     0    24
       0     0    33     0
```

```
     41     0     0    44
      0    52     0     0
      0     0    63     0
      0     0     0    74]
```

Here `m =7`, `n = 4`, and `p = 3`.

The statement `[B,d] = spdiags(A)` produces `d = [-3 0 2]'` and

```
B = [41    11     0
     52    22     0
     63    33    13
     74    44    24]
```

Conversely, with the above `B` and `d`, the expression `spdiags(B,d,7,4)` reproduces the original `A`.

Example 4

This example shows how `spdiags` creates the diagonals when the columns of `B` are longer than the diagonals they are replacing.

```
B = repmat((1:6)',[1 7])

B =

    1  1  1  1  1  1  1
    2  2  2  2  2  2  2
    3  3  3  3  3  3  3
    4  4  4  4  4  4  4
    5  5  5  5  5  5  5
    6  6  6  6  6  6  6

d = [-4 -2 -1 0 3 4 5];
A = spdiags(B,d,6,6);
full(A)

ans =

  1  0  0  4  5  6
  1  2  0  0  5  6
  1  2  3  0  0  6
  0  2  3  4  0  0
  1  0  3  4  5  0
```

```
  0   2   0   4   5   6
```

Example 5A

This example illustrates the use of the syntax `A = spdiags(B,d,m,n)`, under three conditions:

- `m` is equal to `n`
- `m` is greater than `n`
- `m` is less than `n`

The command used in this example is

```
A = full(spdiags(B, [-2 0 2], m, n))
```

where `B` is the 5-by-3 matrix shown below. The resulting matrix `A` has dimensions `m`-by-`n`, and has nonzero diagonals at `[-2 0 2]` (a sub-diagonal at `-2`, the main diagonal, and a super-diagonal at `2`).

```
B =
    1     6    11
    2     7    12
    3     8    13
    4     9    14
    5    10    15
```

The first and third columns of matrix `B` are used to create the sub- and super-diagonals of `A` respectively. In all three cases though, these two outer columns of `B` are longer than the resulting diagonals of `A`. Because of this, only a part of the columns is used in `A`.

When `m == n` or `m > n`, `spdiags` takes elements of the super-diagonal in `A` from the lower part of the corresponding column of `B`, and elements of the sub-diagonal in `A` from the upper part of the corresponding column of `B`.

When `m < n`, `spdiags` does the opposite, taking elements of the super-diagonal in `A` from the upper part of the corresponding column of `B`, and elements of the sub-diagonal in `A` from the lower part of the corresponding column of `B`.

Part 1 — m is equal to n.

```
A = full(spdiags(B, [-2 0 2], 5, 5))
 Matrix B                      Matrix A
```

```
1    6   11                 6    0   13    0    0
2    7   12                 0    7    0   14    0
3    8   13  == spdiags =>  1    0    8    0   15
4    9   14                 0    2    0    9    0
5   10   15                 0    0    3    0   10
```

`A(3,1)`, `A(4,2)`, and `A(5,3)` are taken from the upper part of `B(:,1)`.

`A(1,3)`, `A(2,4)`, and `A(3,5)` are taken from the lower part of `B(:,3)`.

Part 2 — m is greater than n.

```
A = full(spdiags(B, [-2 0 2], 5, 4))
 Matrix B                      Matrix A

1    6   11                 6    0   13    0
2    7   12                 0    7    0   14
3    8   13  == spdiags =>  1    0    8    0
4    9   14                 0    2    0    9
5   10   15                 0    0    3    0
```

Same as in Part A.

Part 3 — m is less than n.

```
A = full(spdiags(B, [-2 0 2], 4, 5))
 Matrix B                      Matrix A

1    6   11                 6    0   11    0    0
2    7   12                 0    7    0   12    0
3    8   13  == spdiags =>  3    0    8    0   13
4    9   14                 0    4    0    9    0
5   10   15
```

`A(3,1)` and `A(4,2)` are taken from the lower part of `B(:,1)`.

`A(1,3)`, `A(2,4)`, and `A(3,5)` are taken from the upper part of `B(:,3)`.

Example 5B

Extract the diagonals from the first part of this example back into a column format using the command

```
B = spdiags(A)
```

You can see that in each case the original columns are restored (minus those elements that had overflowed the super- and sub-diagonals of matrix A).

Part 1.

```
      Matrix A                        Matrix B

6     0    13     0     0                1    6    0
0     7     0    14     0                2    7    0
1     0     8     0    15  == spdiags =>  3    8   13
0     2     0     9     0                0    9   14
0     0     3     0    10                0   10   15
```

Part 2.

```
    Matrix A                     Matrix B

6     0    13     0                1    6    0
0     7     0    14                2    7    0
1     0     8     0  == spdiags =>  3    8   13
0     2     0     9                0    9   14
0     0     3     0
```

Part 3.

```
      Matrix A                        Matrix B

6     0    11     0     0                0    6   11
0     7     0    12     0                0    7   12
3     0     8     0    13  == spdiags =>  3    8   13
0     4     0     9     0                4    9    0
```

See Also

diag | speye

Introduced before R2006a

specular

Calculate specular reflectance

Syntax

```
R = specular(Nx,Ny,Nz,S,V)
```

Description

`R = specular(Nx,Ny,Nz,S,V)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` and `V` specify the direction to the light source and to the viewer, respectively. You can specify these directions as three vectors`[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

The specular highlight is strongest when the normal vector is in the direction of `(S+V)/2` where `S` is the source direction, and `V` is the view direction.

The surface spread exponent can be specified by including a sixth argument as in `specular(Nx,Ny,Nz,S,V,spread)`.

speye

Sparse identity matrix

Syntax

```
S = speye(m,n)
S = speye([m n])
S = speye(n)
S = speye
```

Description

`S = speye(m,n)` and `S = speye([m n])` form an `m`-by-`n` sparse matrix with `1`s on the main diagonal.

`S = speye(n)` abbreviates `speye(n,n)`.

`S = speye` returns the sparse form of the 1-by-1 identity matrix.

Examples

`I = speye(1000)` forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as `I = sparse(eye(1000,1000))`, but the latter requires eight megabytes for temporary storage for the full representation.

See Also

`spalloc` | `sprand` | `spones` | `spdiags` | `sprandn`

Introduced before R2006a

spfun

Apply function to nonzero sparse matrix elements

Syntax

```
f = spfun(fun,S)
```

Description

The `spfun` function selectively applies a function to only the *nonzero* elements of a sparse matrix `S`, preserving the sparsity pattern of the original matrix (except for underflow or if `fun` returns zero for some nonzero elements of `S`).

`f = spfun(fun,S)` evaluates `fun(S)` on the elements of `S` that are nonzero. `fun` is a function handle.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

Examples

Given the 4-by-4 sparse diagonal matrix

```
S = spdiags([1:4]',0,4,4)
```

```
S =
   (1,1)         1
   (2,2)         2
   (3,3)         3
   (4,4)         4
```

Because `fun` returns nonzero values for all nonzero element of `S`, `f = spfun(@exp,S)` has the same sparsity pattern as `S`.

```
f =
   (1,1)       2.7183
   (2,2)       7.3891
```

```
   (3,3)       20.0855
   (4,4)       54.5982
```

whereas `exp(S)` has `1`s where `S` has `0`s.

```
full(exp(S))

ans =
    2.7183    1.0000    1.0000    1.0000
    1.0000    7.3891    1.0000    1.0000
    1.0000    1.0000   20.0855    1.0000
    1.0000    1.0000    1.0000   54.5982
```

More About

Tips

Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

See Also

`function_handle`

Introduced before R2006a

sph2cart

Transform spherical coordinates to Cartesian

Syntax

```
[x,y,z] = sph2cart(azimuth,elevation,r)
```

Description

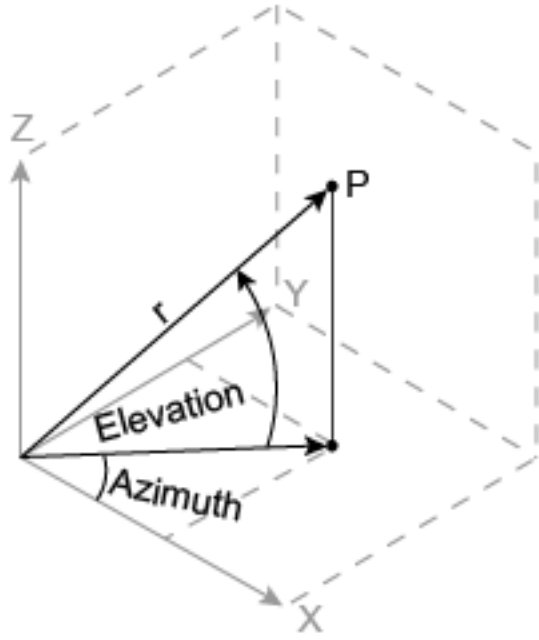
`[x,y,z] = sph2cart(azimuth,elevation,r)` transforms the corresponding elements of spherical coordinate arrays to Cartesian, or *xyz*, coordinates. `azimuth`, `elevation`, and `r` must all be the same size (or any of them can be scalar). `azimuth` and `elevation` are angular displacements in radians. `azimuth` is the counterclockwise angle in the *x*-*y* plane measured from the positive *x*-axis. `elevation` is the elevation angle from the *x*-*y* plane.

More About

Algorithms

The mapping from spherical coordinates to three-dimensional Cartesian coordinates is

```
x = r .* cos(elevation) .* cos(azimuth)
y = r .* cos(elevation) .* sin(azimuth)
z = r .* sin(elevation)
```



See Also

`cart2pol` | `cart2sph` | `pol2cart`

Introduced before R2006a

sphere

Generate sphere

Syntax

```
sphere
sphere(n)
[X,Y,Z] = sphere(n)
```

Description

The `sphere` function generates the *x*-, *y*-, and *z*-coordinates of a unit sphere for use with `surf` and `mesh`.

`sphere` generates a sphere consisting of 20-by-20 faces.

`sphere(n)` draws a `surf` plot of an `n`-by-`n` sphere in the current figure.

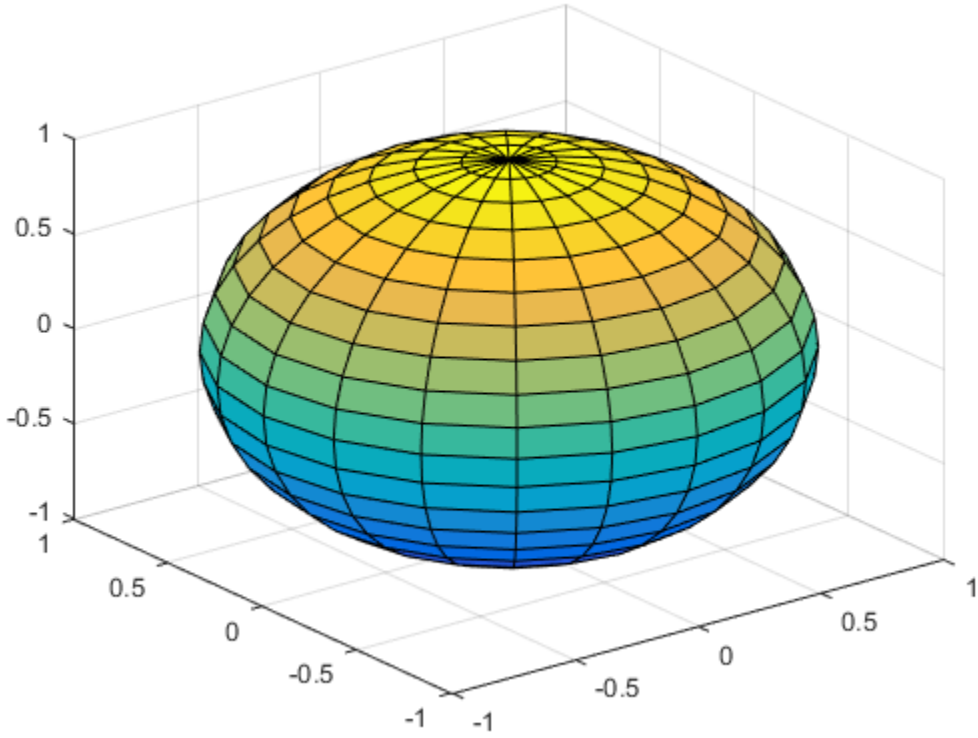
`[X,Y,Z] = sphere(n)` returns the coordinates of a sphere in three matrices that are `(n+1)`-by-`(n+1)` in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

Examples

Plot Sphere

Generate and plot a sphere.

```
figure
sphere
```



Plot Multiple Spheres

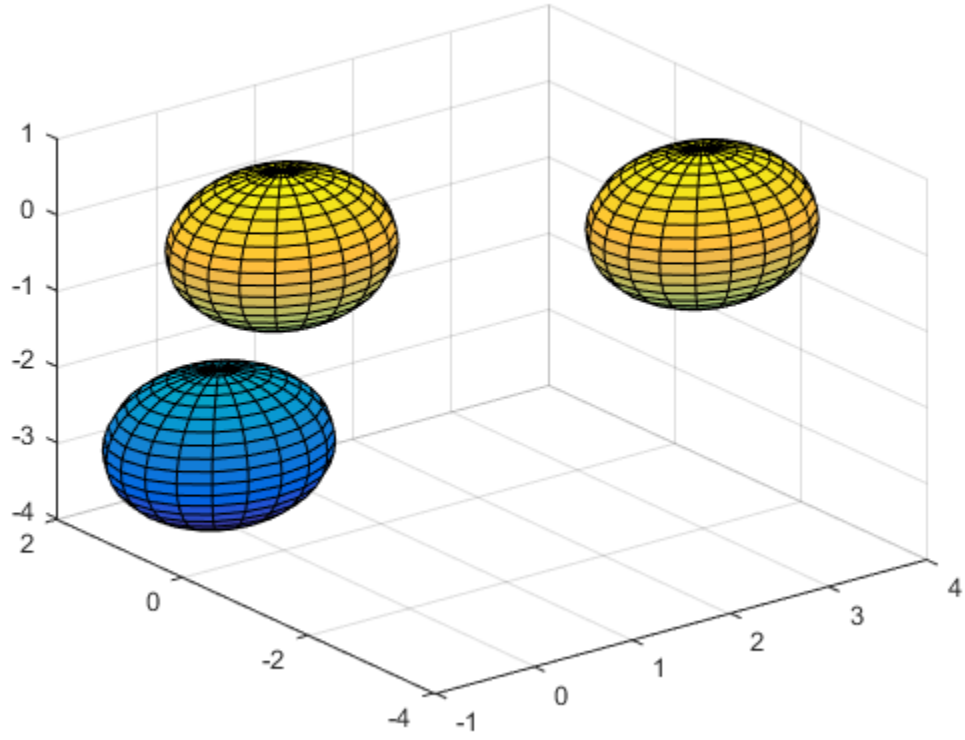
Define `x`, `y`, and `z` as coordinates of a sphere.

```
[x,y,z] = sphere;
```

Plot a sphere centered at the origin. Plot two more spheres centered at `(3,-2,0)` and `(0,1,-3)`.

```
figure
surf(x,y,z)

hold on
surf(x+3,y-2,z) % centered at (3,-2,0)
```

```
surf(x,y+1,z-3) % centered at (0,1,-3)
```



See Also

`cylinder` | `axis`

Introduced before R2006a

spinmap

Spin colormap

Syntax

```
spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')
```

Description

The `spinmap` function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

`spinmap` cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

`spinmap(t)` rotates the colormap for approximately `10*t` seconds. The amount of time specified by `t` depends on your hardware configuration (e.g., if you are running MATLAB software over a network).

`spinmap(t,inc)` rotates the colormap for approximately `10*t` seconds and specifies an increment `inc` by which the colormap shifts. When `inc` is 1, the rotation appears smoother than the default (i.e., 2). Increments greater than 2 are less smooth than the default. A negative increment (e.g., –2) rotates the colormap in a negative direction.

`spinmap('inf')` rotates the colormap for an infinite amount of time. To break the loop, press **Ctrl+C**.

See Also

`colormap` | `colormapeditor`

Introduced before R2006a

spline

Cubic spline data interpolation

Syntax

```
yy = spline(x,Y,xx)
pp = spline(x,Y)
```

Description

`yy = spline(x,Y,xx)` uses a cubic spline interpolation to find `yy`, the values of the underlying function `Y` at the values of the interpolant `xx`. For the interpolation, the independent variable is assumed to be the final dimension of `Y` with the breakpoints defined by `x`. The values in `x` must be distinct.

The sizes of `xx` and `yy` are related as follows:

- If `Y` is a scalar or vector, `yy` has the same size as `xx`.
- If `Y` is an array that is not a vector,
  - If `xx` is a scalar or vector, `size(yy)` equals `[d1, d2, ..., dk, length(xx)]`.
  - If `xx` is an array of size `[m1,m2,...,mj]`, `size(yy)` equals `[d1,d2,...,dk,m1,m2,...,mj]`.

`pp = spline(x,Y)` returns the piecewise polynomial form of the cubic spline interpolant for later use with `ppval` and the spline utility `unmkpp`. `x` must be a vector with distinct values. `Y` can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If `x` and `Y` are vectors of the same size, the not-a-knot end conditions are used.
- If `x` or `Y` is a scalar, it is expanded to have the same length as the other and the not-a-knot end conditions are used. (See Exceptions (1) below).
- If `Y` is a vector that contains two more values than `x` has entries, the first and last value in `Y` are used as the endslopes for the cubic spline. (See Exceptions (2) below.)

Exceptions

1. If `Y` is a vector that contains two more values than `x` has entries, the first and last value in `Y` are used as the endslopes for the cubic spline. If `Y` is a vector, this means
   - `f(x) = Y(2:end-1)`
   - `df(min(x)) = Y(1)`
   - `df(max(x)) = Y(end)`
2. If `Y` is a matrix or an `N`-dimensional array with `size(Y,N)` equal to `length(x)+2`, the following hold:
   - `f(x(j))` matches the value `Y(:,...,:,j+1)` for `j=1:length(x)`
   - `Df(min(x))` matches `Y(:,:,...:,1)`
   - `Df(max(x))` matches `Y(:,:,...:,end)`

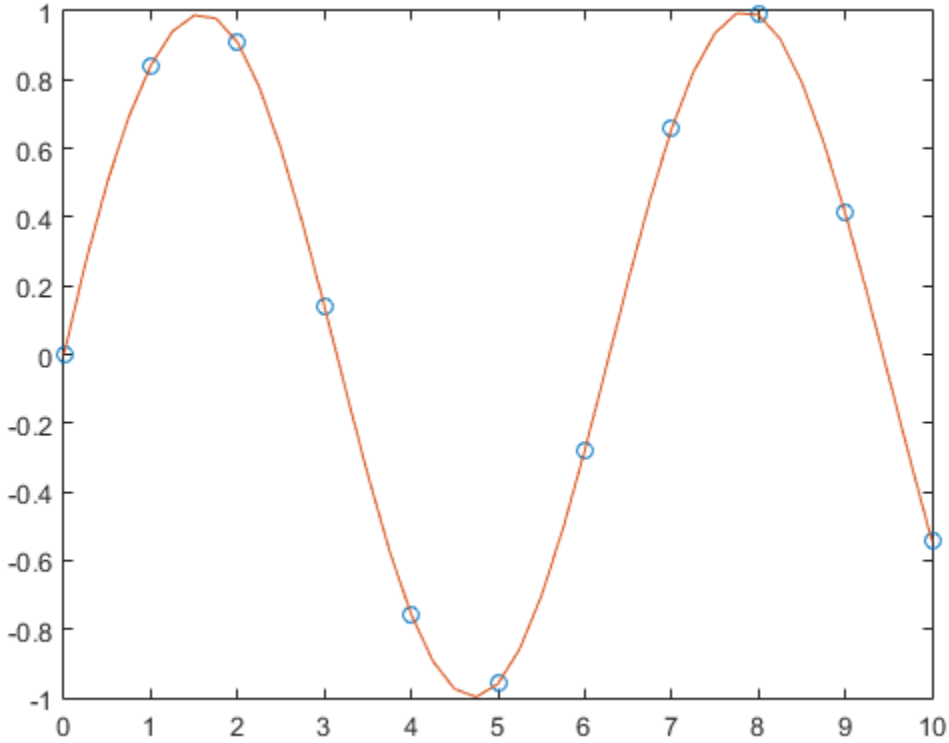
Note You can also perform spline interpolation using the `interp1` function with the command `interp1(x,y,xx,'spline')`. Note that while `spline` performs interpolation on rows of an input matrix, `interp1` performs interpolation on columns of an input matrix.

Examples

Spline Interpolation of Sine Data

This generates a sine curve, then samples the spline over a finer mesh.

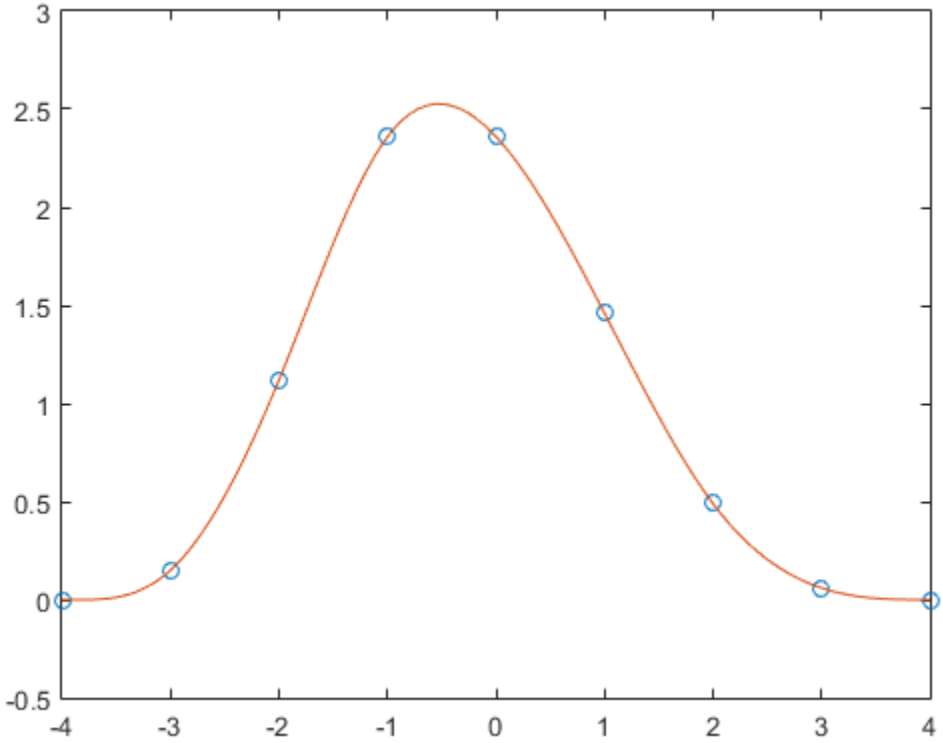
```
x = 0:10;
y = sin(x);
xx = 0:.25:10;
yy = spline(x,y,xx);
plot(x,y,'o',xx,yy)
```



Spline Interpolation of Distribution and Specify Endpoint Slopes

This illustrates the use of clamped or complete spline interpolation where end slopes are prescribed. Zero slopes at the ends of an interpolant to the values of a certain distribution are enforced.

```
x = -4:4;
y = [0 .15 1.12 2.36 2.36 1.46 .49 .06 0];
cs = spline(x,[0 y 0]);
xx = linspace(-4,4,101);
plot(x,y,'o',xx,ppval(cs,xx),'-');
```



Extrapolation Using Cubic Spline

The two vectors

```
t = 1900:10:1990;
p = [ 75.995  91.972  105.711  123.203  131.669 ...
     150.697 179.323  203.212  226.505  249.633 ];
```

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

```
spline(t,p,2000)
```

uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

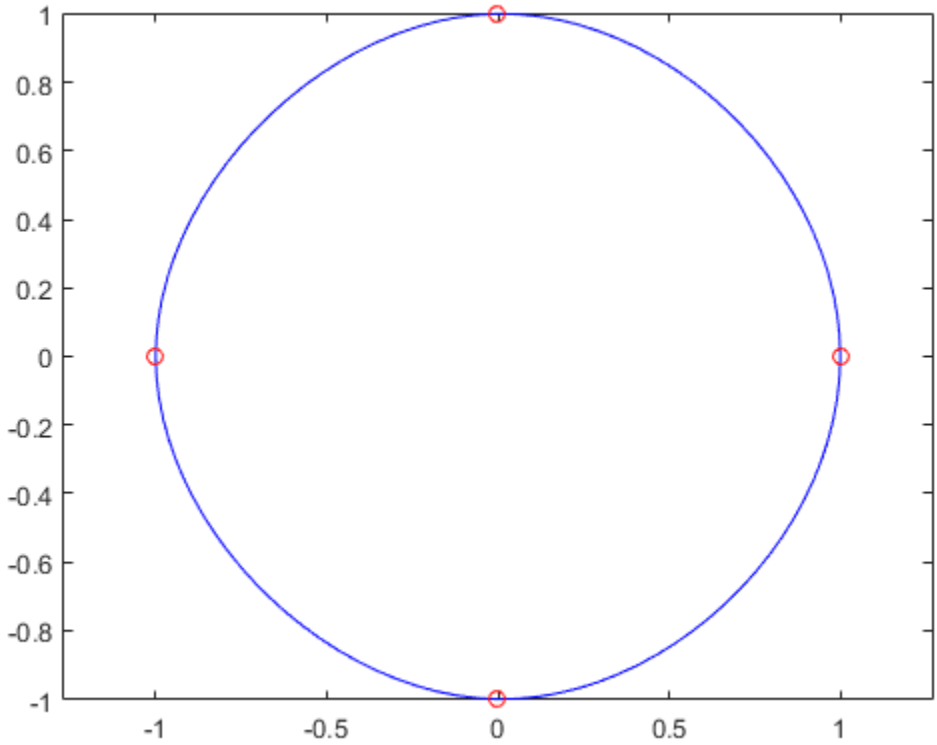
```
ans =

  270.6060
```

Spline Interpolation of Angular Data

The statements

```
x = pi*[0:.5:2];
y = [0  1  0 -1  0  1  0;
     1  0  1  0 -1  0  1];
pp = spline(x,y);
yy = ppval(pp, linspace(0,2*pi,101));
plot(yy(1,:),yy(2,:),'-b',y(1,2:5),y(2,2:5),'or'), axis equal
```

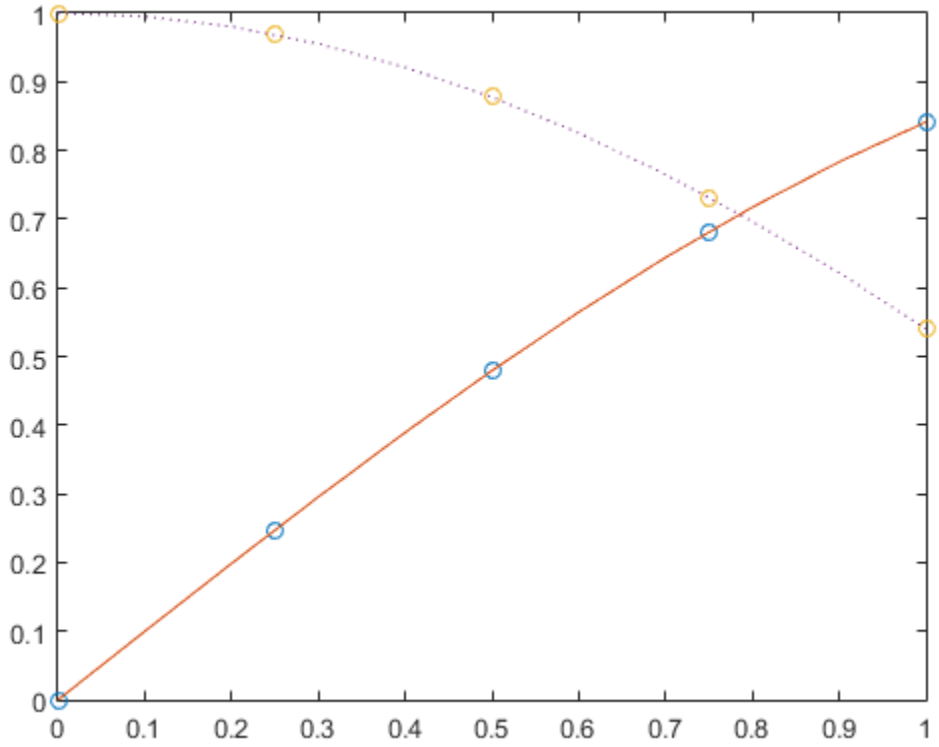
generate the plot of a circle, with the five data points `y(:,2),...,y(:,6)` marked with o's. Note that this `y` contains two more values (i.e., two more columns) than does `x`, hence `y(:,1)` and `y(:,end)` are used as endslopes.



Spline Interpolation of Sine and Cosine Data

The following code generates sine and cosine curves, then samples the splines over a finer mesh.

```
x = 0:.25:1;
Y = [sin(x); cos(x)];
xx = 0:.1:1;
YY = spline(x,Y,xx);
plot(x,Y(1,:),'o',xx,YY(1,:),'-')
hold on
plot(x,Y(2,:),'o',xx,YY(2,:),':')
hold off
```



More About

Algorithms

A tridiagonal linear system (with, possibly, several right sides) is being solved for the information needed to describe the coefficients of the various cubic polynomials which make up the interpolating spline. `spline` uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. For access to more advanced features, see the `interp1` reference page, the command-line help for these functions, and the Curve Fitting Toolbox spline functions.

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

See Also

`interp1` | `ppval` | `mkpp` | `pchip` | `unmkpp`

Introduced before R2006a

split

Split calendar duration into numeric and duration units

Syntax

```
[X1,X2,...] = split(t,units)
```

Description

`[X1,X2,...] = split(t,units)` returns the calendar duration values in `t` as separate numeric arrays, one for each of the date or time units specified by `units`. The number of date and time units specified by `units` determines the number of output arguments.

Examples

Split Calendar Duration Array

Create a `calendarDuration` array.

```
T = calmonths(15:17) + caldays(8) + hours(1.2345)
```

```
T =

   1y 3mo 8d 1h 14m 4.2s   1y 4mo 8d 1h 14m 4.2s   1y 5mo 8d 1h 14m 4.2s
```

Get the month, day, and time values.

```
[m,d,t] = split(T,{'months','days','time'})
```

```
m =

    15    16    17
```

```
d =

     8     8     8

t =

   01:14:04   01:14:04   01:14:04
```

Get the year, month, day, and time values

```
[y,m,d,t] = split(T,{'years','months','days','time'})

y =

     1     1     1

m =

     3     4     5

d =

     8     8     8

t =

   01:14:04   01:14:04   01:14:04
```

When you request both the year and month values, `split` carries over month values greater than 12 to the year value.

Input Arguments

`t` — Input calendar duration

`calendarDuration` array

Input calendar duration, specified as a `calendarDuration` array.

`units` — Date and time units

string | cell array of strings

Date and time units, specified as a string or a cell array of strings. The string values can be one or more of the values in this table.

String Value	Split `t` into Units of...
`'years'`	years
`'quarters'`	quarters
`'months'`	months
`'weeks'`	weeks
`'days'`	days
`'time'`	time, in the format hours:minutes:seconds

You must specify date and time units from largest to smallest. For example, `{'years','months'}` is valid, but `{'months','years'}` is not.

Example: `{'years','months','days'}`

Data Types: `char` | `cell`

Output Arguments

`X1,X2,...` — Output arrays

scalar | vector | matrix | multidimensional array

Output arrays, returned as one or more scalars, vectors, matrices, or multidimensional arrays. `split` returns year, month, and day values in numeric arrays and time values in `duration` arrays.

See Also

`caldays` | `calmonths` | `calquarters` | `calweeks` | `calyears` | `time`

Introduced in R2014b

spones

Replace nonzero sparse matrix elements with ones

Syntax

```
R = spones(S)
```

Description

`R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with `1`'s in the nonzero positions.

Examples

`c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

See Also

`nnz` | `spalloc` | `spfun`

Introduced before R2006a

spparms

Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description

`spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse routines. In ordinary use, you should never need to deal with this function.

The meanings of the `key` parameters are

`'spumoni'`	Sparse Monitor flag:
`0`	Produces no diagnostic output, the default
`1`	Produces information about choice of algorithm based on matrix structure, and about storage allocation
`2`	Also produces very detailed information about the sparse matrix algorithms
`'thr_rel'`, `'thr_abs'`	Minimum degree threshold is `thr_rel*mindegree + thr_abs`.
`'exact_d'`	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.
`'supernd'`	If positive, minimum degree amalgamates the supernodes every `supernd` stages.

`'rreduce'`	If positive, minimum degree does row reduction every `rreduce` stages.
`'wh_frac'`	Rows with `density > wh_frac` are ignored in `colmmd`.
`'autommd'`	Nonzero to use minimum degree (MMD) orderings with QR-based \ and /.
`'autoamd'`	Nonzero to use `colamd` ordering with the LU-based \ and /, and to use `amd` with Cholesky-based \ and /.
`'piv_tol'`	Pivot tolerance used by the LU-based \ and /.
`'bandden'`	Band density used by \ and / for banded matrices. Band density is defined as (# nonzeros in the band)/(# nonzeros in a full band). If `bandden = 1.0`, never use band solver. If `bandden = 0.0`, always use band solver. Default is `0.5`.
`'umfpack'`	Nonzero to use UMFPACK instead of the v4 LU-based solver in \ and /.
`'sym_tol'`	Symmetric pivot tolerance. See `lu` for more information about the role of the symmetric pivot tolerance.

`spparms`, by itself, prints a description of the current settings.

`values = spparms` returns a vector whose components give the current settings.

`[keys,values] = spparms` returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

`spparms(values)`, with no output argument, sets all the parameters to the values specified by the argument vector.

`value = spparms('key')` returns the current setting of one parameter.

`spparms('default')` sets all the parameters to their default settings.

`spparms('tight')` sets the minimum degree ordering parameters to their *tight* settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for `default` and `tight` settings are

	Keyword	Default	Tight
`values(1)`	`'spumoni'`	`0.0`	

	Keyword	Default	Tight
values(2)	'thr_rel'	1.1	1.0
values(3)	'thr_abs'	1.0	0.0
values(4)	'exact_d'	0.0	1.0
values(5)	'supernd'	3.0	1.0
values(6)	'rreduce'	3.0	1.0
values(7)	'wh_frac'	0.5	0.5
values(8)	'autommd'	1.0	
values(9)	'autoamd'	1.0	
values(10)	'piv_tol'	0.1	
values(11)	'bandden'	0.5	
values(12)	'umfpack'	1.0	
values(13)	'sym_tol'	0.001	

See Also

chol | lu | qr | colamd | symamd

Introduced before R2006a

sprand

Sparse uniformly distributed random matrix

Syntax

```
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)
```

Description

`R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

`R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (`0 <= density <= 1`).

`R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

More About

Tips

- `sprand` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

See Also

`sprandn` | `sprandsym`

Introduced before R2006a

sprandn

Sparse normally distributed random matrix

Syntax

```
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)
```

Description

`R = sprandn(S)` has the same sparsity structure as `S`, but normally distributed random entries with mean `0` and variance `1`.

`R = sprandn(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (`0 <= density <= 1`).

`R = sprandn(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

More About

Tips

- `sprandn` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

See Also

`sprand` | `sprandsym`

Introduced before R2006a

sprandsym

Sparse symmetric random matrix

Syntax

```
R = sprandsym(S)
R = sprandsym(n,density)
R = sprandsym(n,density,rc)
R = sprandsym(n,density,rc,kind)
R = sprandsym(S,[],rc,3)
```

Description

`R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as `S`. Its elements are normally distributed, with mean `0` and variance `1`.

`R = sprandsym(n,density)` returns a symmetric random, `n`-by-`n`, sparse matrix with approximately `density*n*n` nonzeros; each entry is the sum of one or more normally distributed random samples, and (`0 <= density <= 1`).

`R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to `rc`. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in [−1,1].

If `rc` is a vector of length `n`, then `R` has eigenvalues `rc`. Thus, if `rc` is a positive (nonnegative) vector then `R` is a positive (nonnegative) definite matrix. In either case, `R` is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

`R = sprandsym(n,density,rc,kind)` is positive definite.

- If kind = 1, `R` is generated by random Jacobi rotation of a positive definite diagonal matrix. `R` has the desired condition number exactly.
- If kind = 2, `R` is a shifted sum of outer products. `R` has the desired condition number only approximately, but has less structure.

`R = sprandsym(S,[],rc,3)` has the same structure as the matrix `S` and approximate condition number `1/rc`.

More About

Tips

`sprandsym` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

See Also

`sprand | sprandn`

Introduced before R2006a

sprank

Structural rank

Syntax

```
r = sprank(A)
```

Description

`r = sprank(A)` is the structural rank of the sparse matrix `A`. For all values of `A`,

```
sprank(A) >= rank(full(A))
```

In exact arithmetic, `sprank(A) == rank(full(sprandn(A)))` with a probability of one.

Examples

```
A = [1    0    2    0
     2    0    4    0 ];

A = sparse(A);

sprank(A)

ans =
     2

rank(full(A))

ans =
     1
```

See Also

`dmperm`

Introduced before R2006a

sprintf

Format data into string

Syntax

```
str = sprintf(formatSpec,A1,...,An)
[str,errmsg] = sprintf(formatSpec,A1,...,An)
```

Description

`str = sprintf(formatSpec,A1,...,An)` formats the data in arrays `A1,...,An` according to `formatSpec` in column order, and returns the results to string `str`.

`[str,errmsg] = sprintf(formatSpec,A1,...,An)` returns an error message string when the operation is unsuccessful. Otherwise, `errmsg` is empty.

Examples

Floating-Point Formats

Format a floating-point number using `%e`, `%f`, and `%g` specifiers.

```
A = 1/eps;
str_e = sprintf('%0.5e',A)
str_f = sprintf('%0.5f',A)
str_g = sprintf('%0.5g',A)
```

```
str_e =

4.50360e+15


str_f =

4503599627370496.00000


str_g =
```

```
4.5036e+15
```

Literal Text and Array Inputs

Combine literal text with array values to create a string.

```
formatSpec = 'The array is %dx%d.';
A1 = 2;
A2 = 3;
str = sprintf(formatSpec,A1,A2)
```

```
str =

The array is 2x3.
```

Integer Format with Floating-Point Inputs

Explicitly convert double-precision values to integers.

```
str = sprintf('%d',round(pi))
```

```
str =

3
```

Specify Field Width of a Printed Value

Specify the minimum width of the printed value.

```
str = sprintf('%025d',[123456])
```

```
str =
0000000000000000000123456
```

The `0` flag in the `%025d` format specifier requests leading zeros in the output.

Reorder Inputs Using Position Identifier (n$)

Reorder the input values using the `n$` position identifier.

```
A1 = 'X';
```

```
A2 = 'Y';
A3 = 'Z';
formatSpec = ' %3$s %2$s %1$s';
str = sprintf(formatSpec,A1,A2,A3)

str =
 Z Y X
```

Create a String from Values in Cell Array

```
C = { 1,   2,   3 ;
     'AA','BB','CC'};

str = sprintf(' %d %s',C{:})

str =
 1 AA 2 BB 3 CC
```

The syntax `C{:}` creates a comma-separated list of arrays that contain the contents of each cell from `C` in column order. For example, `C{1}==1` and `C{2}=='AA'`.

Input Arguments

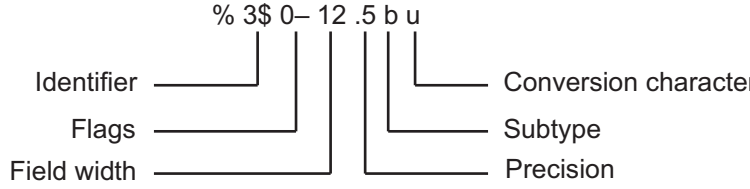
`formatSpec` — Format of output fields

string containing formatting operators

Format of the output fields, specified as a string containing formatting operators. `formatSpec` also can include ordinary text and special characters.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).



Conversion Character

This table shows conversion characters to format numeric and character data as strings.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a`–`f`
	`%X`	Same as `%x`, uppercase letters `A`–`F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters	`%c`	Single character
	`%s`	String of characters

Optional Operators

The optional identifier, flags, field width, precision, and subtype operators further define the format of the output string.

- **Identifier**

Order for processing values from the input list. Use the syntax *n*$, where *n* represents the position of the value in the input list.

Example: `'%3$s %2$s %1$s %2$s'` prints inputs `'A'`, `'B'`, `'C'` as follows: `C B A B`.

- **Flags**

`'–'`	Left-justify. **Example:** `%-5.2f`
`'+'`	Always print a sign character (+ or –) for any value. **Example:** `%+5.2f`
`' '`	Insert a space before the value. **Example:** `% 5.2f`
`'0'`	Pad to field width with zeros before the value. **Example:** `%05.2f`
`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. **Example:** `%#5.0f`

- **Field Width**

  Minimum number of characters to print. The field width operator can be a number, or an asterisk (`*`) to refer to an argument in the input list.

  Example: The input list `('%12d',intmax)` is equivalent to `('%*d', 12, intmax)`.

  The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`

For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `' 3.142'`

The precision operator can be a number, or an asterisk (`*`) to refer to an argument in the input list.

Example: The input list `('%6.4f', pi)` is equivalent to `('%*.*f', 6, 4, pi)`.

Note: If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

- **Subtypes**

  Certain conversion characters can support a subtype. The subtype operator immediately precedes the conversion character. This table shows the conversions that can use subtypes.

Input Value Type	Subtype and Conversion Character	Output Value Type

Floating-point number	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Integer	`%ld` or `%li`
	`%lo`
	`%lu`
	`%lx` or `%lX`
		64-bit value

Integer	`%hd` or `%hi`
	`%ho`
	`%hu`
		16-bit value

Input Value Type	Subtype and Conversion Character	Output Value Type
	`%hx` or `%hX`	

Text Before or After Formatting Operators

`formatSpec` can also include additional text before a percent sign, `%`, or after a conversion character. The text can be:

- Ordinary text to print.
- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in `formatSpec`.

Special Character	Representation
Single quotation mark	`''`
Percent character	`%%`
Backslash	`\\`
Alarm	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Character whose ASCII code is the hexadecimal number, `N`	`\xN`
Character whose ASCII code is the octal number, `N`	`\N`

Notable Behavior of Conversions with Formatting Operators

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a string conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.

  Example: `'%s'` converts `pi` to `3.141593e+00`.

- If you apply a string conversion (`%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters.

  Example: `'%s'` converts `[65 66 67]` to `ABC`.

`A1,...,An` — Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Output Arguments

`str` — Formatted text

string

Formatted text, returned as a string.

`errmsg` — Error message

string

Error message, returned as a string, when the operation is unsuccessful. Otherwise, `errmsg` is empty.

More About

Tips

- The `sprintf` function is similar to `fprintf`, but `fprintf` prints to a file or to the Command Window.
- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

- “Formatting Strings”

References

[1] Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: "Programming Language C," ANSI, 1430 Broadway, New York, NY 10018.

See Also

`char` | `fprintf` | `fscanf` | `int2str` | `num2str` | `sscanf`

Introduced before R2006a

spy

Visualize sparsity pattern

Syntax

```
spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',markersize)
```

Description

`spy(S)` plots the sparsity pattern of any matrix `S`.

`spy(S,markersize)`, where `markersize` is an integer, plots the sparsity pattern using markers of the specified point size.

`spy(S,'LineSpec')`, where `LineSpec` is a string, uses the specified plot marker type and color.

`spy(S,'LineSpec',markersize)` uses the specified type, color, and size for the plot markers.

`S` is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

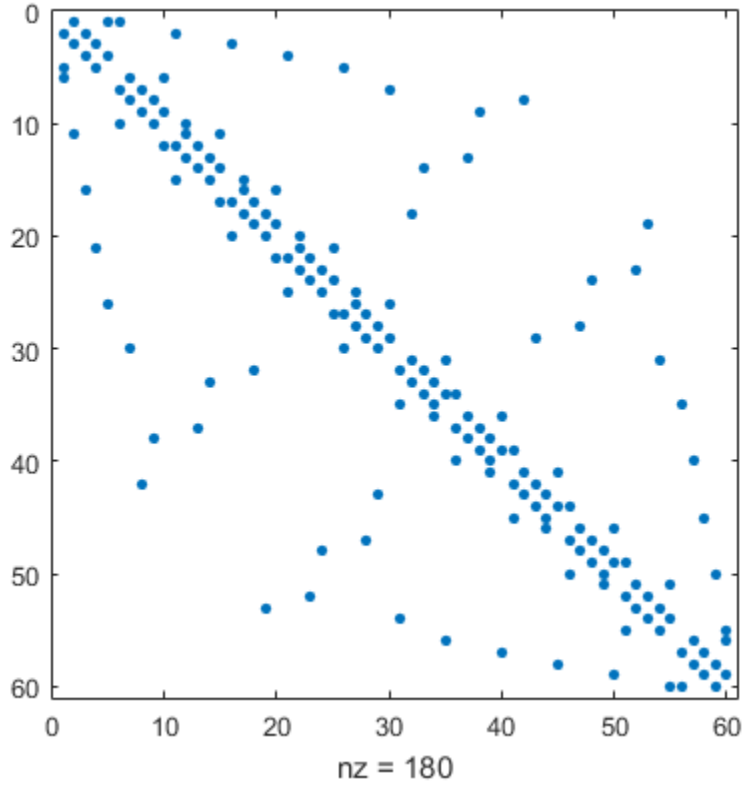
Note `spy` replaces `format +`, which takes much more space to display essentially the same information.

Examples

Plot Sparsity Pattern

This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

```
B = bucky;
spy(B)
```



See Also

`find` | `gplot` | `LineSpec` | `symamd` | `symrcm`

Introduced before R2006a

sqrt

Square root

Syntax

```
B = sqrt(X)
```

Description

`B = sqrt(X)` returns the square root of each element of the array `X`. For the elements of `X` that are negative or complex, `sqrt(X)` produces complex results.

Examples

```
sqrt((-2:2)')
ans =
      0 + 1.4142i
      0 + 1.0000i
      0
 1.0000
 1.4142
```

More About

Tips

See `sqrtm` for the matrix square root.

See Also

`nthroot` | `sqrtm` | `realsqrt`

Introduced before R2006a

sqrtm

Matrix square root

Syntax

```
X = sqrtm(A)
[X, resnorm] = sqrtm(A)
[X, alpha, condest] = sqrtm(A)
```

Description

`X = sqrtm(A)` is the principal square root of the matrix `A`, i.e. `X*X = A`.

`X` is the unique square root for which every eigenvalue has nonnegative real part. If `A` has any eigenvalues with negative real parts then a complex result is produced. If `A` is singular then `A` may not have a square root. A warning is printed if exact singularity is detected.

`[X, resnorm] = sqrtm(A)` does not print any warning, and returns the residual, `norm(A-X^2,'fro')/norm(A,'fro')`.

`[X, alpha, condest] = sqrtm(A)` returns a stability factor `alpha` and an estimate `condest` of the matrix square root condition number of `X`. The residual `norm(A-X^2,'fro')/norm(A,'fro')` is bounded approximately by `n*alpha*eps` and the Frobenius norm relative error in `X` is bounded approximately by `n*alpha*condest*eps`, where `n = max(size(A))`.

Examples

Example 1

A matrix representation of the fourth difference operator is

```
A =
     5    -4     1     0     0
```

```
    -4     6    -4     1     0
     1    -4     6    -4     1
     0     1    -4     6    -4
     0     0     1    -4     5
```

This matrix is symmetric and positive definite. Its unique positive definite square root, `Y = sqrtm(A)`, is a representation of the second difference operator.

```
 Y =
     2    -1    -0    -0    -0
    -1     2    -1     0    -0
     0    -1     2    -1     0
    -0     0    -1     2    -1
    -0    -0    -0    -1     2
```

Example 2

The matrix

```
A =
     7    10
    15    22
```

has four square roots. Two of them are

```
Y1 =
    1.5667    1.7408
    2.6112    4.1779
```

and

```
Y2 =
     1     2
     3     4
```

The other two are `-Y1` and `-Y2`. All four can be obtained from the eigenvalues and vectors of `A`.

```
[V,D] = eig(A);
D =
    0.1386         0
         0   28.8614
```

The four square roots of the diagonal matrix `D` result from the four choices of sign in

```
S =
    ±0.3723          0
          0    ±5.3723
```

All four Ys are of the form

```
Y = V*S/V
```

The `sqrtm` function chooses the two plus signs and produces `Y1`, even though `Y2` is more natural because its entries are integers.

More About

Tips

If `A` is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like `A = [0 1; 0 0]`, do not have any square roots, real or complex, and `sqrtm` cannot be expected to produce one.

See Also

`expm` | `funm` | `logm`

Introduced before R2006a

squeeze

Remove singleton dimensions

Syntax

```
B = squeeze(A)
```

Description

`B = squeeze(A)` returns an array `B` with the same elements as `A`, but with all singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. Two-dimensional arrays are unaffected by `squeeze`; if `A` is a row or column vector or a scalar (1-by-1) value, then `B = A`.

Examples

Remove Singleton Dimension from Array

Create a 2-by-1-by-3 array and remove the singleton column dimension to form a 2-by-3 matrix.

```
y = rand(2,1,3)
z = squeeze(y)
```

```
y(:,:,1) =

    0.8147
    0.9058


y(:,:,2) =

    0.1270
    0.9134
```

```
y(:,:,3) =

    0.6324
    0.0975


z =

    0.8147    0.1270    0.6324
    0.9058    0.9134    0.0975
```

Create Vector from Array with Singleton Dimensions

Create a 1-by-1-by-5 array of ones.

```
mat = repmat(1,[1,1,5])

mat(:,:,1) =

     1


mat(:,:,2) =

     1


mat(:,:,3) =

     1


mat(:,:,4) =

     1


mat(:,:,5) =

     1
```

Condense the data in the third dimension to create a 5-by-1 column vector.

```
squeeze(mat)

ans =

     1
     1
     1
     1
     1
```

See Also

reshape | shiftdim | permute

Introduced before R2006a

ss2tf

Convert state-space representation to transfer function

Syntax

```
[b,a] = ss2tf(A,B,C,D)
[b,a] = ss2tf(A,B,C,D,ni)
```

Description

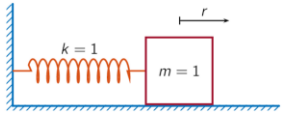
`[b,a] = ss2tf(A,B,C,D)` converts a state-space representation of a system into an equivalent transfer function. `ss2tf` returns the Laplace-transform transfer function for continuous-time systems and the Z-transform transfer function for discrete-time systems.

`[b,a] = ss2tf(A,B,C,D,ni)` returns the transfer function that results when the `ni`th input of a system with multiple inputs is excited by a unit impulse.

Examples

Mass-Spring System

A one-dimensional discrete-time oscillating system consists of a unit mass, m, attached to a wall by a spring of unit elastic constant. A sensor samples the acceleration, a, of the mass at $F_s = 5$ Hz.



Generate 50 time samples. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 5;
```

```
dt = 1/Fs;
N = 50;
t = dt*(0:N-1);
```

The oscillator can be described by the state-space equations

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = (r \quad v)^T$ is the state vector, r and v are respectively the position and velocity of the mass, and the matrices

$$A = \begin{pmatrix} \cos\Delta t & \sin\Delta t \\ -\sin\Delta t & \cos\Delta t \end{pmatrix}, \quad B = \begin{pmatrix} 1-\cos\Delta t \\ \sin\Delta t \end{pmatrix}, \quad C = (-1 \quad 0), \quad D = (1).$$

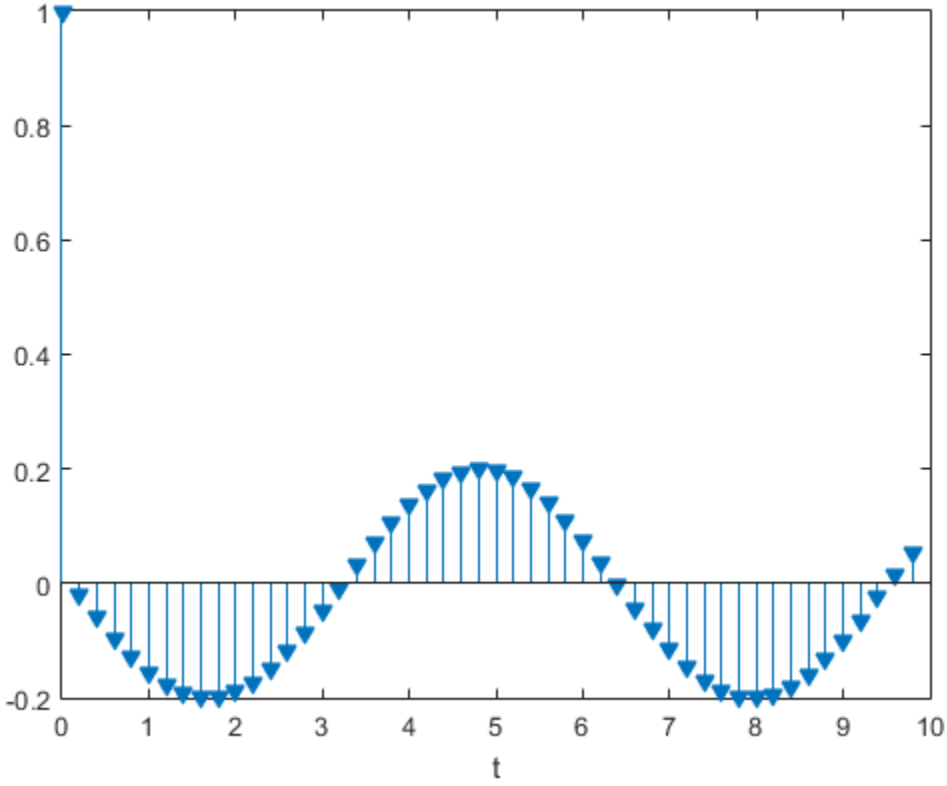
```
A = [cos(dt) sin(dt);-sin(dt) cos(dt)];
B = [1-cos(dt);sin(dt)];
C = [-1 0];
D = 1;
```

The system is excited with a unit impulse in the positive direction. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state.

```
u = [1 zeros(1,N-1)];

x = [0;0];
for k = 1:N
    y(k) = C*x + D*u(k);
    x = A*x + B*u(k);
end
```

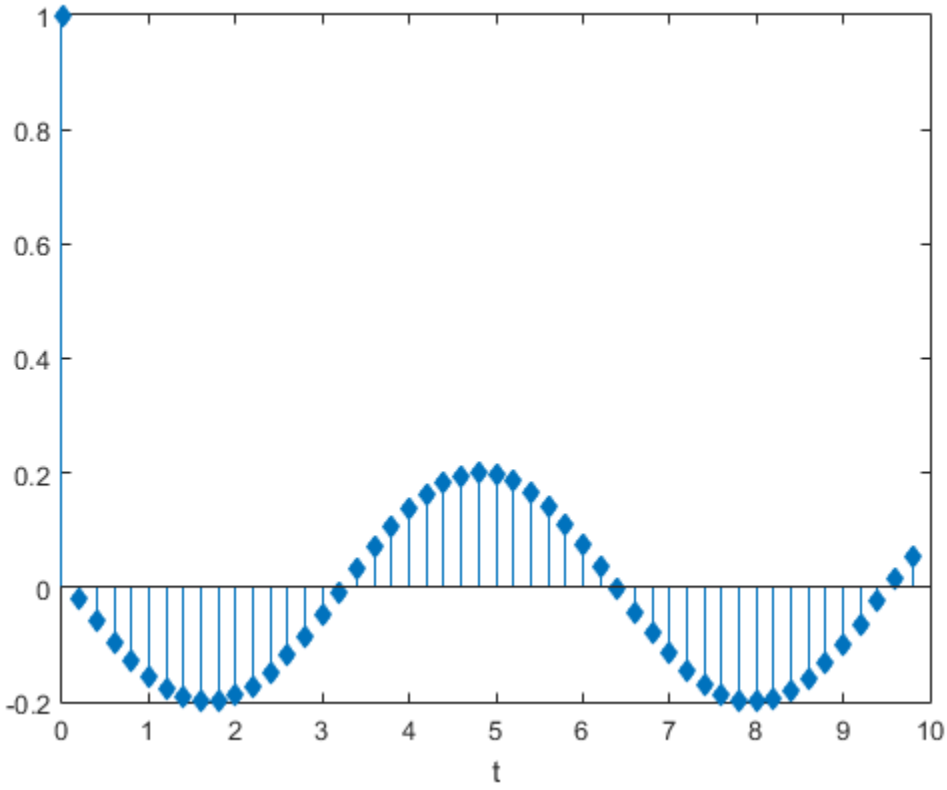
Plot the acceleration of the mass as a function of time.

```
stem(t,y,'v','filled')
xlabel('t')
```



Compute the time-dependent acceleration using the transfer function $H(z)$ to filter the input. Plot the result.

```
[b,a] = ss2tf(A,B,C,D);
yt = filter(b,a,u);
stem(t,yt,'d','filled')
xlabel('t')
```

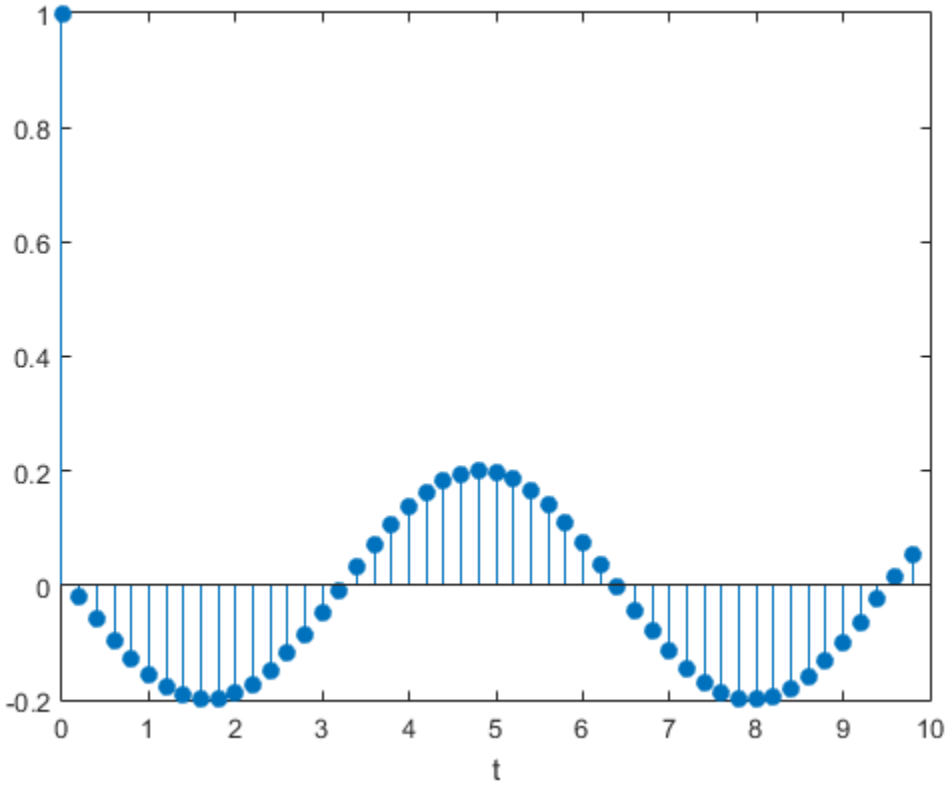


The transfer function of the system has an analytic expression:

$$H(z) = \frac{1 - z^{-1}(1 + \cos \Delta t) + z^{-2} \cos \Delta t}{1 - 2z^{-1} \cos \Delta t + z^{-2}}.$$

Use the expression to filter the input. Plot the response.

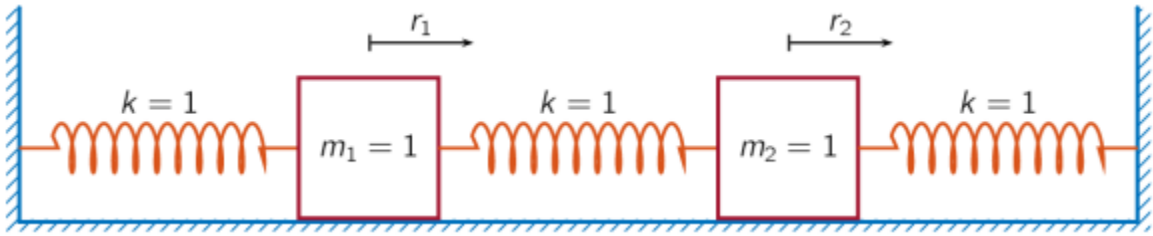
```
bf = [1 -(1+cos(dt)) cos(dt)];
af = [1 -2*cos(dt) 1];
yf = filter(bf,af,u);
stem(t,yf,'o','filled')
xlabel('t')
```



The result is the same in all three cases.

Two-Body Oscillator

An ideal one-dimensional oscillating system consists of two unit masses, m_1 and m_2, confined between two walls. Each mass is attached to the nearest wall by a spring of unit elastic constant. Another such spring connects the two masses. Sensors sample a_1 and a_2, the accelerations of the masses, at $F_s = 16$ Hz.



Specify a total measurement time of 16 s. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 16;
dt = 1/Fs;
N = 257;
t = dt*(0:N-1);
```

The system can be described by the state-space model

$$x(n+1) = Ax(n) + Bu(n),$$
$$y(n) = Cx(n) + Du(n),$$

where $x = (r_1 \quad v_1 \quad r_2 \quad v_2)^T$ is the state vector and r_i and v_i are respectively the location and the velocity of the i-th mass. The input vector $u = (u_1 \quad u_2)^T$ and the output vector $y = (a_1 \quad a_2)^T$. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = \begin{pmatrix} -2 & 0 & 1 & 0 \\ 1 & 0 & -2 & 0 \end{pmatrix}, \quad D = I,$$

the continuous-time state-space matrices are

$$A_c = \begin{pmatrix} 0 & 1 & 0 & 0 \\ -2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & -2 & 0 \end{pmatrix}, \quad B_c = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix},$$

and I denotes an identity matrix of the appropriate size.

```
Ac = [0 1 0 0;-2 0 1 0;0 0 0 1;1 0 -2 0];
A = expm(Ac*dt);
Bc = [0 0;1 0;0 0;0 1];
B = Ac\(A-eye(4))*Bc;
C = [-2 0 1 0;1 0 -2 0];
D = eye(2);
```

The first mass, m_1, receives a unit impulse in the positive direction.

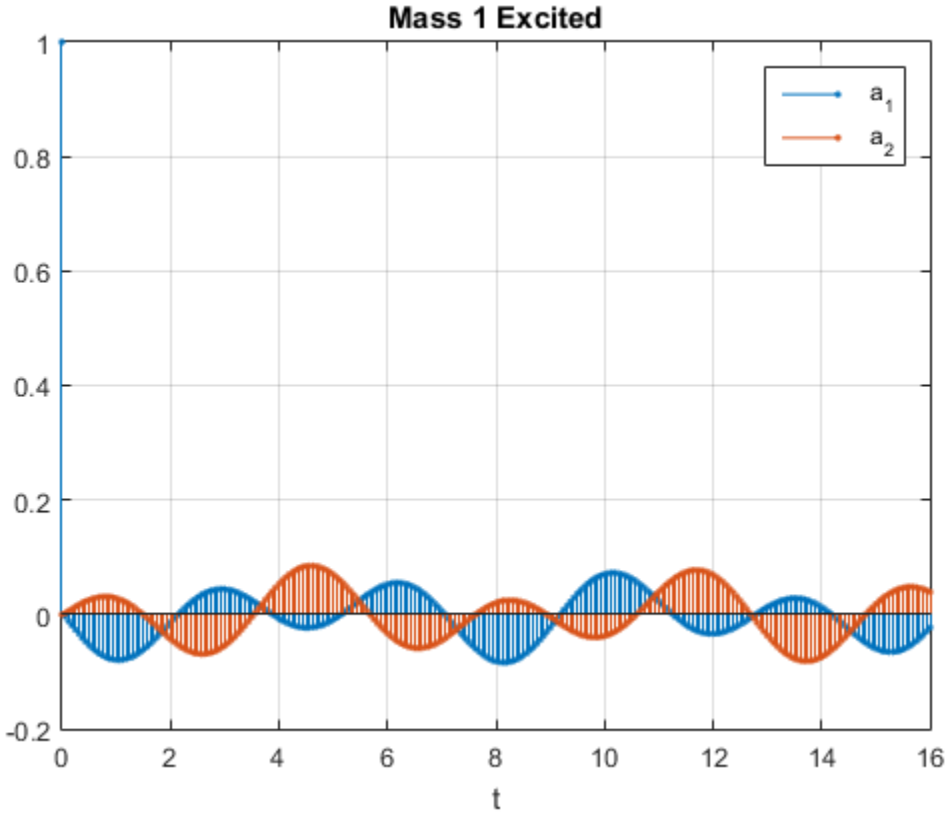
```
ux = [1 zeros(1,N-1)];
u0 = zeros(1,N);
u = [ux;u0];
```

Use the model to compute the time evolution of the system starting from an all-zero initial state.

```
x = [0;0;0;0];
for k = 1:N
    y(:,k) = C*x + D*u(:,k);
    x = A*x + B*u(:,k);
end
```

Plot the accelerations of the two masses as functions of time.

```
stem(t,y','.')
xlabel('t')
legend('a_1','a_2')
title('Mass 1 Excited')
grid
```

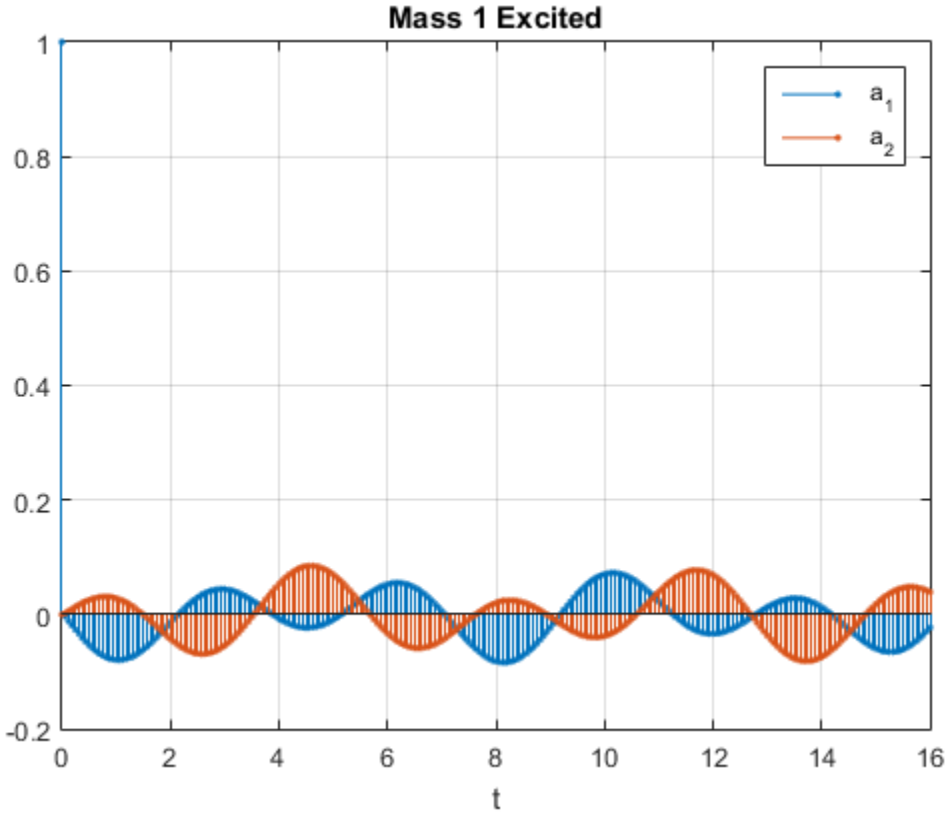


Convert the system to its transfer function representation. Find the response of the system to a positive unit impulse excitation on the first mass.

```
[b1,a1] = ss2tf(A,B,C,D,1);
y1u1 = filter(b1(1,:),a1,ux);
y1u2 = filter(b1(2,:),a1,ux);
```

Plot the result. The transfer function gives the same response as the state-space model.

```
stem(t,[y1u1;y1u2]','.')
xlabel('t')
legend('a_1','a_2')
title('Mass 1 Excited')
grid
```



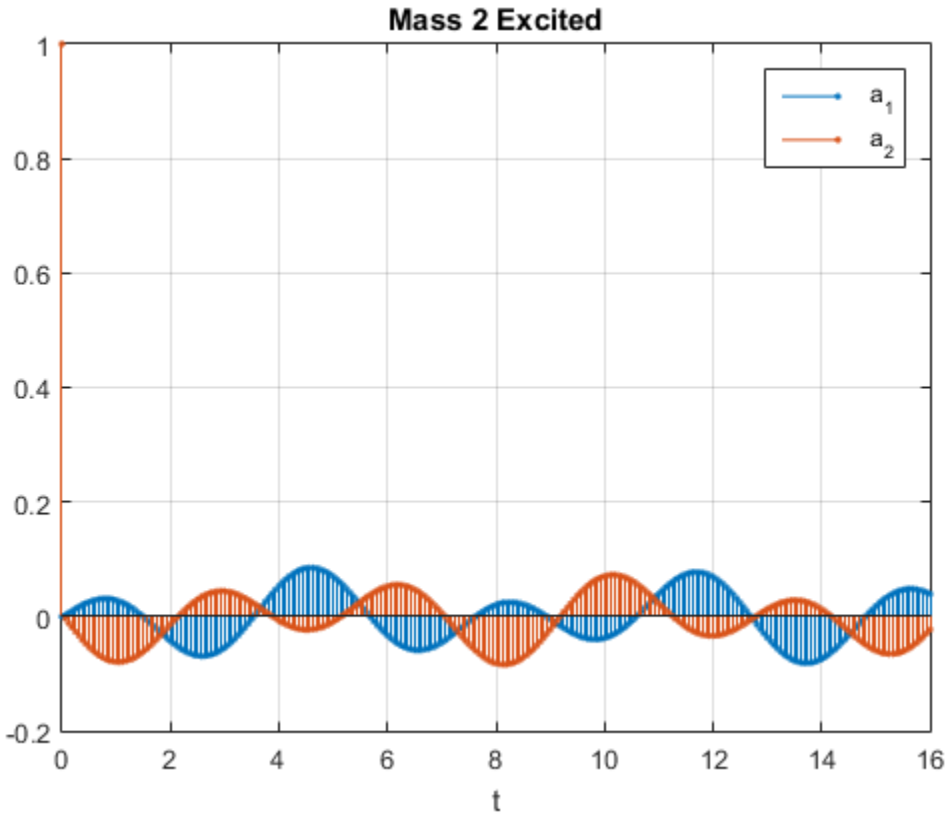
The system is reset to its initial configuration. Now the other mass, m_2, receives a unit impulse in the positive direction. Compute the time evolution of the system.

```
u = [u0;ux];

x = [0;0;0;0];
for k = 1:N
    y(:,k) = C*x + D*u(:,k);
    x = A*x + B*u(:,k);
end
```

Plot the accelerations. The responses of the individual masses are switched.

```
stem(t,y','.')
```

```
xlabel('t')
legend('a_1','a_2')
title('Mass 2 Excited')
grid
```



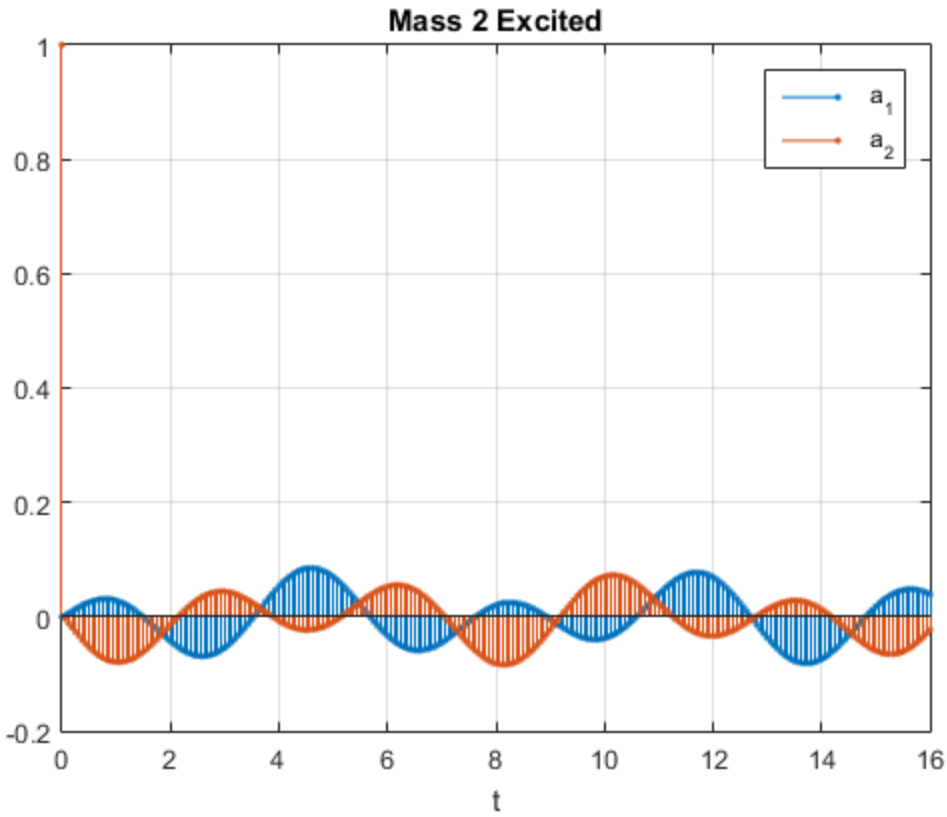
Find the response of the system to a positive unit impulse excitation on the second mass.

```
[b2,a2] = ss2tf(A,B,C,D,2);
y2u1 = filter(b2(1,:),a2,ux);
y2u2 = filter(b2(2,:),a2,ux);
```

Plot the result. The transfer function gives the same response as the state-space model.

```
stem(t,[y2u1;y2u2]','.')
```

```
xlabel('t')
legend('a_1','a_2')
title('Mass 2 Excited')
grid
```



Input Arguments

A — State matrix

matrix

State matrix, specified as a matrix. If the system has p inputs and q outputs and is described by n state variables, then A is n-by-n.

Data Types: `single` | `double`

`B` — Input-to-state matrix
matrix

Input-to-state matrix, specified as a matrix. If the system has p inputs and q outputs and is described by n state variables, then B is n-by-p.

Data Types: `single` | `double`

`C` — State-to-output matrix
matrix

State-to-output matrix, specified as a matrix. If the system has p inputs and q outputs and is described by n state variables, then C is q-by-n.

Data Types: `single` | `double`

`D` — Feedthrough matrix
matrix

Feedthrough matrix, specified as a matrix. If the system has p inputs and q outputs and is described by n state variables, then D is q-by-p.

Data Types: `single` | `double`

`ni` — Input index
1 (default) | integer scalar

Input index, specified as an integer scalar. If the system has p inputs, use `ss2tf` with a trailing argument ni = 1, …, p to compute the response to a unit impulse applied to the nith input.

Data Types: `single` | `double`

Output Arguments

`b` — Transfer function numerator coefficients
vector | matrix

Transfer function numerator coefficients, returned as a vector or matrix. If the system has p inputs and q outputs and is described by n state variables, then b is q-by-$(n + 1)$ for each input. The coefficients are returned in descending powers of s or z.

a — Transfer function denominator coefficients

vector

Transfer function denominator coefficients, returned as a vector. If the system has p inputs and q outputs and is described by n state variables, then a is 1-by-(n + 1) for each input. The coefficients are returned in descending powers of s or z.

More About

Transfer Function

- For discrete-time systems, the state-space matrices relate the state vector x, the input u, and the output y through

  $$x(k+1) = \mathrm{A}\,x(k) + \mathrm{B}\,u(k)$$
  $$y(k) = \mathrm{C}\,x(k) + \mathrm{D}\,u(k).$$

  The transfer function is the Z-transform of the system's impulse response. It can be expressed in terms of the state-space matrices as

  $$\mathrm{H}(z) = \mathrm{C}(z\mathrm{I} - \mathrm{A})^{-1}\mathrm{B} + \mathrm{D}.$$

- For continuous-time systems, the state-space matrices relate the state vector x, the input u, and the output y through

  $$\dot{x} = \mathrm{A}\,x + \mathrm{B}\,u$$
  $$y = \mathrm{C}\,x + \mathrm{D}\,u.$$

  The transfer function is the Laplace transform of the system's impulse response. It can be expressed in terms of the state-space matrices as

  $$\mathrm{H}(s) = \mathrm{C}(s\mathrm{I} - \mathrm{A})^{-1}\mathrm{B} + \mathrm{D}.$$

See Also

`latc2tf` | `sos2tf` | `ss2sos` | `ss2zp` | `tf2ss` | `zp2tf`

Introduced before R2006a

sscanf

Read formatted data from string

Syntax

```
A = sscanf(str, format)
A = sscanf(str, format, sizeA)
[A, count] = sscanf(...)
[A, count, errmsg] = sscanf(...)
[A, count, errmsg, nextindex] = sscanf(...)
```

Description

`A = sscanf(str, format)` reads data from string `str`, converts it according to the `format`, and returns the results in array `A`. The `sscanf` function reapplies the `format` until either reaching the end of `str` or failing to match the `format`. If `sscanf` cannot match the `format` to the data, it reads only the portion that matches into `A` and stops processing. If `str` is a character array with more than one row, `sscanf` reads the characters in column order.

`A = sscanf(str, format, sizeA)` reads `sizeA` elements into `A`, where `sizeA` can be an integer or can have the form `[m,n]`.

`[A, count] = sscanf(...)` returns the number of elements that `sscanf` successfully reads.

`[A, count, errmsg] = sscanf(...)` returns an error message string when the operation is unsuccessful. Otherwise, `errmsg` is an empty string.

`[A, count, errmsg, nextindex] = sscanf(...)` returns one more than the number of characters scanned in `str`.

Input Arguments

`format`

String enclosed in single quotation marks that describes each type of element (field). Includes one or more of the following specifiers.

Field Type	Specifier	Details
Integer, signed	`%d`	Base 10
	`%i`	Base determined from the values. Defaults to base 10. If initial digits are `0x` or `0X`, it is base 16. If initial digit is `0`, it is base 8.
	`%ld` or `%li`	64-bit values, base 10, 8, or 16
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal)
	`%lu`, `%lo`, `%lx`	64-bit values, base 10, 8, or 16
Floating-point number	`%f`	Floating-point fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`.
	`%e`	
	`%g`	
Character string	`%s`	Read series of characters, until find white space.
	`%c`	Read any single character, including white space. (To read multiple characters, specify field length.)
	`%[...]`	Read only characters in the brackets, until the first nonmatching character or white space.

Optionally:

- To skip fields, insert an asterisk (`*`) after the percent sign (`%`). For example, to skip integers, specify `%*d`.
- To specify the maximum width of a field, insert a number. For example, `%10c` reads exactly 10 characters at a time, including white space.

- To skip a specific set of characters, insert the literal characters in the *format*. For example, to read only the floating-point number from `'pi=3.14159'`, specify a *format* of `'pi=%f'`.

sizeA

Dimensions of the output array *A*. Specify in one of the following forms:

`inf`	Read to the end of the input string. (default)
n	Read at most *n* elements.
`[`*m*`,`*n*`]`	Read at most *m***n* elements in column order. *n* can be `inf`, but *m* cannot.

When the *format* includes `%s`, *A* can contain more than *n* columns. *n* refers to elements, not characters.

str

Character string.

Output Arguments

A

An array. If the *format* includes:

- Only numeric specifiers, *A* is numeric. If *format* includes only 64-bit signed integer specifiers, *A* is of class `int64`. Similarly, if *format* includes only 64-bit unsigned integer specifiers, *A* is of class `uint64`. Otherwise, *A* is of class `double`. If *sizeA* is `inf` or *n*, then *A* is a column vector. If the input contains fewer than *sizeA* elements, MATLAB pads *A* with zeros.
- Only character or string specifiers (`%c` or `%s`), *A* is a character array. If *sizeA* is `inf` or *n*, *A* is a row vector. If the input contains fewer than *sizeA* characters, MATLAB pads *A* with `char(0)`.
- A combination of numeric and character specifiers, *A* is numeric, of class `double`. MATLAB converts each character to its numeric equivalent. This conversion occurs even when the *format* explicitly skips all numeric values (for example, a *format* of `'%*d %s'`).

If MATLAB cannot match the input to the *format*, and the *format* contains both numeric and character specifiers, *A* can be numeric or character. The class of *A* depends on the values MATLAB reads before processing stops.

count

Number of elements sscanf reads into *A*.

errmsg

An error message string when sscanf cannot open the specified file. Otherwise, an empty string.

nextindex

sscanf counts the number of characters sscanf reads from *str*, and then adds one.

Examples

Example 1

Read multiple floating-point values from a string:

```
s = '2.7183  3.1416';
A = sscanf(s,'%f')
A =
    2.7183
    3.1416
```

Example 2

Read an octal integer from a string, identified by the '0' prefix, using %i to preserve the sign:

```
sscanf('-010','%i')
ans =
    -8
```

Example 3

Read numeric values from a two-dimensional character array. By default, `sscanf` reads characters in column order. To preserve the original order of the values, read one row at a time.

```
mixed = ['abc 45 6 ghi'; 'def 7 89 jkl'];

[nrows, ncols] = size(mixed);
for k = 1:nrows
    nums(k,:) = sscanf(mixed(k,:), '%*s %d %d %*s', [1, inf]);
end;

% type the variable name to see the result
nums =
    45     6
     7    89
```

Example 4

`sscanf` finds one match for `%s`

```
[str count] = sscanf('ThisIsOneString', '%s')
str =
    ThisIsOneString
count =
     1
```

`sscanf` finds four matches for `%s`. Because it does not match space characters, there are no spaces in the output string:

```
[str count] = sscanf('These Are Four Strings', '%s')
str =
    TheseAreFourStrings
count =
     4
```

`sscanf` finds five word matches for `%s` and four space character matches for `%c`. Because the `%c` specifier does match a space character, the output string does include spaces:

```
[str count] = sscanf('Five strings and four spaces', '%s%c')
str =
    Five strings and four spaces
```

```
count =
    9
```

`sscanf` finds three word matches for `%s` and two numeric matches for `%d`. Because the format specifier has a mixed %d and %s format, sscanf converts all nonnumeric characters to numeric:

```
[str count] = sscanf('5 strings and 4 spaces', '%d%s%s%d%s');
str'
  Columns 1 through 9
     5   115   116   114   105   110   103   115    97
  Columns 10 through 18
   110   100     4   115   112    97    99   101   115
count
count =
     5
```

Example 5

```
[str, count] = sscanf('one two three', '%c')
str =
    one two three
count =
    13

[str, count] = sscanf('one two three', '%13c')
str =
    one two three
count =
     1


[str, count] = sscanf('one two three', '%s')
str =
    onetwothree
count =
     3

[str, count] = sscanf('one two three', '%1s')
str =
    onetwothree
count =
    11
```

Example 6

```
tempString = '78°F 72°F 64°F 66°F 49°F';

degrees = char(176);
tempNumeric = sscanf(tempString, ['%d' degrees 'F'])'
tempNumeric =
    78    72    64    66    49
```

More About

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

See Also

`fscanf` | `sprintf` | `textscan`

Introduced before R2006a

stack

Stack data from multiple variables into single variable

Syntax

```
T = stack(W,vars)
T = stack(W,vars,Name,Value)
[T,iw] = stack(___)
```

Description

`T = stack(W,vars)` converts the wide table, `W`, into an equivalent table, `T`, that is in tall format. `stack` stacks up multiple variables from `W`, specified by `vars`, into a single variable in `T`. In general, `T` contains fewer variables, but more rows, than `W`.

The output table, `T`, contains a new categorical variable to indicate which variable in `W` the stacked data in each row came from. `stack` replicates data from the variables in `W` that are not stacked.

`T = stack(W,vars,Name,Value)` converts the table, `W`, to tall format with additional options specified by one or more Name,Value pair arguments.

For example, you can specify variable names for the new and stacked variables in `W`.

`[T,iw] = stack(___)` also returns an index vector, iw, indicating the correspondence between rows in `T` and rows in `W`. You can use any of the previous input arguments.

Examples

Stack Three Variables into One

Create a table containing test scores from three separate tests.

```
Test1 = [93;57;87;89];
Test2 = [89;77;92;86];
Test3 = [95;62;89;91];
```

```
W = table(Test1,Test2,Test3)

W =

    Test1    Test2    Test3
    _____    _____    _____

    93       89       95
    57       77       62
    87       92       89
    89       86       91
```

The table contains four rows and three variables.

Stack the test scores into a single variable.

```
T = stack(W,1:3)

T =

    Test1_Test2_Test3_Indicator    Test1_Test2_Test3
    _______________________________    _________________

    Test1                              93
    Test2                              89
    Test3                              95
    Test1                              57
    Test2                              77
    Test3                              62
    Test1                              87
    Test2                              92
    Test3                              89
    Test1                              89
    Test2                              86
    Test3                              91
```

`T` contains twelve rows and two variables.

The categorical variable, `Test1_Test2_Test3_Indicator`, identifies which test corresponds to the score in the stacked data variable, `Test1_Test2_Test3`.

Stack Variables and Specify Variable Names

Create a table indicating the amount of snowfall at three locations from five separate storms.

```
Storm = [1;2;3;4;5];
Date = {'12/25/11';'1/2/12';'1/23/12';'2/7/12';'2/15/12'};
Natick = [20;5;13;0;17];
Boston = [18;9;21;5;12];
Worcester = [26;10;16;3;15];

W = table(Storm,Date,Natick,Boston,Worcester)
```

```
W =

    Storm       Date       Natick    Boston    Worcester
    _____    __________    ______    ______    _________

    1        '12/25/11'    20        18        26
    2        '1/2/12'       5         9        10
    3        '1/23/12'     13        21        16
    4        '2/7/12'       0         5         3
    5        '2/15/12'     17        12        15
```

The variables `Storm` and `Date` contain data that is constant at each location.

Stack the variables `Natick`, `Boston`, and `Worcester` into a single variable. Name the variable containing the stacked data, `Snowfall`, and name the new indicator variable, `Town`.

```
T = stack(W,{'Natick','Boston','Worcester'},...
    'NewDataVariableName','Snowfall',...
    'IndexVariableName','Town')
```

```
T =

    Storm       Date         Town       Snowfall
    _____    __________    _________    ________

    1        '12/25/11'    Natick       20
    1        '12/25/11'    Boston       18
    1        '12/25/11'    Worcester    26
    2        '1/2/12'      Natick        5
    2        '1/2/12'      Boston        9
    2        '1/2/12'      Worcester    10
    3        '1/23/12'     Natick       13
    3        '1/23/12'     Boston       21
    3        '1/23/12'     Worcester    16
    4        '2/7/12'      Natick        0
    4        '2/7/12'      Boston        5
```

```
    4         '2/7/12'       Worcester      3
    5         '2/15/12'      Natick        17
    5         '2/15/12'      Boston        12
    5         '2/15/12'      Worcester     15
```

`T` contains three rows for each storm, and `stack` repeats the data in the constant variables, `Storm` and `Date`, accordingly.

The categorical variable, `Town`, identifies which variable in `W` contains the corresponding `Snowfall` data.

Stack Variables and Output an Index Vector

Create a table containing estimated influenza rates along the east coast of the United States. Create a different variable for the Northeast, Mid Atlantic, and South Atlantic. Data Source: Google Flu Trends (http://www.google.org/flutrends).

```
Month = {'October';'November';'December';...
    'January';'February';'March'};
Year = [2005*ones(3,1); 2006*ones(3,1)];
NE = [1.1902; 1.3610; 1.5003; 1.7772; 2.1350; 2.2345];
MidAtl = [1.1865; 1.4120; 1.6043; 1.8830; 2.1227; 1.9920];
SAtl = [1.2730; 1.5820; 1.8625; 1.9540; 2.4803; 2.0203];

fluW = table(Month,Year,NE,MidAtl,SAtl)
```

```
fluW =

      Month        Year      NE       MidAtl      SAtl
    ___________    ____    ______    ______    ______

    'October'      2005    1.1902    1.1865     1.273
    'November'     2005     1.361     1.412     1.582
    'December'     2005    1.5003    1.6043    1.8625
    'January'      2006    1.7772     1.883     1.954
    'February'     2006     2.135    2.1227    2.4803
    'March'        2006    2.2345     1.992    2.0203
```

The variables `Month` and `Year` contain data that is constant across the row.

Stack the variables `NE`, `MidAtl`, and `SAtl` into a single variable called `FluRate`. Name the new indicator variable `Region` and output an index vector, `ifluW`, to indicate the correspondence between rows in the input wide table, `fluW`, and the output tall table, `fluT`.

```
[fluT,ifluW] = stack(fluW,3:5,...
    'NewDataVariableName','FluRate',...
    'IndexVariableName','Region')
```

```
fluT =

      Month        Year    Region    FluRate
    ___________    ____    ______    _______

    'October'      2005    NE         1.1902
    'October'      2005    MidAtl     1.1865
    'October'      2005    SAtl        1.273
    'November'     2005    NE          1.361
    'November'     2005    MidAtl      1.412
    'November'     2005    SAtl        1.582
    'December'     2005    NE         1.5003
    'December'     2005    MidAtl     1.6043
    'December'     2005    SAtl       1.8625
    'January'      2006    NE         1.7772
    'January'      2006    MidAtl      1.883
    'January'      2006    SAtl        1.954
    'February'     2006    NE          2.135
    'February'     2006    MidAtl     2.1227
    'February'     2006    SAtl       2.4803
    'March'        2006    NE         2.2345
    'March'        2006    MidAtl      1.992
    'March'        2006    SAtl       2.0203


ifluW =

     1
     1
     1
     2
     2
     2
     3
     3
     3
     4
     4
     4
     5
     5
```

```
5
6
6
6
```

`ifluW(5)` is `2`. The fifth row in the output table, `fluT`, contains data from the second row in the input table `fluW`.

Input Arguments

`W` — Wide table

table

Wide table, specified as a table.

`vars` — Variables in `W` to stack

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables in `W` to stack, specified as a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NewDataVariableName','StackedData'` names the new data variable `StackedData`.

`'ConstantVariables'` — Variables other than `vars` to include in the output

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables other than `vars` to include in the output, specified as the comma-separated pair consisting of `'ConstantVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector. `stack` replicates the data from the constant variables for each stacked entry from a row.

The default is all the variables in W not specified by vars. You can specify the `'ConstantVariables'` name-value pair argument to exclude variables not specified by `vars` or `'ConstantVariables'` from the output table, `T`.

`'NewDataVariableName'` — Name for the new data variable in `T`
string

Name for the new data variable in `T`, specified as the comma-separated pair consisting of `'NewDataVariableName'` and a string. The default is a concatenation of the names of the variables from `W` that are stacked up.

`'IndexVariableName'` — Name for the new indicator variable in `T`
string

Name for the new indicator variable in `T`, specified as the comma-separated pair consisting of `'IndexVariableName'` and a string. The default is a name based on `NewDataVariableName`.

Output Arguments

`T` — Tall table
table

Tall table, returned as a table. `T` contains a stacked data variable, a categorical indicator variable, and any constant variables.

You can store additional metadata such as descriptions, variable units, variable names, and row names in the table. For more information, see `Table Properties`.

`stack` assigns the variable units and variable description property values from the first variable listed in vars to the corresponding `T.Properties.VariableUnits` and `T.Properties.VariableDescrisciptions` values for the new data variable.

`iw` — Index to `W`
column vector

Index to `W`, returned as a column vector. The index vector, `iw`, identifies the row in the input table, `W`, containing the corresponding data. `stack` creates the `j`th row in the output table, `T`, using `W(iw(j),vars)`.

More About

Tips

- You can specify more than one group of data variables in `W`, and each group becomes a stacked data variable in `T`. Use a cell array to contain multiple values for `vars`, and a cell array of strings to contain multiple values for the `'NewDataVariableName'` name-value pair argument. All groups must contain the same number of variables.

See Also

`join` | `unstack`

stairs

Stairstep graph

Syntax

```
stairs(Y)
stairs(X,Y)
stairs( ___ ,LineSpec)
stairs( ___ ,Name,Value)

stairs(ax, ___ )

h = stairs( ___ )

[xb,yb] = stairs( ___ )
```

Description

`stairs(Y)` draws a stairstep graph of the elements in `Y`.

- If `Y` is a vector, then `stairs` draws one line.
- If `Y` is a matrix, then `stairs` draws one line per matrix column.

`stairs(X,Y)` plots the elements in `Y` at the locations specified by `X`. The inputs `X` and `Y` must be vectors or matrices of the same size. Additionally, `X` can be a row or column vector and `Y` must be a matrix with `length(X)` rows.

`stairs( ___ ,LineSpec)` specifies a line style, marker symbol, and color. For example, `':*r'` specifies a dotted red line with asterisk markers. Use this option with any of the input argument combinations in the previous syntaxes.

`stairs( ___ ,Name,Value)` specifies stair properties using one or more `Name,Value` pair arguments. For example, `'Marker','o','MarkerSize',8` specifies 8 point circle markers.

`stairs(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in the previous syntaxes.

`h = stairs( ___ )` returns one or more stair objects. Use `h` to make changes to properties of a specific stair object after it is created.

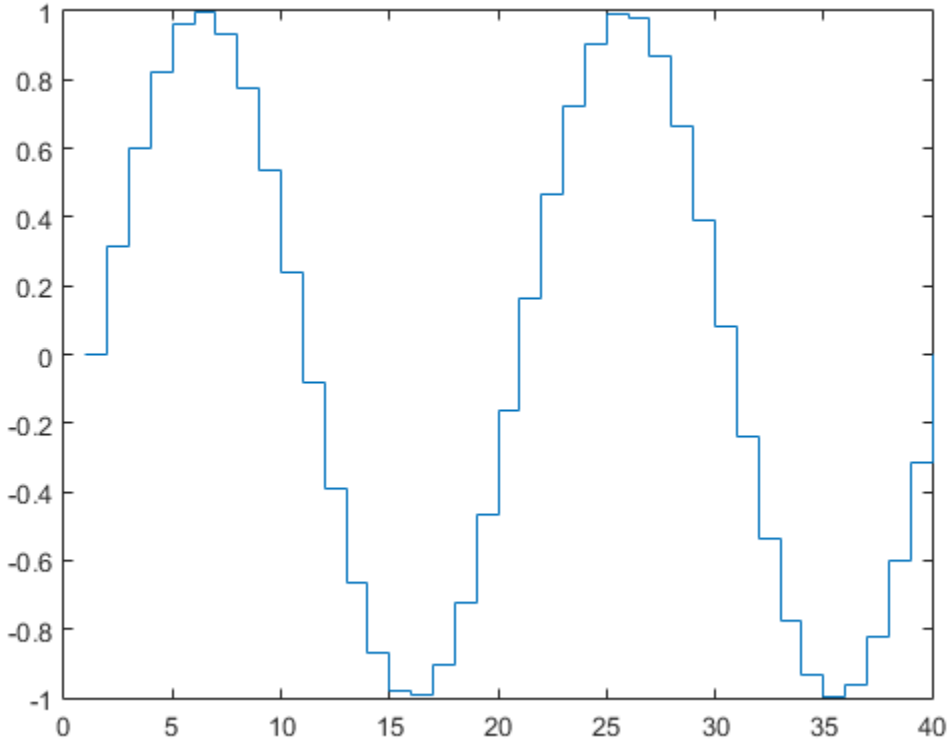
`[xb,yb] = stairs( ___ )` does not create a plot, but returns matrices `xb` and `yb` of the same size, such that `plot(xb,yb)` plots the stairstep graph.

Examples

Plot Single Data Series

Create a stairstep plot of sine evaluated at 40 equally spaced values between 0 and 4π.

```
X = linspace(0,4*pi,40);
Y = sin(X);

figure
stairs(Y)
```

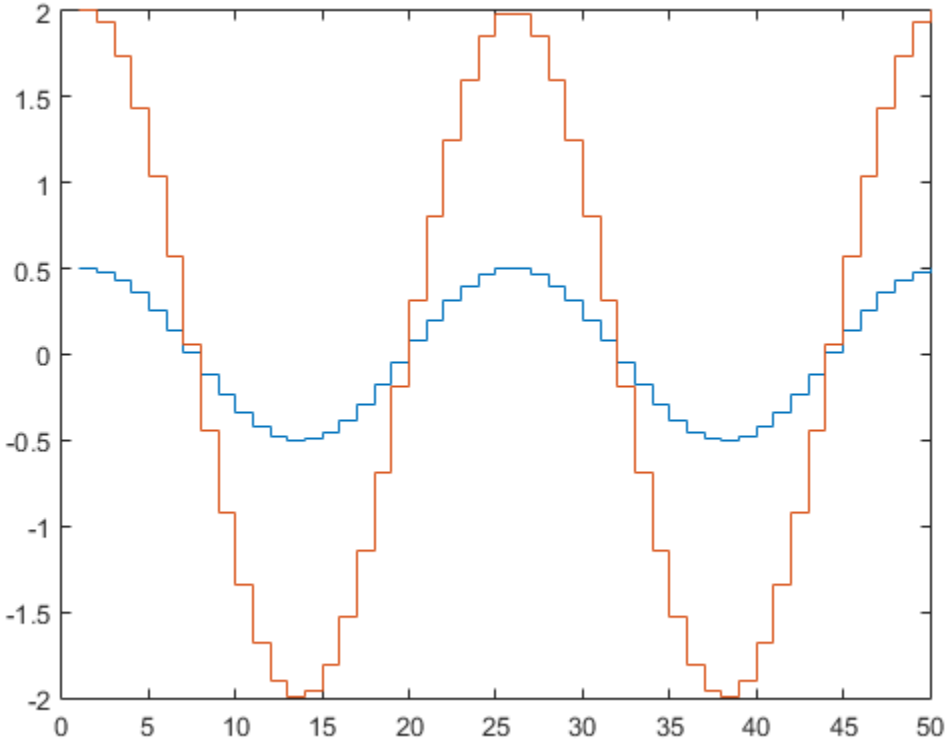


The length of Y automatically determines and generates the *x*-axis scale.

Plot Multiple Data Series

Create a stairstep plot of two cosine functions evaluated at 50 equally spaced values between 0 and 4π.

```
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];

figure
stairs(Y)
```

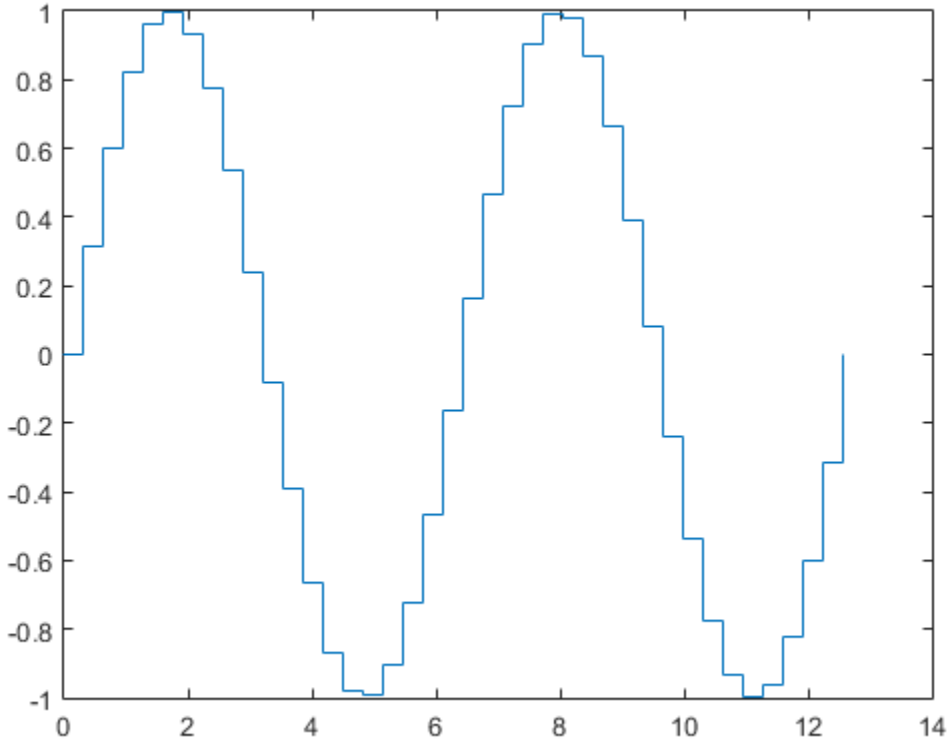


The number of rows in Y automatically determines and generates the *x*-axis scale.

Plot Single Data Series at Specified x-Values

Create a stairstep plot of a sine wave evaluated at equally spaced values between 0 and 4π. Specify the set of *x*-values for the plot.

```
X = linspace(0,4*pi,40);
Y = sin(X);

figure
stairs(X,Y)
```

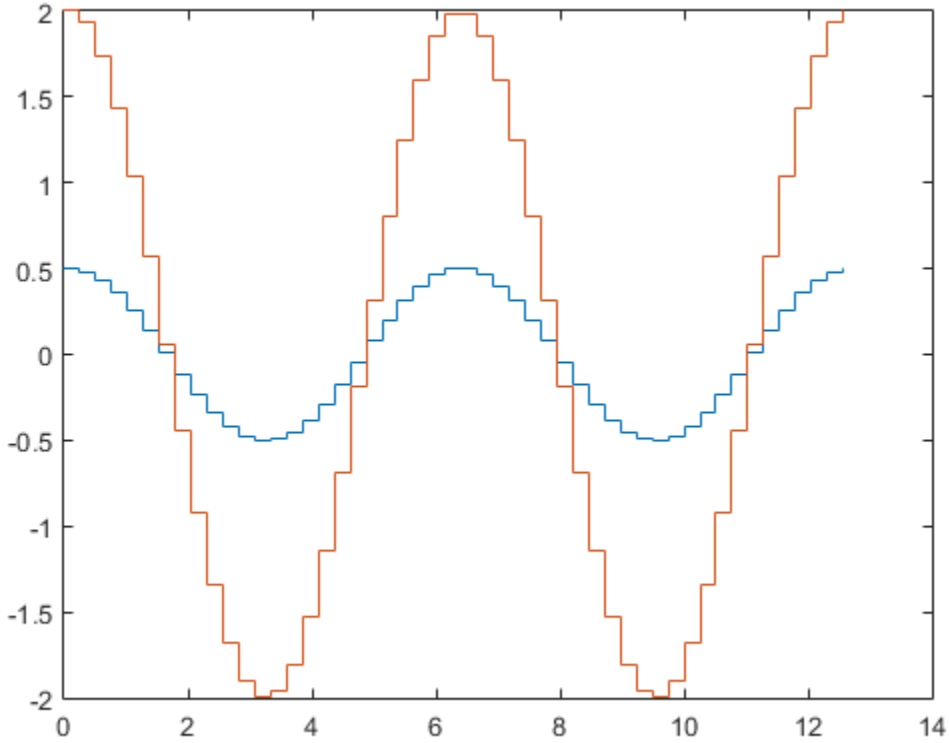


The entries in Y are plotted against the corresponding entries in X.

Plot Multiple Data Series at Specified x-Values

Create a stairstep plot of two cosine waves evaluated at equally spaced values between 0 and 4π. Specify the set of *x*-values for the plot.

```
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];

figure
stairs(X,Y)
```

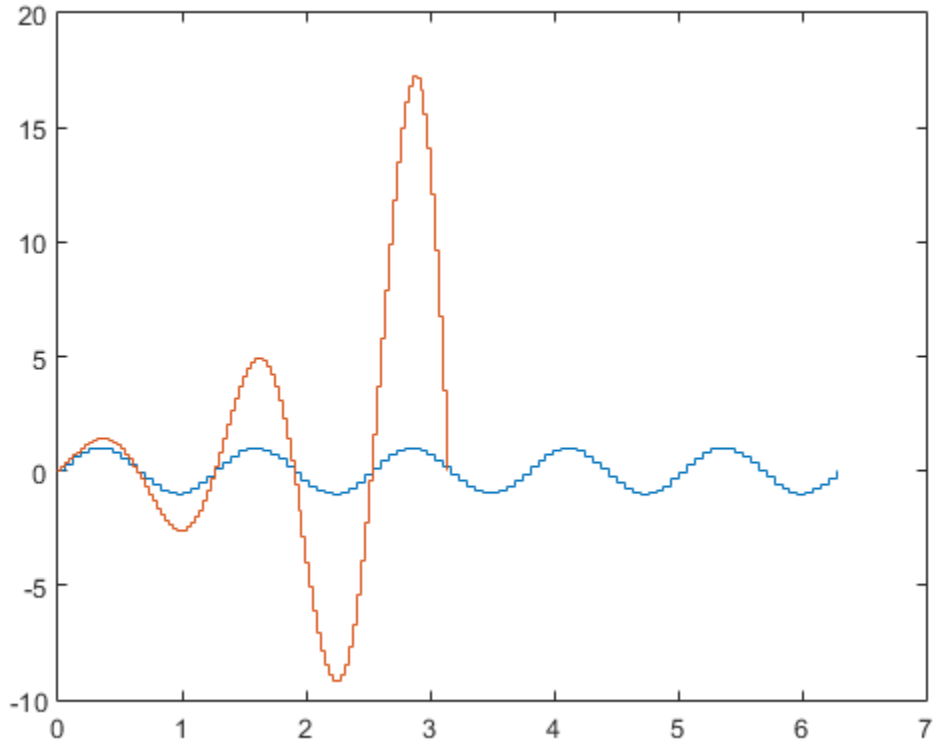


The first vector input, `X`, determines the *x*-axis positions for both data series.

Plot Multiple Data Series at Unique Sets of x-Values

Create a stairstep plot of two sine waves evaluated at different values. Specify a unique set of *x*-values for plotting each data series.

```
x1 = linspace(0,2*pi)';
x2 = linspace(0,pi)';
X = [x1,x2];
Y = [sin(5*x1),exp(x2).*sin(5*x2)];

figure
stairs(X,Y)
```

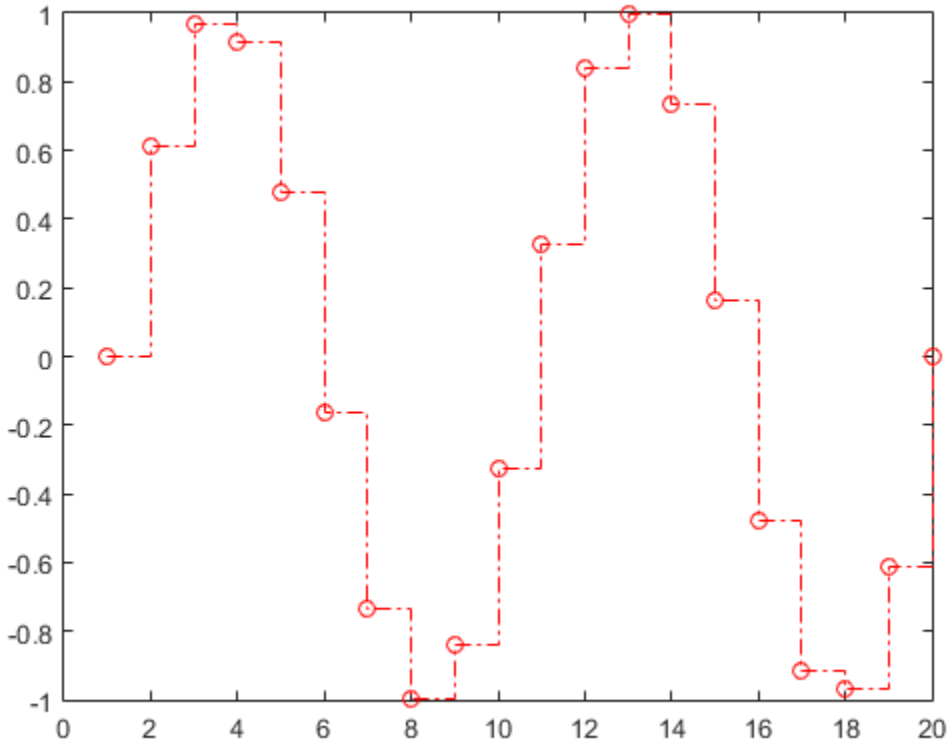


Each column of X is plotted against the corresponding column of Y.

Specify Line Style, Marker Symbol and Color

Create a stairstep plot and set the line style to a dot-dashed line, the marker symbol to circles, and the color to red.

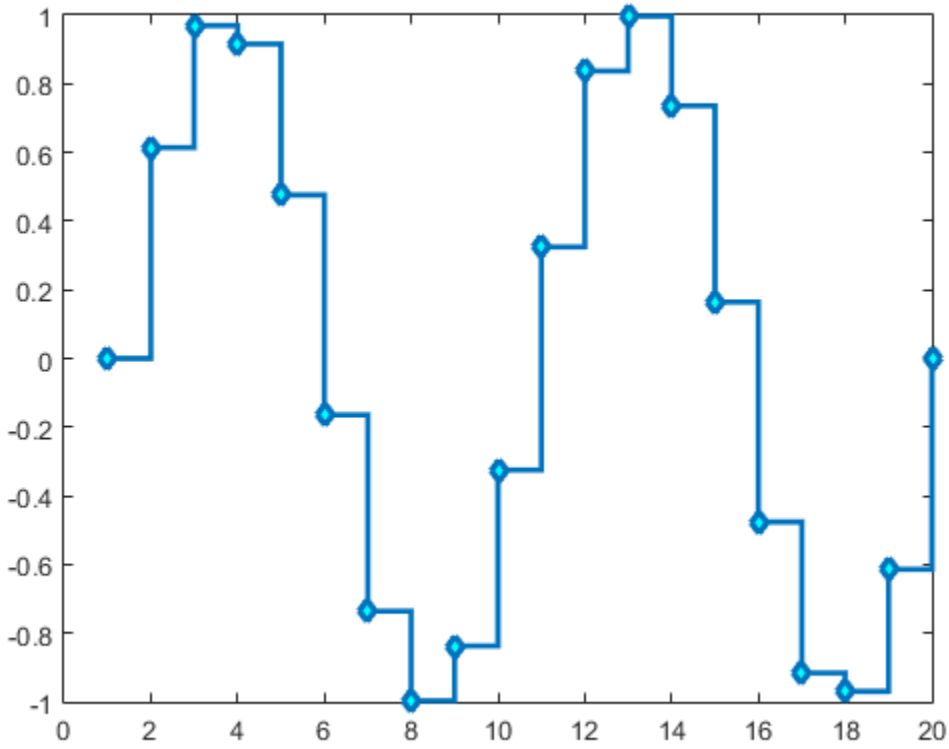
```
X = linspace(0,4*pi,20);
Y = sin(X);

figure
stairs(Y, '-.or')
```



Specify Additional Style Options

Create a stairstep plot and set the line width to 2, the marker symbols to diamonds, and the marker face color to cyan using `Name,Value` pair arguments.

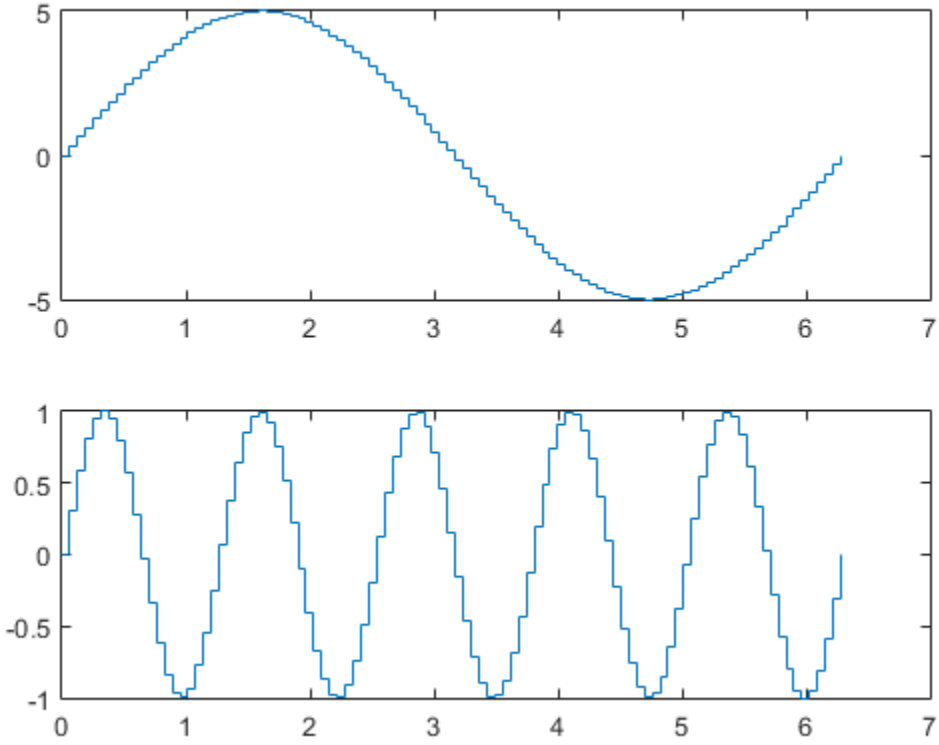
```
X = linspace(0,4*pi,20);
Y = sin(X);

figure
stairs(Y,'LineWidth',2,'Marker','d','MarkerFaceColor','c')
```



Specify Axes for Stairstep Plots

Create a figure with two subplots and return the two axes handles, `s(1)` and `s(2)`. Create a stairstep plot in each subplot by referring to the axes handles.

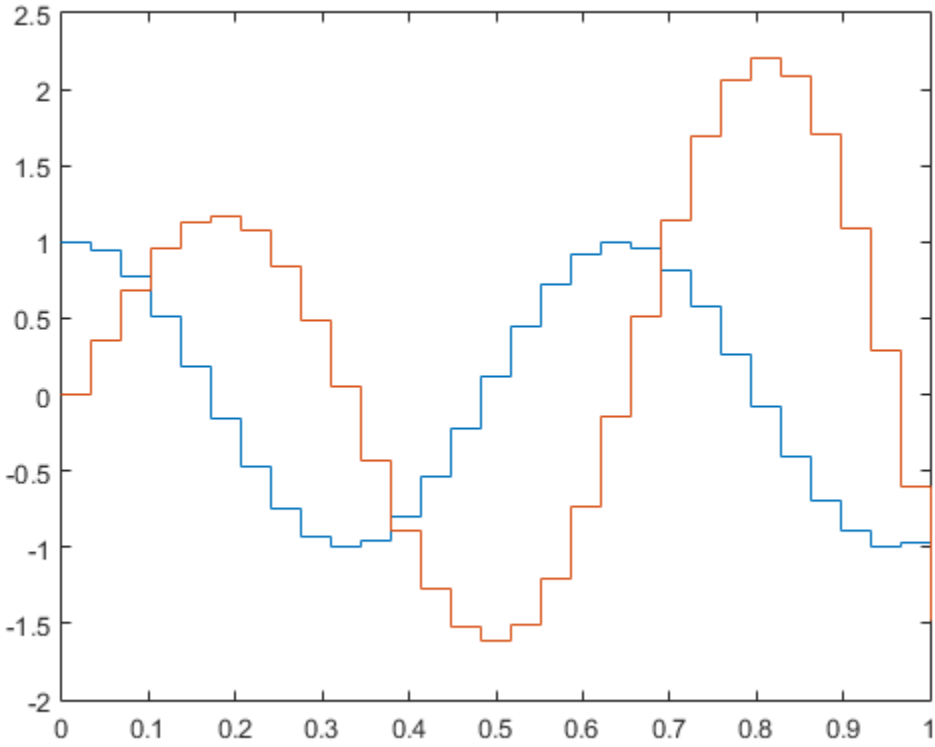
```
figure
s(1) = subplot(2,1,1);
s(2) = subplot(2,1,2);

X = linspace(0,2*pi);
Y1 = 5*sin(X);
Y2 = sin(5*X);
stairs(s(1),X,Y1)
stairs(s(2),X,Y2)
```



Modify Stairstep Plot After Creation

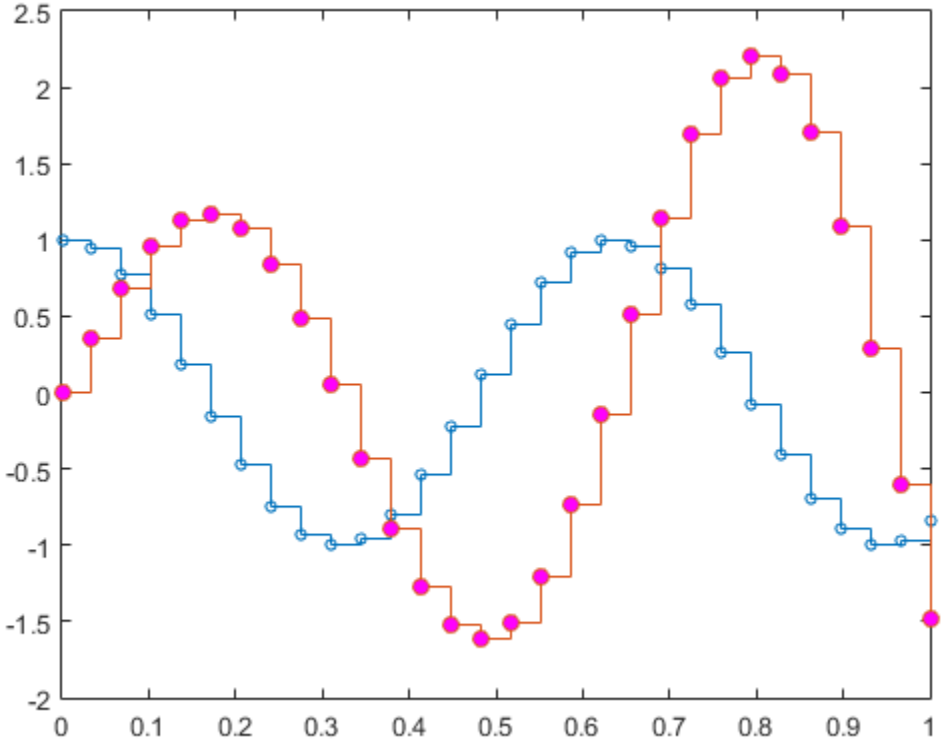
Create a stairstep plot of two data series and return the two stair objects.

```
X = linspace(0,1,30)';
Y = [cos(10*X), exp(X).*sin(10*X)];
h = stairs(X,Y);
```



Use small circle markers for the first data series. Use magenta filled circles for the second series. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
h(1).Marker = 'o';
h(1).MarkerSize = 4;
h(2).Marker = 'o';
h(2).MarkerFaceColor = 'm';
```



Create a Stairstep Plot using plot Function

Evaluate two cosine functions at 50 equally spaced values between 0 and 4π and create a stairstep plot using `plot`.

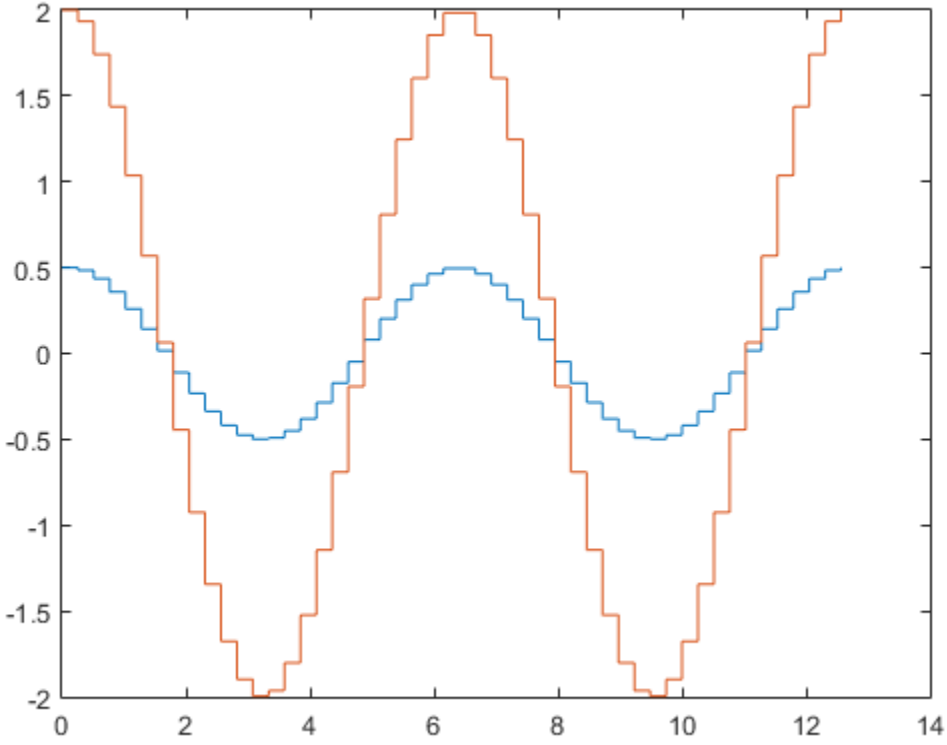
```
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];
[xb,yb] = stairs(X,Y);
```

`stairs` returns two matrices of the same size, `xb` and `yb`, but no plot.

Use `plot` to create the stairstep plot with `xb` and `yb`.

```
figure
```

```
plot(xb,yb)
```



Input Arguments

Y — y values

vector or matrix

y values, specified as a vector or matrix. When `Y` is a vector, `stairs` creates one stair object. When `Y` is a matrix, `stairs` draws one line per matrix column and creates a separate stair object for each column.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

X — x values
vector or matrix

x values, specified as a vector or matrix. When `Y` is a vector, `X` must be a vector of the same size. When `Y` is a matrix, `X` must be a matrix of the same size, or a vector whose length equals the number of rows in `Y`.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

LineSpec — Line style, marker symbol, and color
string

Line style, marker symbol, and color, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

ax — Axes object
axes object

Axes object. If you do not specify an axes, then `stairs` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Marker','s','MarkerFaceColor','red'` plots the stairstep graph with red square markers.

The properties listed here are only a subset. For a complete list, see Stair Properties.

'LineStyle' — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	

String	Line Style	Resulting Line
'--'	Dashed line	- - - - -
':'	Dotted line	.....
'-.'	Dash-dotted line	-.-.-.-
'none'	No line	No line

'LineWidth' — Line width
0.5 (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: 0.75

'Color' — Line color
[0 0.4470 0.7410] (default) | RGB triplet | color string | 'none'

Line color, specified as an RGB triplet, a color string, or 'none'. If you specify the Color as 'none', then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `'blue'`

Example: `[0 0 1]`

`'Marker'` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the stairs object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`'MarkerSize'` — Marker size

`6` (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

'MarkerEdgeColor' — Marker outline color

'auto' (default) | 'none' | RGB triplet | color string

Marker outline color, specified as one of these values:

- 'auto' — Use the same color as the Color property.
- 'none' — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.5 0.5 0.5]

Example: 'blue'

'MarkerFaceColor' — Marker fill color

'none' (default) | 'auto' | RGB triplet | color string

Marker fill color, specified as one of these values:

- 'none' — Use no color, which makes the interior invisible.
- 'auto' — Use the same color as the Color property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Output Arguments

h — Stair objects

scalar or column vector

Stair objects, returned as a scalar or a column vector. These are unique identifiers, which you can use to query and modify the properties of a specific stair object after it is created.

xb — x values for use with `plot`

vector or matrix

x values for use with `plot`, specified as a vector or matrix. `xb` contains the appropriate values such that `plot(xb,yb)` creates the stairstep graph.

yb — y values for use with `plot`

vector or matrix

y values for use with `plot`, specified as a vector or matrix. `yb` contains the appropriate values such that `plot(xb,yb)` creates the stairstep graph.

See Also

Functions
`bar` | `histogram` | `LineSpec` | `stem`

Properties
Stair Properties

Introduced before R2006a

Stair Properties

Control stair appearance and behavior

Stair properties control the appearance and behavior of a stair object. By changing property values, you can modify certain aspects of the stairs.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = stairs(1:10);
c = h.Color;
h.Color = 'red';
```

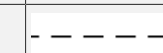
If you are using an earlier release, use the `get` and `set` functions instead.

Line

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Line color, specified as an RGB triplet, a color string, or `'none'`. The default RGB triplet value of `[0 0 0]` corresponds to black. If you specify the `Color` as `'none'`, then the line is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

Markers

`Marker` — Marker symbol

`'none'` (default) | marker string

Marker symbol, specified as one of the marker strings listed in this table. By default, the stair object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

`6` (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.3 0.2 0.1]

Example: 'green'

Data

XData — x values

[ ] (default) | vector

x values, specified as a vector. The input argument X to the stairs function sets the *x* values. If you do not specify the *x* values, then stairs uses the indices of YData. XData and YData must have equal lengths.

Example: 1:10

YData — y values

[ ] (default) | vector

y values, specified as a vector. The input argument Y to the stairs function sets the *y* values. XData and YData must have equal lengths.

Example: 1:10

XDataSource — Variable linked to XData

'' (default) | string containing MATLAB workspace variable name

Variable linked to XData, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the XData.

By default, there is no linked variable so the value is an empty string, ''. If you link a variable, then MATLAB does not update the XData values immediately. To force an update of the data values, use the refreshdata function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YDataSource` — Variable linked to `YData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `YData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`XDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData` (or `ZData` for 3-D plots).
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

Visibility

`Visible` — Visibility of stair

`'on'` (default) | `'off'`

Visibility of stair, specified as one of these values:

- `'on'` — Display the stair.

- `'off'` — Hide the stair without deleting it. You still can access the properties of an invisible stair object.

`Clipping` — Clipping of stair to axes limits

`'on'` (default) | `'off'`

Clipping of stair to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the stair that are outside the axes limits.
- `'off'` — Display the entire stair, even if parts of it appear outside the axes limits. Parts of the stair might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the stair that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'stair'`

Type of graphics object, returned as `'stair'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

`Tag` — Tag to associate with stair

`''` (default) | string

Tag to associate with the stair, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with stair

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the stair object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the stair.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the stair object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the stair from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.
3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the stair object in the legend as one entry (default).
   - `'off'` — Do not include the stair object in the legend.
   - `'children'` — Include only children of the stair object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of stair
axes object | group object | transform object

Parent of stair, specified as an axes, group, or transform object.

`Children` — Children of stair
empty `GraphicsPlaceholder` array

The stair has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of stair object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The stair object handle is always visible.
- `'off'` — The stair object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The stair object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the stair at the command-line, but allows callback functions to access it.

If the stair object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the stair. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The stair object — You can access properties of the stair object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the stair. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the stair when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the stair.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the stair that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the stair responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the stair passes the click to the object below it in the current view of the figure window. The HitTest property of the stair has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the stair. If you have defined the UIContextMenu property, then invoke the context menu.

- `'off'` — Trigger the callbacks for the nearest ancestor of the stair that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the stair object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for stair

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for stair, specified as one of these values:

- `'off'` — Click the stair plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the stair plot to select it, that is, anywhere within the rectangle that encloses the stair plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the ButtonDownFcn callback of the stair is the running callback, then the Interruptible property determines if it another callback can interrupt it:

- 'on' — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a drawnow, figure, getframe, waitfor, or pause command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- 'off' — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

BusyAction — Callback queuing
'queue' (default) | 'cancel'

Callback queuing specified as 'queue' or 'cancel'. The BusyAction property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the ButtonDownFcn callback of the stair tries to interrupt a running callback that cannot be interrupted, then the BusyAction property determines if it is discarded or put in the queue. Specify the BusyAction property as one of these values:

- 'queue' — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.

- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the stair. Setting the `CreateFcn` property on an existing stair has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during stair creation. MATLAB executes the callback after creating the stair and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The stair object — You can access properties of the stair object from within the callback function. You also can access the stair object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle

- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the stair. MATLAB executes the callback before destroying the stair so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The stair object — You can access properties of the stair object from within the callback function. You also can access the stair object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of stair

`'off'` (default) | `'on'`

Deletion status of stair, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the stair begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the stair no longer exists.

Check the value of the BeingDeleted property to verify that the stair is not about to be deleted before querying or modifying it.

See Also

`stairs`

More About

- "Access Property Values"
- "Graphics Object Properties"

standardizeMissing

Insert missing value indicators into table

Syntax

```
B = standardizeMissing(A,id)
B = standardizeMissing(A,id,'DataVariables',vars)
```

Description

`B = standardizeMissing(A,id)` replaces all instances of the values specified in id occurring within table `A` with the standard missing value indicators. `B` is a table.

The standard missing value indicators depend on the data type:

- `NaN` for `double` and `single` floating-point arrays
- `<undefined>` for categorical arrays
- empty string, `{''}`, for cell arrays of strings
- blank string, `[' ']`, for character arrays

`standardizeMissing` checks `double` and `single` variables in `A` against numeric values from `id` and checks string and categorical variables in `A` against strings from `id`. The function `standardizeMissing` ignores integer data types because they cannot contain `NaN`.

`B = standardizeMissing(A,id,'DataVariables',vars)` replaces values only in variables specified by `vars`.

Examples

Replace All Instances of Specified Values

Create a table containing `Inf` and `'N/A'` to represent missing values.

```
dblVar = [NaN;3;Inf;7;9];
```

```
cellstrVar = {'one';'three';'';'NA';'nine'};
charVar = ['A';'C';'E';' ';'I'];
categoryVar = categorical({'red';'yellow';'blue';'violet';''});

A = table(dblVar,cellstrVar,charVar,categoryVar)

A =

    dblVar    cellstrVar    charVar    categoryVar
    ______    __________    _______    ___________

    NaN       'one'         A          red
      3       'three'       C          yellow
    Inf       ''            E          blue
      7       'NA'                     violet
      9       'nine'        I          <undefined>
```

Replace all instances of `Inf` with `NaN` and replace all instances of `'NA'` with the empty string, `''`.

```
B = standardizeMissing(A,{Inf,'NA'})

B =

    dblVar    cellstrVar    charVar    categoryVar
    ______    __________    _______    ___________

    NaN       'one'         A          red
      3       'three'       C          yellow
    NaN       ''            E          blue
      7       ''                       violet
      9       'nine'        I          <undefined>
```

Replace Only Values in Specified Variables

Replace instances of `Inf`, and `'N/A'`, occurring in specified variables of a table, with the standard missing value indicators.

Create a table containing `Inf` and `'N/A'` to represent missing values.

```
a = {'alpha';'bravo';'charlie';'';'N/A'};
x = [1;NaN;3;Inf;5];
y = [57;732;93;1398;Inf];

A = table(a,x,y)
```

```
A =

       a          x       y
    _________    ___    ____

    'alpha'        1      57
    'bravo'      NaN     732
    'charlie'      3      93
    ''           Inf    1398
    'N/A'          5     Inf
```

For the variables `a` and `x`, replace instances of `Inf` with `NaN` and `'N/A'` with the empty string, `''`.

```
B = standardizeMissing(A,{Inf,'N/A'},'DataVariables',{'a','x'})
```

```
B =

       a          x       y
    _________    ___    ____

    'alpha'        1      57
    'bravo'      NaN     732
    'charlie'      3      93
    ''           NaN    1398
    ''             5     Inf
```

`Inf` in the variable `y` remains unchanged because `y` is not included in the `'DataVariables'` name-value pair argument.

Input Arguments

`A` — Input table

table

Input table, specified as a table.

`id` — Nonstandard missing value indicators

numeric vector | string | cell array containing numeric values and strings

Nonstandard missing value indicators, specified as a numeric vector, string, or cell array containing numeric values and strings.

`vars` — Subset of variables to consider

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Subset of variables to consider, specified as a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

Output Arguments

`B` — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

More About

Algorithms

`standardizeMissing` treats leading and trailing white space differently for cell arrays of strings, character arrays, and categorical arrays.

- For cell arrays of strings, `standardizeMissing` does not ignore white space. All strings must match exactly a string specified in `id`.
- For character arrays, `standardizeMissing` ignores trailing white space.
- For categorical arrays, `standardizeMissing` ignores leading and trailing white space.

See Also

`ismissing` | `table`

startup

Startup file for user-defined options

Syntax

```
startup
```

Description

`startup` executes commands of your choosing when the MATLAB program starts.

Create a `startup.m` file in a folder on the MATLAB search path. Add commands you want executed at startup. For example, your `startup.m` file might include physical constants, defaults for Handle Graphics properties, engineering conversion factors, or anything else you want predefined in your workspace.

More About

Tips

- To specify the current folder in MATLAB when it starts, set the **Initial working folder** preference, described in "General Preferences".

Algorithms

At startup, MATLAB automatically executes the file `matlabrc.m` and, if it exists on the MATLAB search path, `startup.m`. The file `matlabrc.m`, which is in the *`matlabroot`*`/toolbox/local` folder, is reserved for use by MathWorks and by system administrators on multiuser systems.

- "Startup Options in MATLAB Startup File"
- Preferences

See Also

`finish` | `matlabrc` | `matlabroot` | `path` | `quit` | `userpath`

std

Standard deviation

Syntax

```
S = std(A)
S = std(A,w)
S = std(A,w,dim)
S = std(___,nanflag)
```

Description

`S = std(A)` returns the standard deviation of the elements of `A` along the first array dimension whose size does not equal 1.

- If `A` is a vector of observations, then the standard deviation is a scalar.
- If `A` is a matrix whose columns are random variables and whose rows are observations, then `S` is a row vector containing the standard deviations corresponding to each column.
- If `A` is a multidimensional array, then `std(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.
- By default, the standard deviation is normalized by `N-1`, where `N` is the number of observations.

`S = std(A,w)` specifies a weighting scheme for any of the previous syntaxes. When `w = 0` (default), `S` is normalized by `N-1`. When `w = 1`, `S` is normalized by the number of observations, `N`. `w` also can be a weight vector containing nonnegative elements. In this case, the length of `w` must equal the length of the dimension over which `std` is operating.

`S = std(A,w,dim)` returns the standard deviation along dimension `dim` for any of the previous syntaxes. To maintain the default normalization while specifying the dimension of operation, set `w = 0` in the second argument.

`S = std(___,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. For example, `std(A,'includenan')` includes all `NaN` values in `A` while `std(A,'omitnan')` ignores them.

Examples

Standard Deviation of Matrix Columns

Create a matrix and compute the standard deviation of each column.

```
A = [4 -5 1; 2 3 5; -9 1 7];
S = std(A)
```

```
S =

    7.0000    4.1633    3.0551
```

Standard Deviation of 3-D Array

Create a 3-D array and compute the standard deviation along the first dimension.

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
S = std(A)
```

```
S(:,:,1) =

    2.8284    2.1213


S(:,:,2) =

    9.8995    4.2426


S(:,:,3) =

    2.8284    4.9497
```

Specify Standard Deviation Weights

Create a matrix and compute the standard deviation of each column according to a weight vector `w`.

```
A = [1 5; 3 7; -9 2];
w = [1 1 0.5];
S = std(A,w)
```

```
S =

    4.4900    1.8330
```

Standard Deviation Along Matrix Rows

Create a matrix and calculate the standard deviation along each row.

```
A = [6 4 23 -3; 9 -10 4 11; 2 8 -5 1];
S = std(A,0,2)
```

```
S =

   11.0303
    9.4692
    5.3229
```

Standard Deviation Excluding `NaN`

Create a vector and compute its standard deviation, excluding `NaN` values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
S = std(A,'omitnan')
```

```
S =

    2.2797
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. If `A` is a scalar, then `std(A)` returns `0`. If `A` is a `0`-by-`0` empty array, then `std(A)` returns `NaN`.

Data Types: `single` | `double`
Complex Number Support: Yes

`w` — Weight

`0` (default) | `1` | vector

Weight, specified as one of these values:

- `0` — Normalize by `N-1`, where `N` is the number of observations. If there is only one observation, then the weight is 1.
- `1` — Normalize by `N`.
- Vector made up of nonnegative scalar weights corresponding to the dimension of `A` along which the standard deviation is calculated.

Data Types: `single` | `double`

`dim` — Dimension to operate along

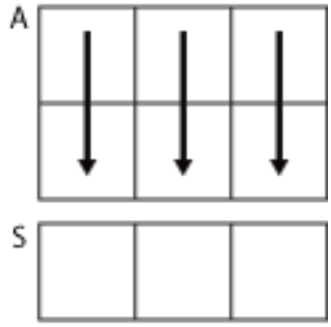
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(S,dim)` is `1`, while the sizes of all other dimensions remain the same.

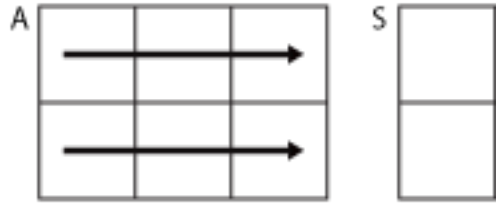
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `std(A,0,1)` returns a row vector containing the standard deviation of the elements in each column.



std(A,0,1)

- If `dim = 2`, then `std(A,0,2)` returns a column vector containing the standard deviation of the elements in each row.



std(A,0,2)

If `dim` is greater than `ndims(A)`, then `std(A)` returns an array of zeros the same size as `A`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition

`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values when computing the standard deviation, resulting in `NaN`.
- `'omitnan'` — Ignore `NaN` values appearing in either the input array or weight vector.

Data Types: `char`

More About

Standard Deviation

For a random variable vector A made up of N scalar observations, the standard deviation is defined as

$$S = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}|A_i - \mu|^2},$$

where μ is the mean of A:

$$\mu = \frac{1}{N}\sum_{i=1}^{N}A_i.$$

The standard deviation is the square root of the variance. Some definitions of standard deviation use a normalization factor of N instead of $N-1$, which you can specify by setting `w` to `1`.

See Also

`corrcoef` | `cov` | `mean` | `median` | `var`

stem

Plot discrete sequence data

Syntax

```
stem(Y)
stem(X,Y)
stem(___,'filled')
stem(___,LineSpec)
stem(___,Name,Value)

stem(ax,___)

h = stem(___)
```

Description

`stem(Y)` plots the data sequence, `Y`, as stems that extend from a baseline along the *x*-axis. The data values are indicated by circles terminating each stem.

- If `Y` is a vector, then the *x*-axis scale ranges from 1 to `length(Y)`.
- If `Y` is a matrix, then `stem` plots all elements in a row against the same *x* value, and the *x*-axis scale ranges from 1 to the number of rows in `Y`.

`stem(X,Y)` plots the data sequence, `Y`, at values specified by `X`. The `X` and `Y` inputs must be vectors or matrices of the same size. Additionally, `X` can be a row or column vector and `Y` must be a matrix with `length(X)` rows.

- If `X` and `Y` are both vectors, then `stem` plots entries in `Y` against corresponding entries in `X`.
- If `X` is a vector and `Y` is a matrix, then `stem` plots each column of `Y` against the set of values specified by `X`, such that all elements in a row of `Y` are plotted against the same value.
- If `X` and `Y` are both matrices, then `stem` plots columns of `Y` against corresponding columns of `X`.

`stem( ___ ,'filled')` fills the circles. Use this option with any of the input argument combinations in the previous syntaxes.

`stem( ___ ,LineSpec)` specifies the line style, marker symbol, and color.

`stem( ___ ,Name,Value)` specifies stem series properties using one or more `Name,Value` pair arguments.

`stem(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in the previous syntaxes.

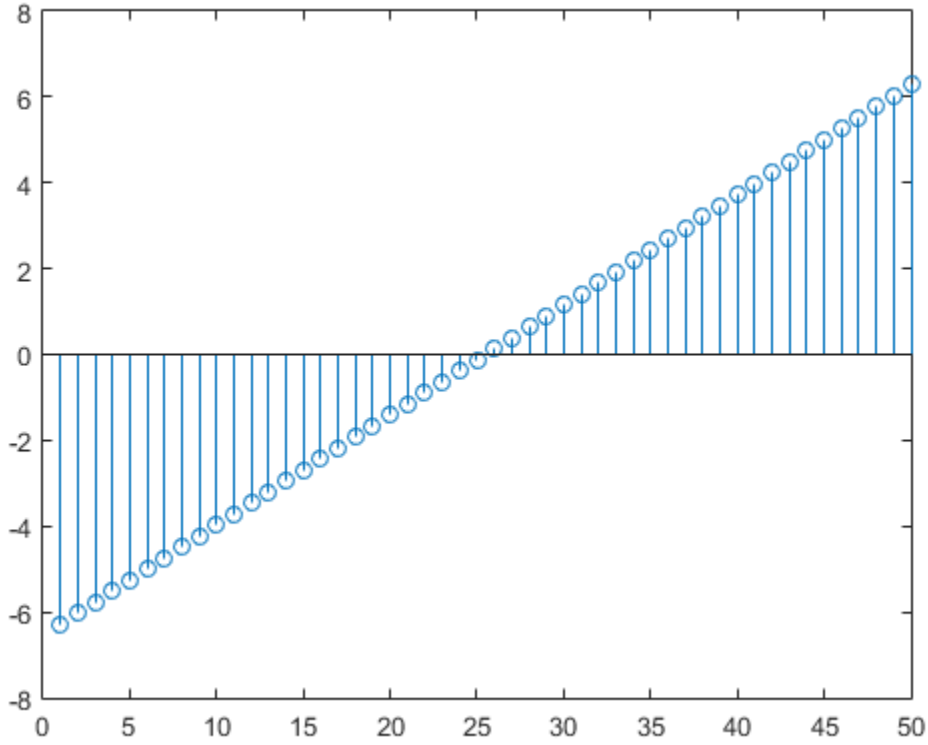
`h = stem( ___ )` returns a vector of stem series handles in `h`. When multiple stem series are present, you can make changes to properties of a specific stem series by specifying a particular handle.

Examples

Plot Single Data Series

Create a stem plot of 50 data values between -2π and 2π.

```
figure
Y = linspace(-2*pi,2*pi,50);
stem(Y)
```

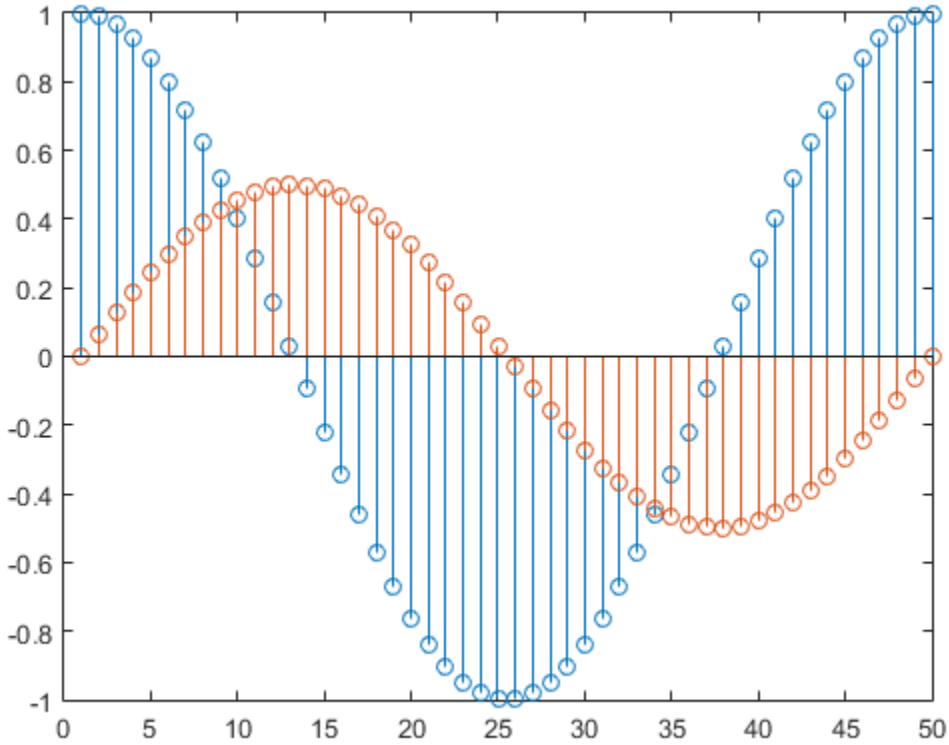


Data values are plotted as stems extending from the baseline and terminating at the data value. The length of `Y` automatically determines the position of each stem on the *x*-axis.

Plot Multiple Data Series

Plot two data series using a two-column matrix.

```
figure
X = linspace(0,2*pi,50)';
Y = [cos(X), 0.5*sin(X)];
stem(Y)
```

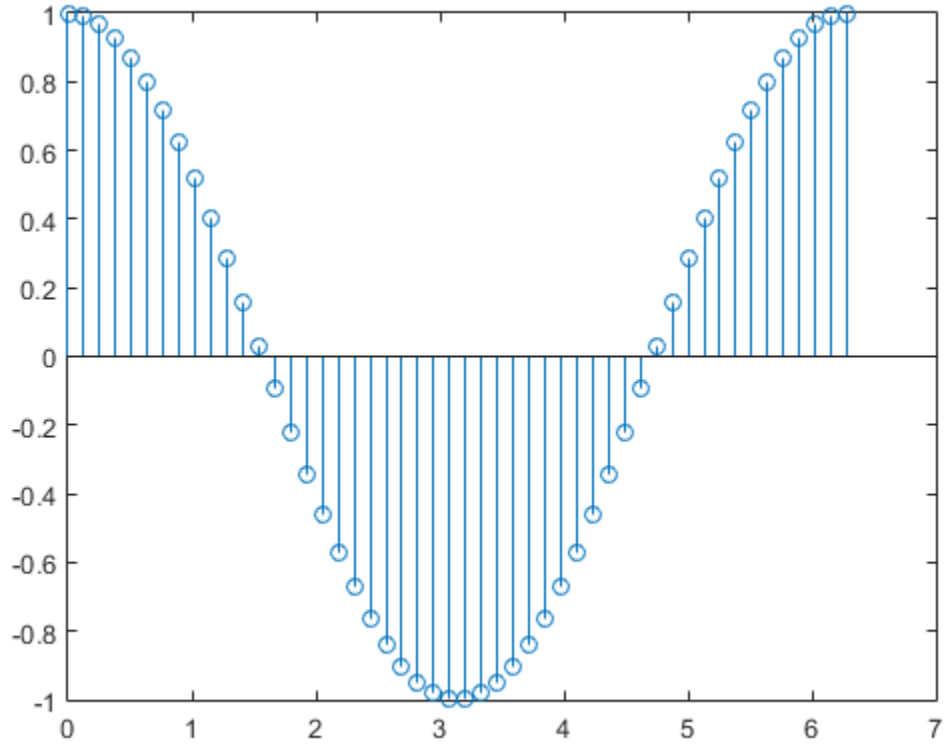


Each column of Y is plotted as a separate series, and entries in the same row of Y are plotted against the same *x* value. The number of rows in Y automatically generates the position of each stem on the *x*-axis.

Plot Single Data Series at Specified x values

Plot 50 data values of cosine evaluated between 0 and 2π and specify the set of *x* values for the stem plot.

```
figure
X = linspace(0,2*pi,50)';
Y = cos(X);
stem(X,Y)
```

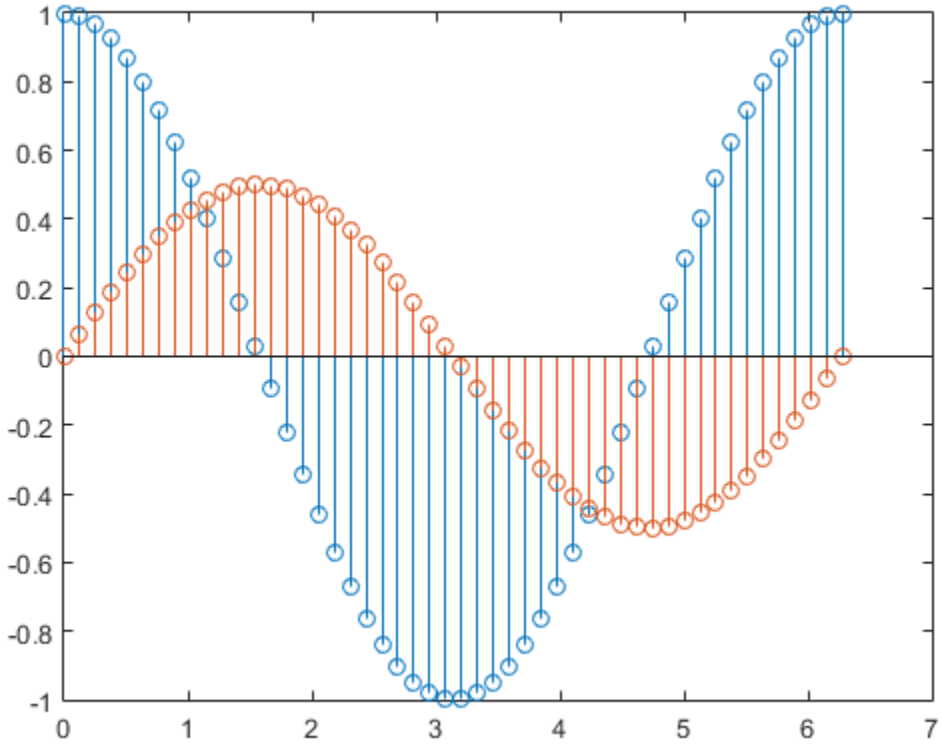


The first vector input determines the position of each stem on the x-axis.

Plot Multiple Data Series at Specified x values

Plot 50 data values of sine and cosine evaluated between 0 and 2π and specify the set of x values for the stem plot.

```
figure
X = linspace(0,2*pi,50)';
Y = [cos(X), 0.5*sin(X)];
stem(X,Y)
```

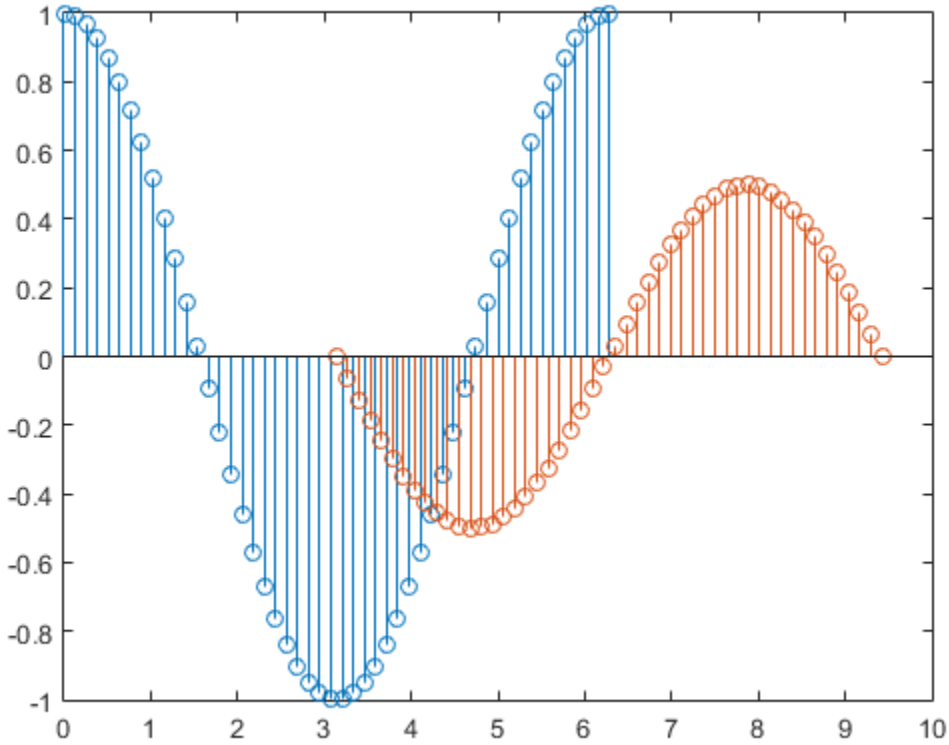


The vector input determines the *x*-axis positions for both data series.

Plot Multiple Data Series at Unique Sets of x values

Plot 50 data values of sine and cosine evaluated at different sets of *x* values. Specify the corresponding sets of *x* values for each series.

```
figure
x1 = linspace(0,2*pi,50)';
x2 = linspace(pi,3*pi,50)';
X = [x1, x2];
Y = [cos(x1), 0.5*sin(x2)];
stem(X,Y)
```

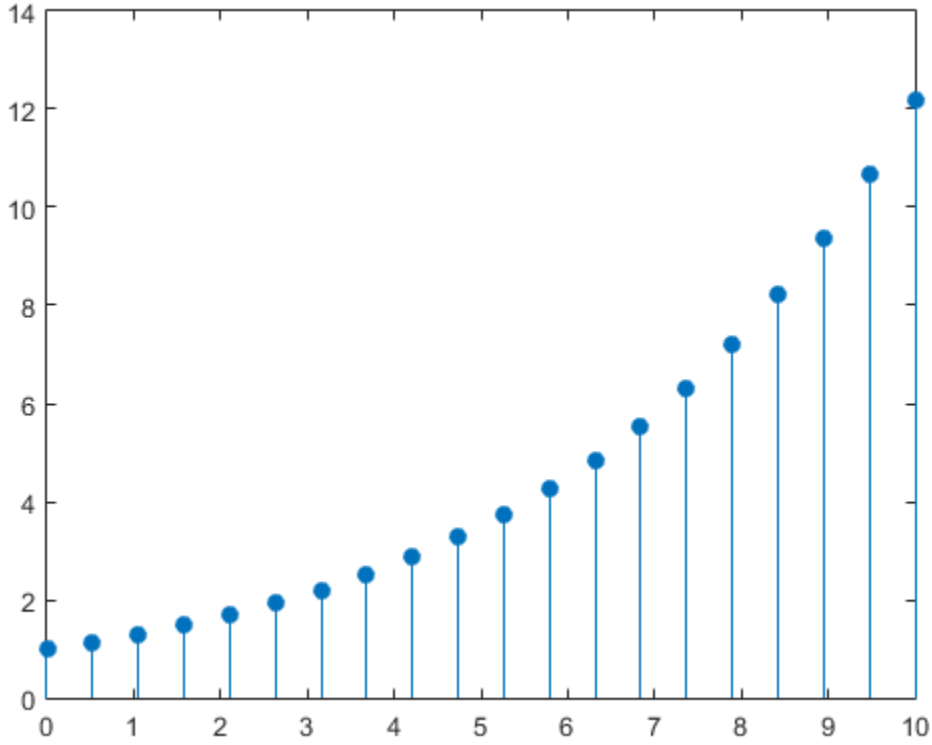


Each column of `X` is plotted against the corresponding column of `Y`.

Fill in Plot Markers

Create a stem plot and fill in the circles that terminate each stem.

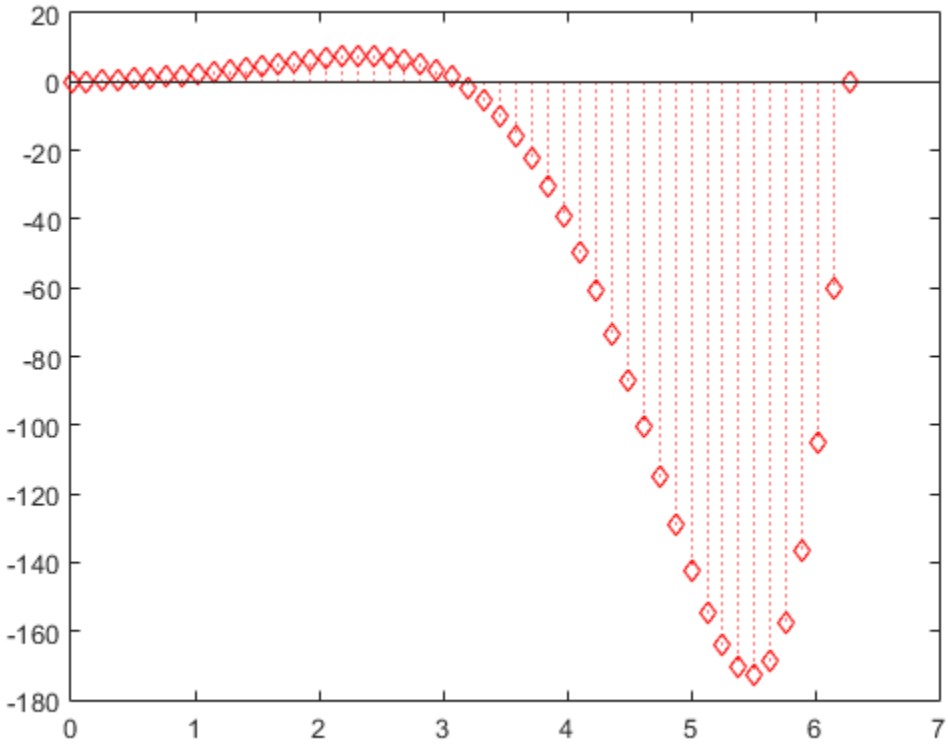
```
X = linspace(0,10,20)';
Y = (exp(0.25*X));
stem(X,Y,'filled')
```



Specify Stem and Marker Options

Create a stem plot and set the line style to a dotted line, the marker symbols to diamonds, and the color to red using the `LineSpec` option.

```
figure
X = linspace(0,2*pi,50)';
Y = (exp(X).*sin(X));
stem(X,Y,':diamondr')
```

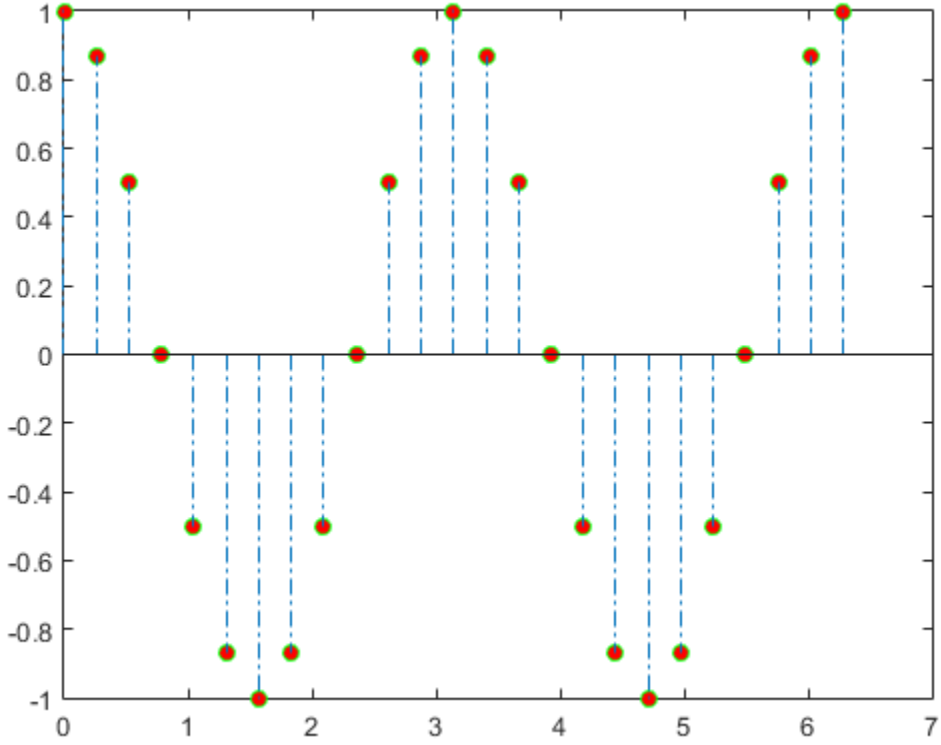


To color the inside of the diamonds, use the `'fill'` option.

Specify Additional Stem and Marker Options

Create a stem plot and set the line style to a dot-dashed line, the marker face color to red, and the marker edge color to green using `Name,Value` pair arguments.

```
figure
X = linspace(0,2*pi,25)';
Y = (cos(2*X));
stem(X,Y,'LineStyle','-.',...
     'MarkerFaceColor','red',...
     'MarkerEdgeColor','green')
```

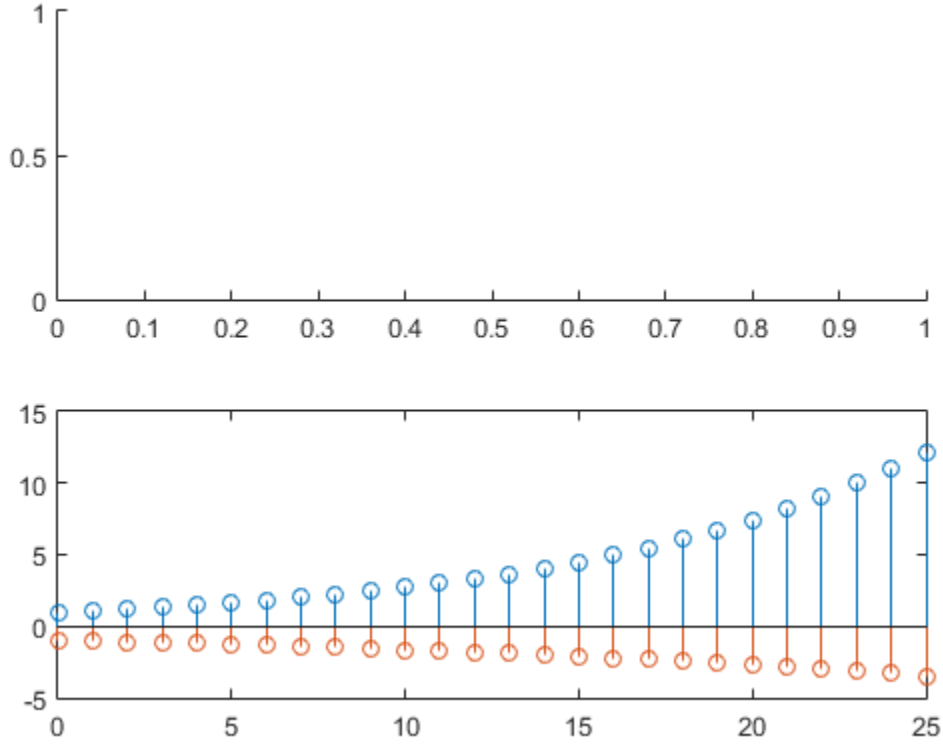


The stem remains the default color.

Specify Axes for Stem Plot

Create a figure with two subplots and return the handles to each axes, `s(1)` and `s(2)`. Create a stem plot in the lower subplot by referring to its axes handle, `s(2)`.

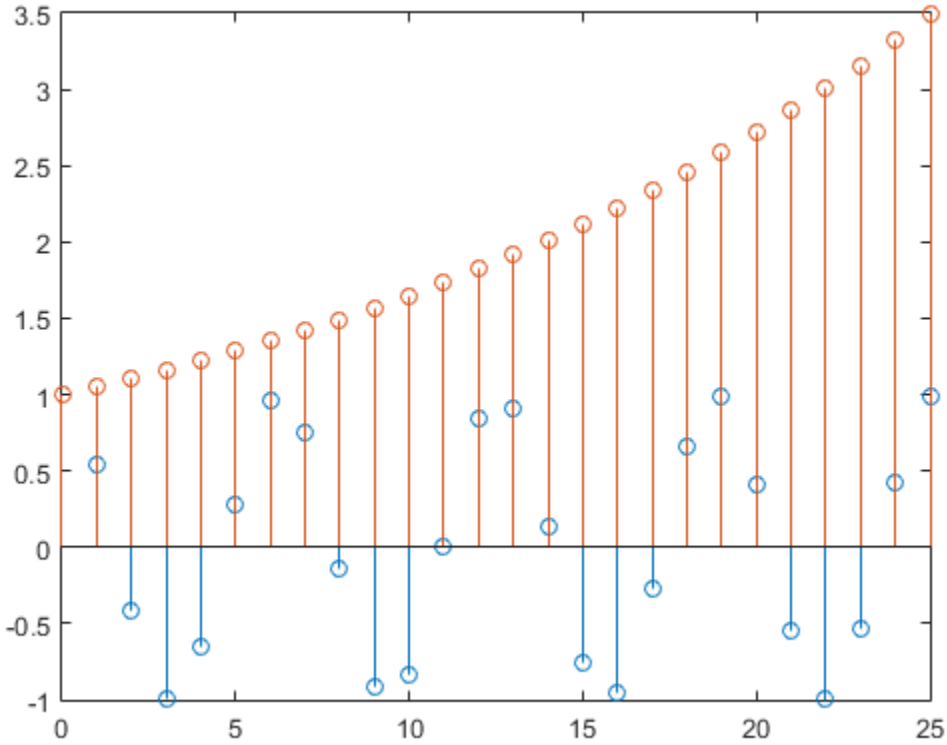
```
figure
s(1) = subplot(2,1,1);
s(2) = subplot(2,1,2);

X = 0:25;
Y = [exp(0.1*X); -exp(.05*X)]';
stem(s(2),X,Y)
```



Modify Stem Series After Creation

Create a stem plot.

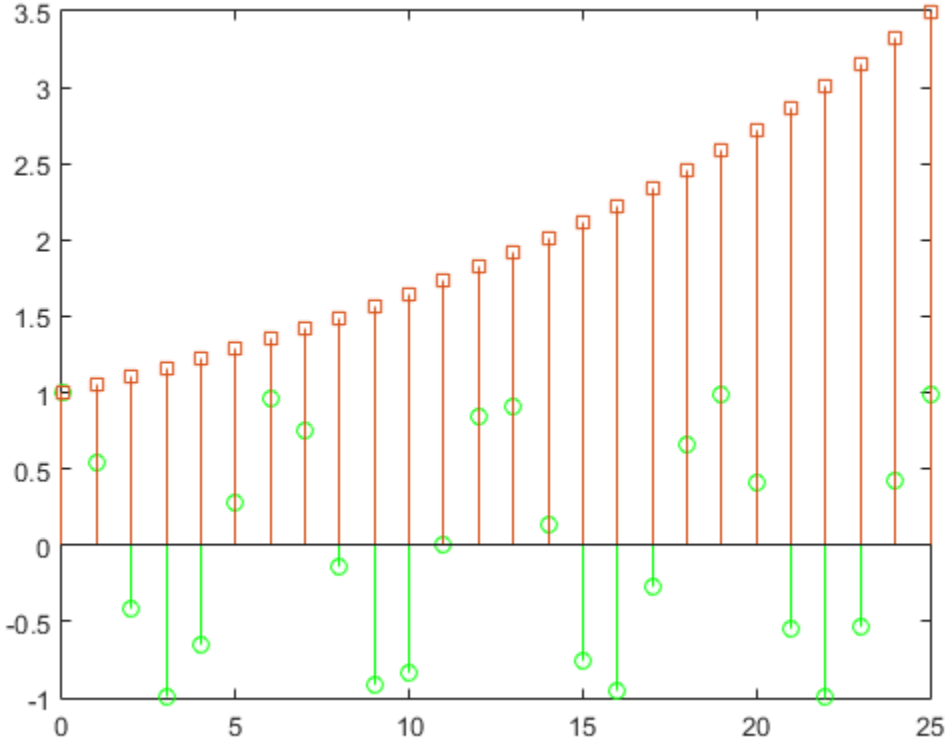
```
X = 0:25;
Y = [cos(X); exp(0.05*X)]';
h = stem(X,Y);
```



The `stem` function creates a stem series object for each column of data. The output argument, `h`, contains the two stem series objects.

Set the first stem series color to green. Change the markers of the second stem series to squares. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

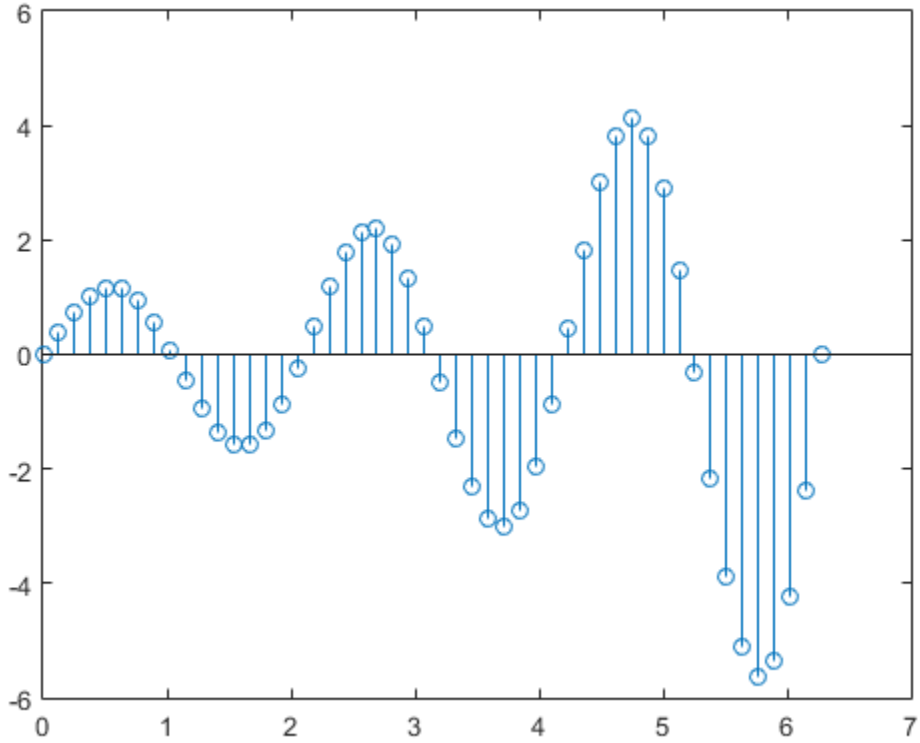
```
h(1).Color = 'green';
h(2).Marker = 'square';
```



Adjust Baseline Properties

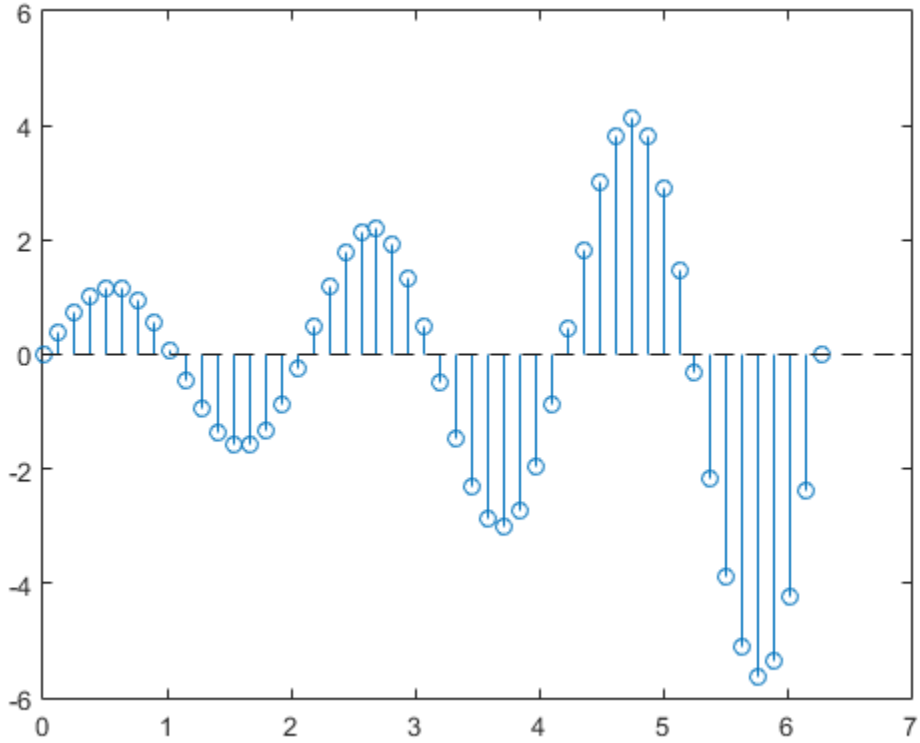
Create a stem plot and change properties of the baseline.

```
X = linspace(0,2*pi,50);
Y = exp(0.3*X).*sin(3*X);
h = stem(X,Y);
```



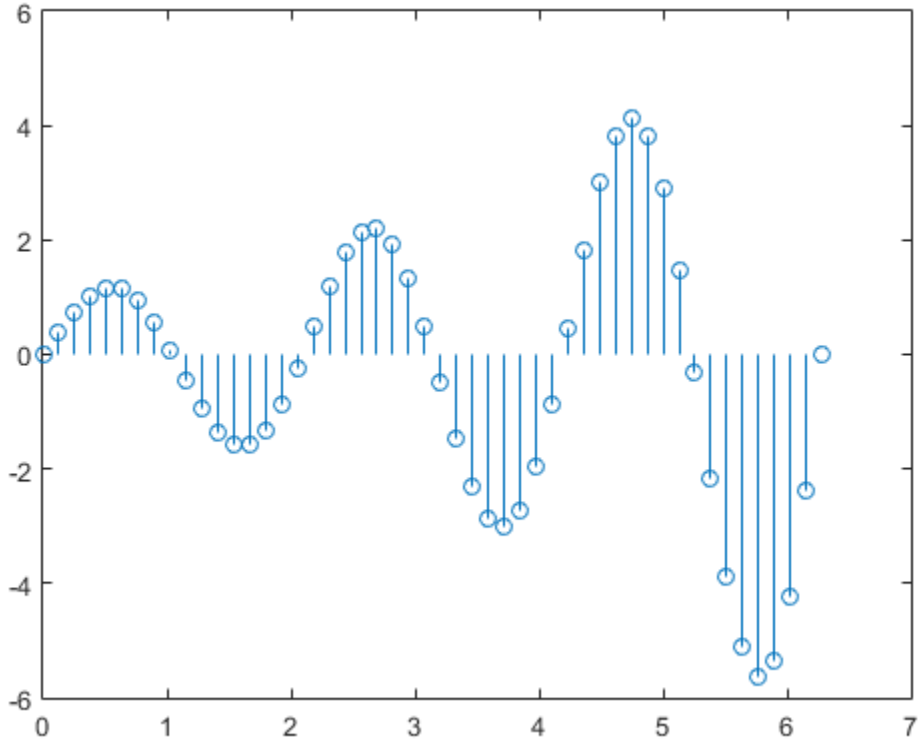
Change the line style of the baseline. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
hbase = h.BaseLine;
hbase.LineStyle = '--';
```



Hide the baseline by setting its `Visible` property to `'off'` .

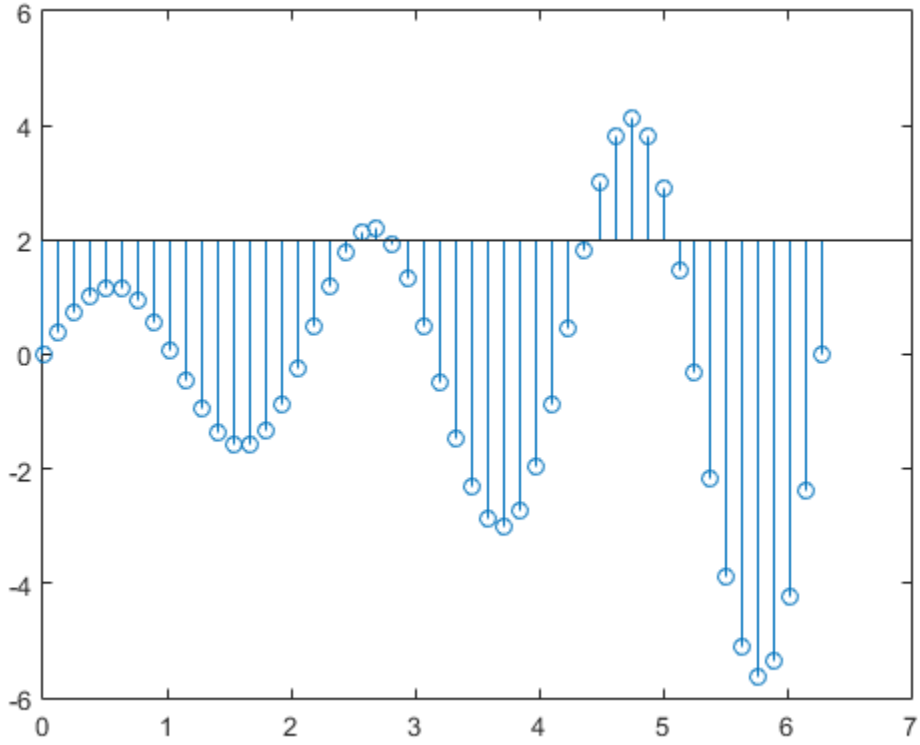
```
hbase.Visible = 'off';
```



Change Baseline Level

Create a stem plot with a baseline level at 2.

```
X = linspace(0,2*pi,50)';
Y = (exp(0.3*X).*sin(3*X));
stem(X,Y,'BaseValue',2);
```



- “Combine Stem Plot and Line Plot”

Input Arguments

Y — Data sequence to display

vector or matrix

Data sequence to display, specified as a vector or matrix. When `Y` is a vector, `stem` creates one stem series. When `Y` is a matrix, `stem` creates a separate stem series for each column.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

X — Locations to plot data values in Y

vector or matrix

Locations to plot data values in Y, specified as a vector or matrix. When Y is a vector, X must be a vector of the same size. When Y is a matrix, X must be a matrix of the same size, or a vector whose length equals the number of rows in Y.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

LineSpec — Line style, marker symbol, and color

string

Line style, marker symbol, and color, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

ax — Axes object

object

Axes object. Use `ax` to modify the stem series after it is created.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

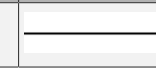
Example: `'LineStyle',':','MarkerFaceColor','red'` plots the stem as a dotted line and colors the marker face red.

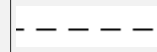
The stem series properties listed here are only a subset. For a complete list, see Stem Series Properties.

'LineStyle' — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	

String	Line Style	Resulting Line
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

'LineWidth' — Line width
0.5 (default) | positive value

Line width, specified as a positive value in point units. If the line has markers, then the line width also affects the marker edges.

Example: 0.75

'Color' — Stem color
[0 0 0] (default) | RGB triplet | color string | 'none'

Stem color, specified as an RGB triplet, a color string, or 'none'. If you specify the Color as 'none', then the stems are invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `'blue'`

Example: `[0 0 1]`

`'Marker'` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the marker strings listed in this table.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `‘diamond’`

`'MarkerSize'` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: `10`

`'MarkerEdgeColor'` — Marker outline color

`'auto'` (default) | `'none'` | RGB triplet | color string

Marker outline color, specified as one of these values:

- `'auto'` — Use the same color as the Color property.
- `'none'` — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`'MarkerFaceColor'` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Output Arguments

h — Stem series object

scalar or column vector

Stem series objects, returned as a scalar or column vector. This is a unique identifier, which you can use to modify the properties of a specific stem series after it is created.

See Also

Functions

`bar` | `LineSpec` | `plot` | `stairs`

Properties

Stem Series Properties

Introduced before R2006a

stem3

Plot 3-D discrete sequence data

Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3(___,'filled')
stem3(___,LineSpec)
stem3(___,Name,Value)

stem3(ax,___)

h = stem3(___)
```

Description

`stem3(Z)` plots entries in `Z` as stems extending from the *xy*-plane and terminating with circles at the entry values. The stem locations in the *xy*-plane are automatically generated.

`stem3(X,Y,Z)` plots entries in `Z` as stems extending from the *xy*-plane where `X` and `Y` specify the stem locations in the *xy*-plane. The inputs `X`, `Y`, and `Z` must be vectors or matrices of the same size.

`stem3(___,'filled')` fills the circles. Use this option with any of the input argument combinations in the previous syntaxes.

`stem3(___,LineSpec)` specifies the line style, marker symbol, and color.

`stem3(___,Name,Value)` specifies stem series properties using one or more `Name,Value` pair arguments.

`stem3(ax,___)` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in the previous syntaxes.

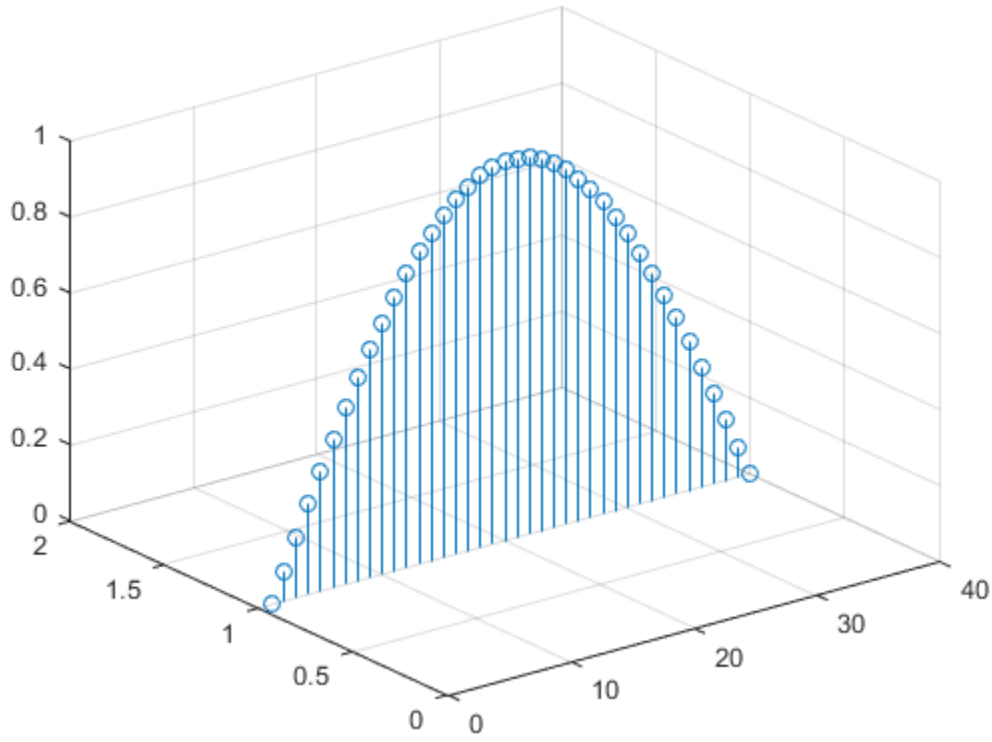
`h = stem3( ___ )` returns the stem series object `h`.

Examples

Row Vector Input

Create a 3-D stem plot of cosine values between $-\pi/2$ and $\pi/2$ with a row vector input.

```
figure
X = linspace(-pi/2,pi/2,40);
Z = cos(X);
stem3(Z)
```

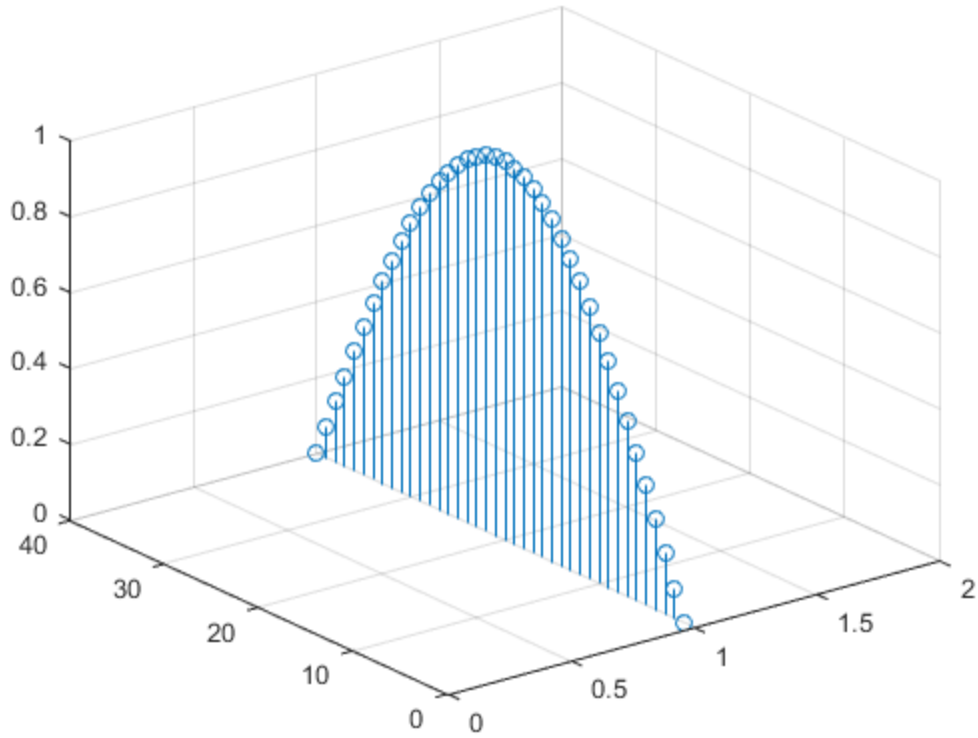


`stem3` plots elements of `Z` against the same *y* value at equally space *x* values.

Column Vector Input

Create a 3-D stem plot of cosine values between $-\pi/2$ and $\pi/2$ with a column vector input.

```
figure
X = linspace(-pi/2,pi/2,40)';
Z = cos(X);
stem3(Z)
```

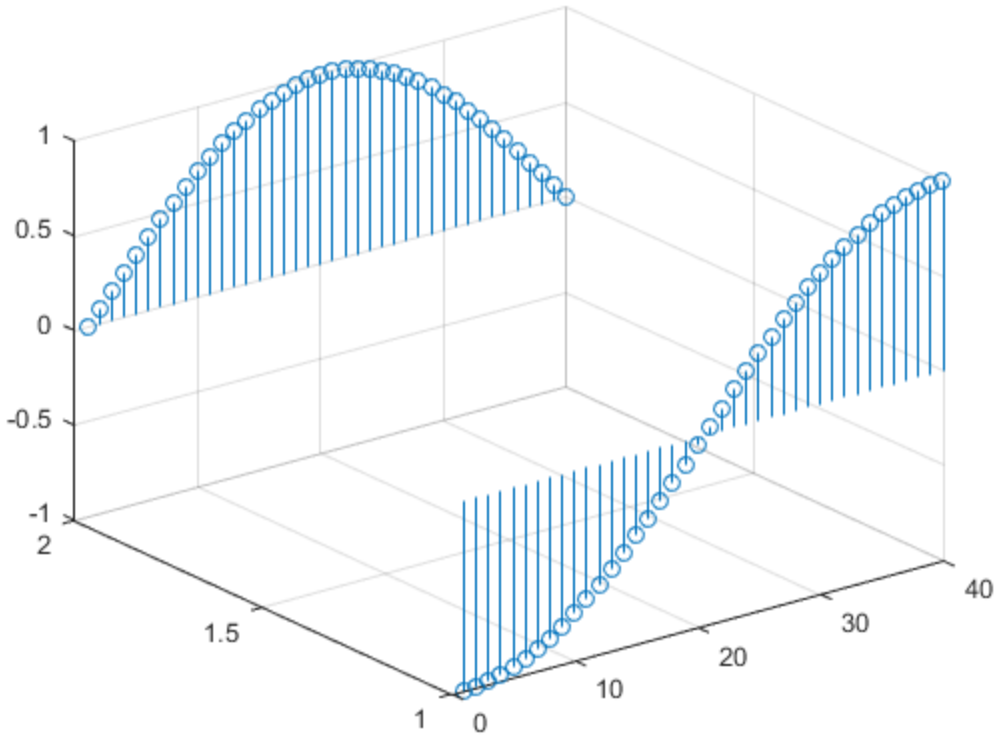


`stem3` plots elements of `Z` against the same *x* value at equally space *y* values.

Matrix Input

Create a 3-D stem plot of sine and cosine values between $-\pi/2$ and $\pi/2$ with a matrix input.

```
figure
X = linspace(-pi/2,pi/2,40);
Z = [sin(X); cos(X)];
stem3(Z)
```

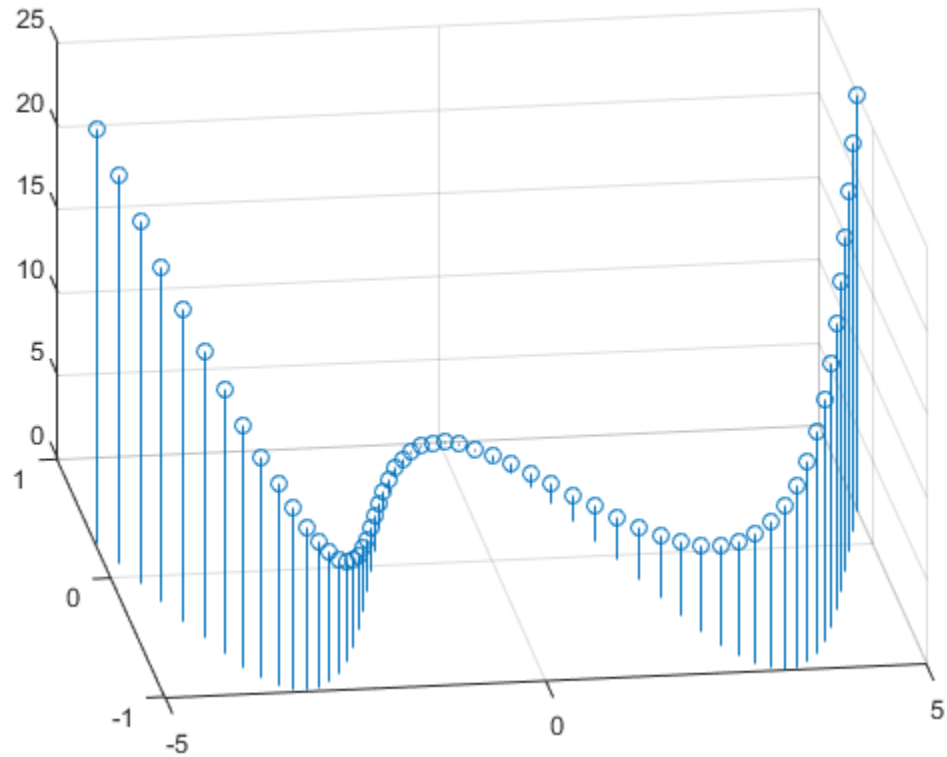


`stem3` plots each row of `Z` against the same *y* value at equally space *x* values.

Specify Stem Locations with Vector Inputs

Create a 3-D stem plot and specify the stem locations along a curve. Use `view` to adjust the angle of the axes in the figure.

```
figure
X = linspace(-5,5,60);
Y = cos(X);
Z = X.^2;
stem3(X,Y,Z)
view(-8,30)
```

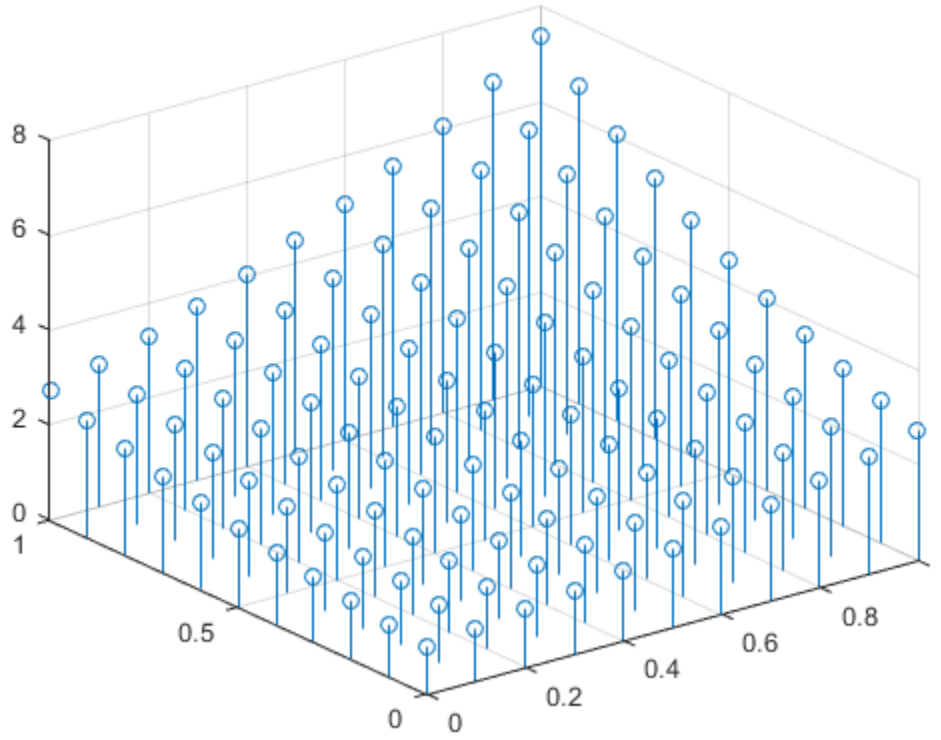


`X` and `Y` determine the stem locations. `Z` determines the marker heights.

Specify Stem Locations with Matrix Inputs

Create a 3-D stem plot with matrix data and specify the stem locations in the *xy*-plane.

```
figure
[X,Y] = meshgrid(0:.1:1);
Z = exp(X+Y);
stem3(X,Y,Z)
```

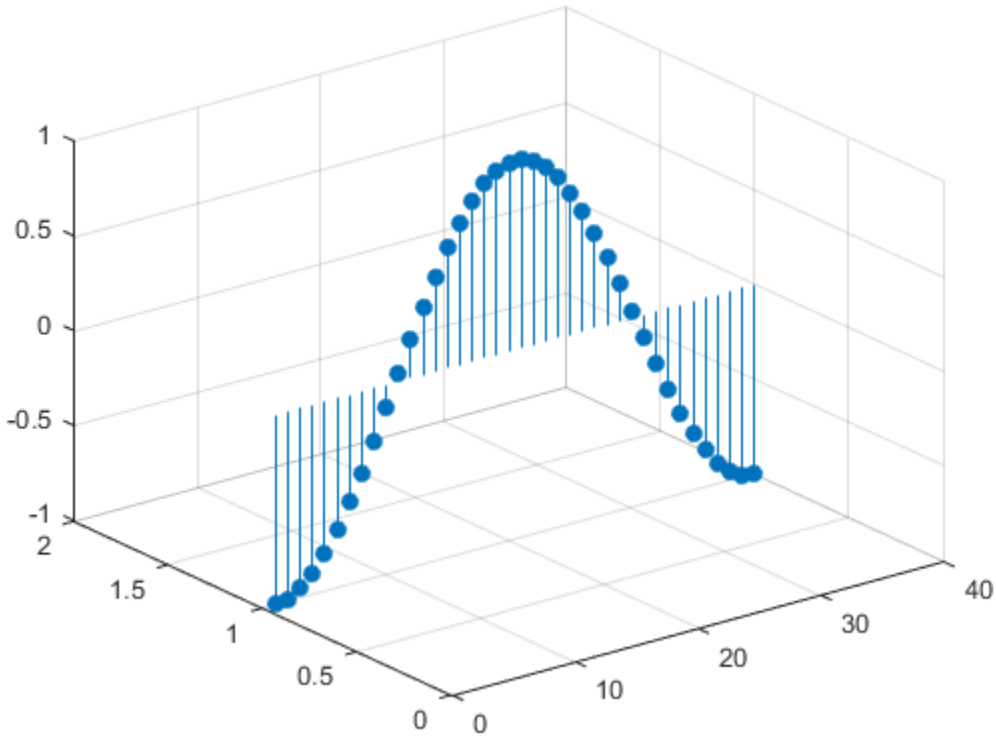


X and Y determine the stem locations. Z determines the marker heights.

Fill in Markers

Create a 3-D stem plot of cosine values between $-\pi$ and π and fill in the markers.

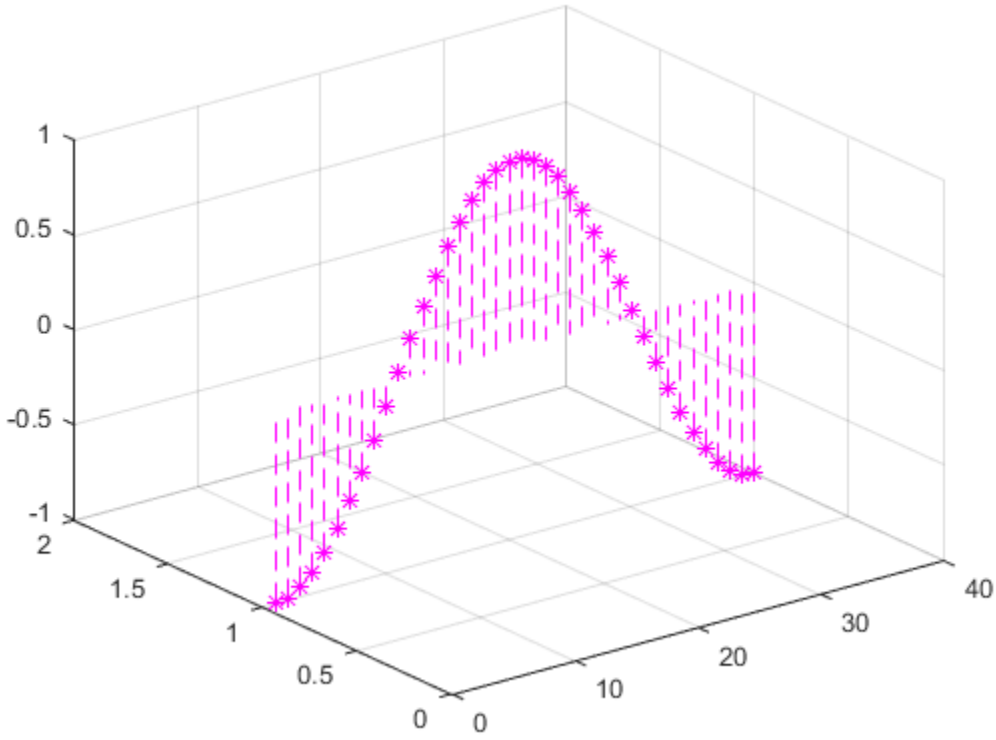
```
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'filled')
```



Line Style, Marker Symbol, and Color Options

Create a 3-D stem plot of cosine values between $-\pi$ and π. Use a dashed line style for the stem, set the marker symbols to stars, and set the color to magenta.

```
figure
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'--*m')
```



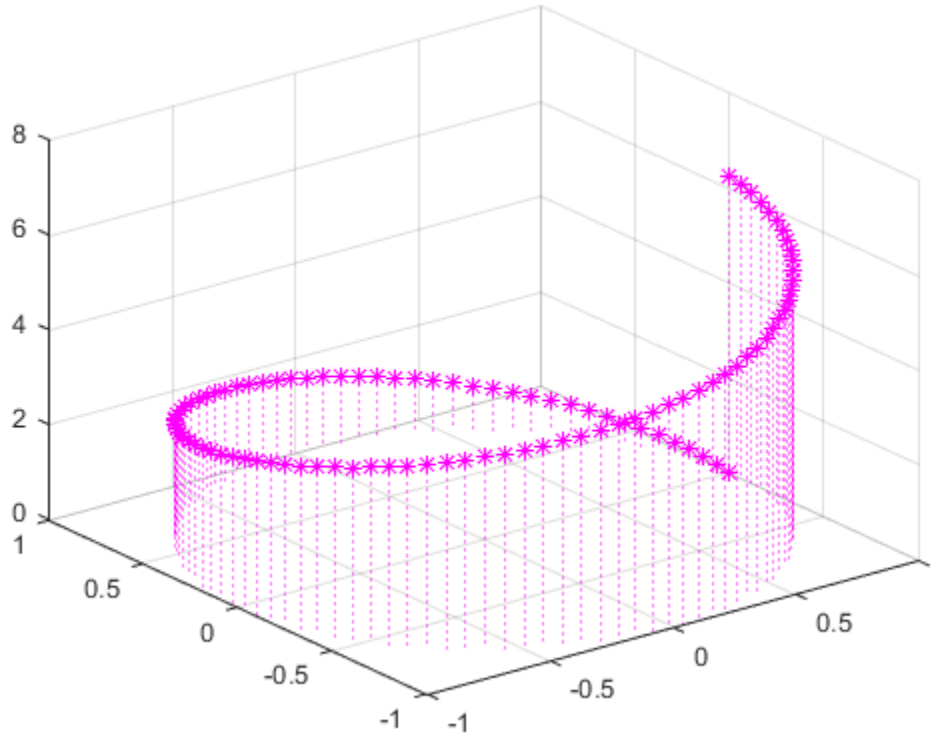
To specify only two of the three `LineSpec` options, omit the third option from the string. For example, `'*m'` sets the marker symbol and the color and uses the default line style.

Line Style, Marker Symbol, and Color Options

Create a 3-D stem plot and specify the stem locations along a circle. Set the stem to a dotted line style, the marker symbols to stars, and the color to magenta.

```
figure
```

```
theta = linspace(0,2*pi);
X = cos(theta);
Y = sin(theta);
Z = theta;
stem3(X,Y,Z,':*m')
```



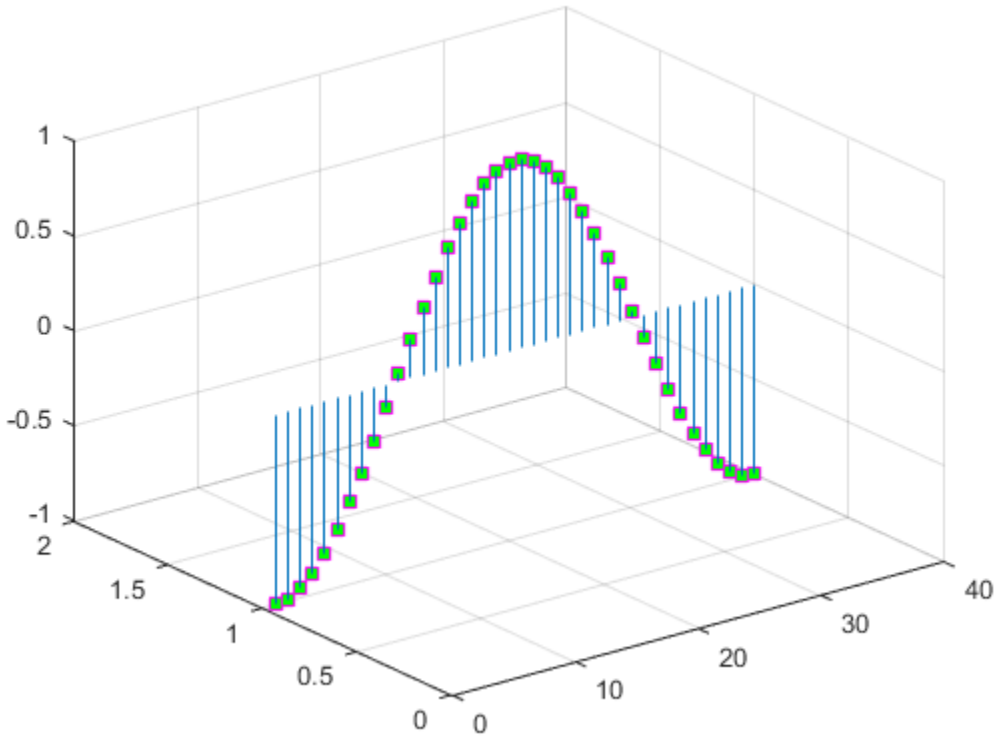
X and Y determine the stem locations. Z determines the marker heights.

Additional Style Options

Create a 3-D stem plot of cosine values between $-\pi$ and π. Set the marker symbols to squares with green faces and magenta edges.

```
figure
```

```
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'Marker','s',...
      'MarkerEdgeColor','m',...
      'MarkerFaceColor','g')
```



Axes Handles

Specify the axes for a 3-D stem plot.

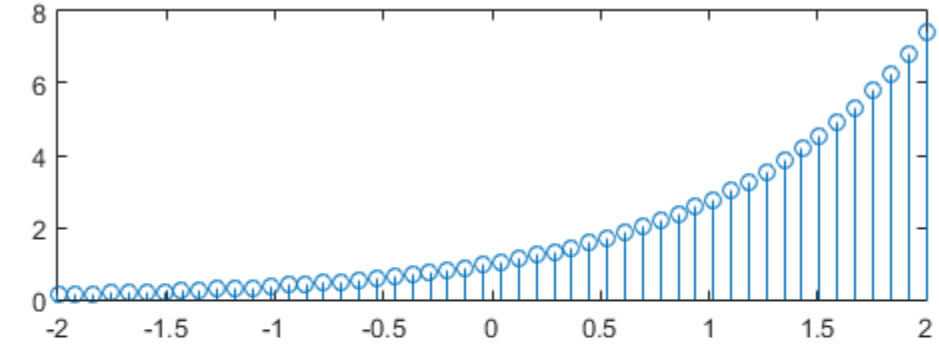
Define vectors `X`, `Y` and `Z`.

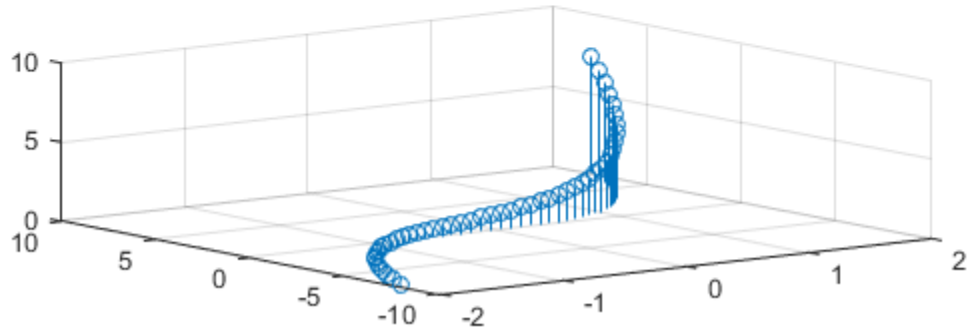
```
X = linspace(-2,2,50);
```

```
Y = X.^3;
Z = exp(X);
```

Create a figure with two subplots and return the handles to each axes, `s(1)` and `s(2)`. Plot a 3-D stem plot in the lower subplot by referring to its axes handle, `s(2)`. For comparison, plot a 2-D stem plot in the upper subplot by referring to its axes handle, `s(1)`.

```
figure
s(1) = subplot(2,1,1);
s(2) = subplot(2,1,2);

stem(s(1),X,Z)
stem3(s(2),X,Y,Z)
```

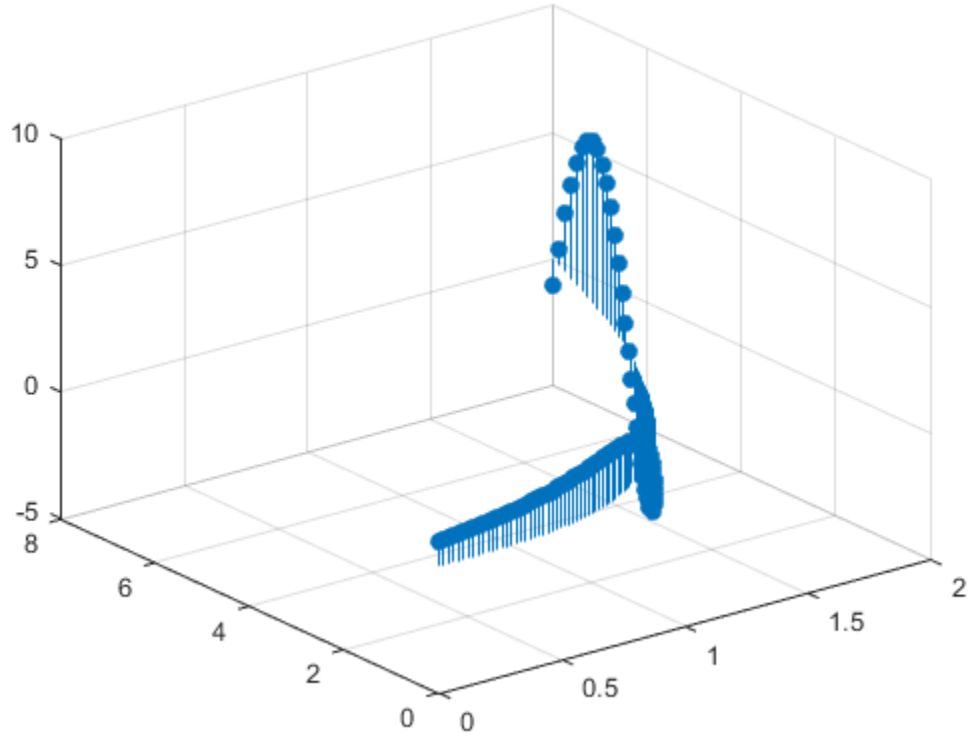




Modify Stem Series After Creation

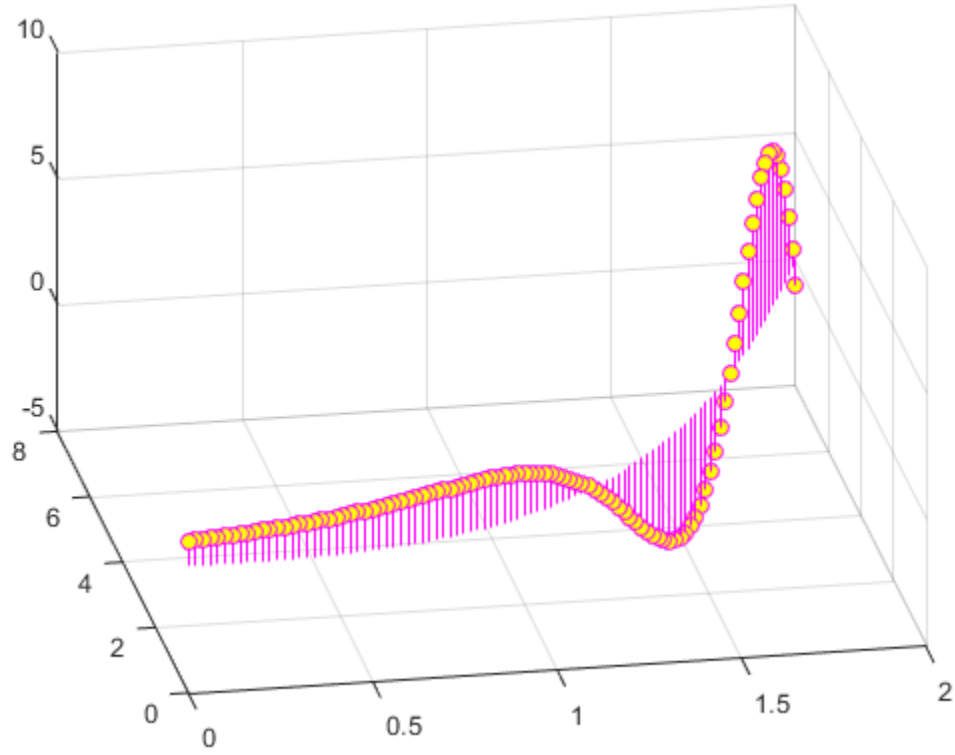
Create a 3-D stem plot and return the stem series object.

```
X = linspace(0,2);
Y = X.^3;
Z = exp(X).*cos(Y);
h = stem3(X,Y,Z,'filled');
```



Change the color to magenta and set the marker face color to yellow. Use `view` to adjust the angle of the axes in the figure. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
h.Color = 'm';
h.MarkerFaceColor = 'y';
view(-10,35)
```



- “Combine Stem Plot and Line Plot”

Input Arguments

Z — Data sequence to display

vector or matrix

Data sequence to display, specified as a vector or matrix. `stem3` plots each element in `Z` as a stem extending from the *xy*-plane and terminating at the data value.

- If `Z` is a row vector, `stem3` plots all elements against the same *y* value at equally spaced *x* values.

- If `Z` is a column vector, `stem3` plots all elements against the same *x* value at equally spaced *y* values.
- If `Z` is a matrix, `stem3` plots each row of `Z` against the same *y* value at equally spaced *x* values.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

`X` — Locations to plot values of `Z`

vector or matrix

Locations to plot values of`Z`, specified as a vector or a matrix. Inputs `X`, `Y` and `Z` must be vectors or matrices of the same size.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

`Y` — Locations to plot values of `Z`

vector or matrix

Locations to plot values of `Z`, specified as a vector or a matrix. Inputs `X`, `Y` and `Z` must be vectors or matrices of the same size.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

`LineSpec` — Line style, marker symbol, and color

string

Line style, marker symbol, and color, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

Data Types: `char`

`ax` — Axes object

axes object

Axes object. If you do not specify an axes, then `stem` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'LineStyle',':','MarkerFaceColor','red'` plots the stem as a dotted line and sets the marker face color to red.

The properties listed here are only a subset. For a complete list, see Stem Series Properties.

`'LineStyle'` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line style strings listed in this table.

String	Line Style	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- - - - -
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	_._._._.-
`'none'`	No line	No line

`'LineWidth'` — Line width of stem and marker edge
`0.5` (default) | positive value

Line width of stem and marker edge, specified as a positive value in point units.

Example: `0.75`

`'Color'` — Stem color
`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Stem color, specified as an RGB triplet, a color string, or `'none'`. If you specify the `Color` as `'none'`, then the stems are invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`'Marker'` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the marker strings listed in this table.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle

String	Marker Symbol
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

Example: '+'

Example: 'diamond'

'MarkerSize' — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: 10

'MarkerEdgeColor' — Marker outline color

'auto' (default) | 'none' | RGB triplet | color string

Marker outline color, specified as one of these values:

- 'auto' — Use the same color as the Color property.
- 'none' — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]

Long Name	Short Name	RGB Triplet
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.5 0.5 0.5]

Example: 'blue'

'MarkerFaceColor' — Marker fill color
'none' (default) | 'auto' | RGB triplet | color string

Marker fill color, specified as one of these values:

- 'none' — Use no color, which makes the interior invisible.
- 'auto' — Use the same color as the Color property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.3 0.2 0.1]

Example: `'green'`

Output Arguments

h — Stem series object
scalar

Stem series object, specified as a scalar. This is a unique identifier, which you can use to modify the properties of the stem series after it is created.

See Also

Functions
`bar` | `plot` | `stairs` | `stem`

Properties
Stem Series Properties

Introduced before R2006a

Stem Series Properties

Control stem series appearance and behavior

Stem series properties control the appearance and behavior of a stem series object. By changing property values, you can modify certain aspects of the stem series.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = stem(1:10);
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Stems

`Color` — Stem color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Stem color, specified as an RGB triplet, a color string, or `'none'`. If you specify the `Color` as `'none'`, then the stems are invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — Stem line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Stem line style, specified as one of the strings listed in this table.

String	Line Style
`'-'`	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line
`'none'`	No stems

Example: `'--'`

`LineWidth` — Line width of stem and marker edge

`0.5` (default) | scalar numeric value greater than 0

Line width of stem and marker edge, specified as a scalar numeric value greater than 0 in point units. The default line width is `0.5` points.

Example: `0.75`

Markers

`Marker` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the marker strings listed in this table.

String	Marker Symbol
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk

String	Marker Symbol
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

Example: '+'

Example: 'diamond'

MarkerSize — Marker size

6 (default) | positive value

Marker size, specified as a positive value in point units.

Example: 10

MarkerEdgeColor — Marker outline color

'auto' (default) | 'none' | RGB triplet | color string

Marker outline color, specified as one of these values:

- 'auto' — Use the same color as the Color property.
- 'none' — Use no color, which makes unfilled markers invisible.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | color string

Marker fill color, specified as one of these values:

- `'none'` — Use no color, which makes the interior invisible.
- `'auto'` — Use the same color as the `Color` property for the axes.
- RGB triplet or color string — Use the specified color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]

Long Name	Short Name	RGB Triplet
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0.3 0.2 0.1]

Example: 'green'

Baseline

BaseValue — Baseline location

0 (default) | numeric scalar value

Baseline location, specified as a numeric scalar value.

ShowBaseLine — Baseline visibility

'on' (default) | 'off'

Baseline visibility, specified as one of these values:

- 'on' — Show the baseline.
- 'off' — Hide the baseline.

BaseLine — Baseline

baseline object

Baseline object. For a list of baseline properties, see Baseline Properties.

Data

XData — Values along x-axis

[ ] (default) | vector

Values along the *x*-axis, specified as a vector.

- For 2-D stem charts, the input argument X to the stem function determines the *x*-values. If you do not specify X, then stem uses the indices of YData as the *x*-values. XData and YData must have equal lengths.

- For 3-D stem charts, the input argument `X` to the `stem3` function determines the *x*-values. If you do not specify `X`, then `stem3` uses the indices of `ZData` as the *x*-values. `XData`, `YData`, and `ZData` must have equal lengths.

`YData` — Values along y-axis

`[ ]` (default) | vector

Values along the *y*-axis, specified as a vector.

- For 2-D stem charts, the input argument `Y` to the `stem` function determines the *y*-values. `YData` defines the stem heights. `XData` and `YData` must have equal lengths.
- For 3-D stem charts, the input argument `Y` to the `stem3` function determines the *y*-values. `YData` defines the locations of the stems along the *y*-axis. `XData`, `YData`, and `ZData` must have equal lengths.

`ZData` — Values along z-axis

`[ ]` (default) | vector

Values along the *z*-axis, specified as a vector.

- For 2-D stem charts, `ZData` is empty by default.
- For 3-D stem charts, the input argument `Z` to the `stem3` function determines the *z*-values. `XData`, `YData`, and `ZData` must have equal lengths.

`XDataSource` — Variable linked to `XData`

`''` (default) | string containing MATLAB workspace variable name

Variable linked to `XData`, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty string, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

YDataSource — Variable linked to YData

'' (default) | string containing MATLAB workspace variable name

Variable linked to YData, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the YData.

By default, there is no linked variable so the value is an empty string, ''. If you link a variable, then MATLAB does not update the YData values immediately. To force an update of the data values, use the refreshdata function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: 'y'

ZDataSource — Variable linked to ZData

'' (default) | string containing MATLAB workspace variable name

Variable linked to ZData, specified as a string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the ZData.

By default, there is no linked variable so the value is an empty string, ''. If you link a variable, then MATLAB does not update the ZData values immediately. To force an update of the data values, use the refreshdata function.

Note: If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: 'z'

XDataMode — Selection mode for XData

'auto' (default) | 'manual'

Selection mode for XData, specified as one of these values:

- 'auto' — Use the indices of the values in YData (or ZData for 3-D plots).

- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

Visibility

`Visible` — Visibility of stem series

`'on'` (default) | `'off'`

Visibility of stem series, specified as one of these values:

- `'on'` — Display the stem series.
- `'off'` — Hide the stem series without deleting it. You still can access the properties of an invisible stem series object.

`Clipping` — Clipping of stem series to axes limits

`'on'` (default) | `'off'`

Clipping of stem series to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the stem series that are outside the axes limits.
- `'off'` — Display the entire stem series, even if parts of it appear outside the axes limits. Parts of the stem series might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the stem series that is larger than the original plot.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.

- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object

`'stem'` (default)

Type of graphics object, returned as `'stem'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

`Tag` — Tag to associate with stem series

`''` (default) | string

Tag to associate with the stem series, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

Data Types: `char`

`UserData` — Data to associate with stem series

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the stem series object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

`DisplayName` — Text used by legend

`''` (default) | string

Text used by the legend, specified as a string. The text appears next to an icon of the stem series.

Example: `'Text Description'`

For multiline text, create the string using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a string. The string has the form `'dataN'`, where `N` is the number assigned to the stem series object based on its location in the list of legend entries.

If you edit interactively the string in an existing legend, then MATLAB updates the `DisplayName` property to the edited string.

`Annotation` — Legend icon display style

`Annotation` object

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the stem series from a legend.

1. Query the `Annotation` property to get the `Annotation` object.

2 Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.

3 Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:

- `'on'` — Include the stem series object in the legend as one entry (default).
- `'off'` — Do not include the stem series object in the legend.
- `'children'` — Include only children of the stem series object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Parent/Child

`Parent` — Parent of stem series

axes object | group object | transform object

Parent of stem series, specified as an axes, group, or transform object.

`Children` — Children of stem series

empty `GraphicsPlaceholder` array

The stem series has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of stem series object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The stem series object handle is always visible.
- `'off'` — The stem series object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The stem series object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the stem series at the command-line, but allows callback functions to access it.

If the stem series object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the stem series. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The stem series object — You can access properties of the stem series object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

UIContextMenu — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the stem series. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

Selected — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the stem series when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the stem series.
- `'off'` — Not selected.

SelectionHighlight — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

PickableParts — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks when visible. The Visible property must be set to `'on'` and you must click a part of the stem series that has a defined color. You

cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The HitTest property determines if the stem series responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the stem series passes the click to the object below it in the current view of the figure window. The HitTest property of the stem series has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the stem series. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the stem series that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the stem series object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`HitTestArea` — (removed) Extents of clickable area for stem series

`'off'` (default) | `'on'`

Note: `HitTestArea` has been removed. Use PickableParts instead.

Extents of clickable area for stem series, specified as one of these values:

- `'off'` — Click the stem series plot to select it. This is the default value.
- `'on'` — Click anywhere within the extent of the stem series plot to select it, that is, anywhere within the rectangle that encloses the stem series plot.

Example: `'off'`

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback. Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the stem series is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the stem series tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the stem series. Setting the `CreateFcn` property on an existing stem series has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during stem series creation. MATLAB executes the callback after creating the stem series and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The stem series object — You can access properties of the stem series object from within the callback function. You also can access the stem series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.

- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the stem series. MATLAB executes the callback before destroying the stem series so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The stem series object — You can access properties of the stem series object from within the callback function. You also can access the stem series object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of stem series

`'off'` (default) | `'on'`

Deletion status of stem series, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the stem series begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the stem series no longer exists.

Check the value of the BeingDeleted property to verify that the stem series is not about to be deleted before querying or modifying it.

See Also

`stem` | `stem3`

More About

- "Access Property Values"
- "Graphics Object Properties"

stopasync

Stop asynchronous read and write operations

Syntax

```
stopasync(obj)
```

Description

`stopasync(obj)` stops any asynchronous read or write operation that is in progress for the serial port object, `obj`.

More About

Tips

You can write data asynchronously using the `fprintf` or `fwrite` function. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual`.
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

See Also

`fwrite` | `readasync` | `fprintf` | `ReadAsyncMode` | `TransferStatus`

Introduced before R2006a

str2double

Convert string to double-precision value

Syntax

```
X = str2double('str')
X = str2double(C)
```

Description

`X = str2double('str')` converts the string `str`, which should be an ASCII character representation of a real or complex scalar value, to the MATLAB double-precision representation. The string can contain digits, a comma (thousands separator), a decimal point, a leading `+` or `-` sign, an `e` preceding a power of 10 scale factor, and an `i` for a complex unit.

If `str` does not represent a valid scalar value, `str2double` returns `NaN`.

`X = str2double(C)` converts the strings in the cell array of strings `C` to double precision. The matrix `X` returned will be the same size as `C`.

Examples

Here are some valid `str2double` conversions, for numbers and cell arrays.

```
str2double('123.45e7')
str2double('123 + 45i')
str2double('3.14159')
str2double('2.7i - 3.14')
str2double({'2.71' '3.1415'})
str2double('1,200.34')
```

See Also

`char` | `hex2num` | `num2str` | `str2num` | `cast`

Introduced before R2006a

str2func

Construct function handle from function name string

Syntax

```
str2func('str')
```

Description

`str2func('str')` constructs a function handle `fhandle` for the function named in the string `'str'`. The contents of `str` can be the name of a file that defines a MATLAB function, or the name of an anonymous function.

You can create a function handle `fh` using any of the following four methods:

- Create a handle to a named function:

  ```
  fh = @functionName;
  fh = str2func('functionName');
  ```

- Create a handle to an anonymous function:

  ```
  fh = @(x)functionDef(x);
  fh = str2func('@(x)functionDef(x)');
  ```

You can create an array of function handles from strings by creating the handles individually with `str2func`, and then storing these handles in a cell array.

Examples

Example 1

To convert the string, `'sin'`, into a handle for that function, type

```
fh = str2func('sin')
fh =
```

```
    @sin
```

Example 2

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using `str2func`. The example below passes the variable, `funcname`, to function `makeHandle`, which then creates a function handle. Here is the function:

```
function fh = makeHandle(funcname)
fh = str2func(funcname);
```

This is the code that calls `makeHandle` to construct the function handle:

```
makeHandle('sin')
ans =
    @sin
```

Example 3

To call `str2func` on a cell array of strings, use the `cellfun` function. This returns a cell array of function handles:

```
fh_array = cellfun(@str2func, {'sin' 'cos' 'tan'}, ...
                   'UniformOutput', false);

fh_array{2}(5)
ans =
   0.2837
```

Example 4

In the following example, the `myminbnd` function expects to receive either a function handle or string in the first argument. If you pass a string, `myminbnd` constructs a function handle from it using `str2func`, and then uses that handle in a call to `fminbnd`:

```
function myminbnd(fhandle, lower, upper)
if ischar(fhandle)
   disp 'converting function string to function handle ...'
   fhandle = str2func(fhandle);
end
fminbnd(fhandle, lower, upper)
```

Whether you call `myminbnd` with a function handle or function name string, the function can handle the argument appropriately:

```
myminbnd('humps', 0.3, 1)
converting function string to function handle ...
ans =
    0.6370
```

Example 5

The `dirByType` function shown here creates an anonymous function called `dirCheck`. What the anonymous function does depends upon the value of the `dirType` argument passed in to the primary function. The example demonstrates one possible use of `str2func` with anonymous functions:

```
function dirByType(dirType)
switch(dirType)
   case 'class', leadchar = '@';
   case 'package', leadchar = '+';
   otherwise disp('ERROR: Unrecognized type'), return;
end

dirfile = @(fs)isdir(fs.name);
dirCheckStr = ['@(fs)strcmp(fs.name(1,1),''', leadchar, ''')'];
dirCheckFun = str2func(dirCheckStr);
s = dir;   filecount = length(s);

for k=1:filecount
   fstruct = s(k);
   if dirfile(fstruct) && dirCheckFun(fstruct)
       fprintf('%s folder: %s\n', dirType, fstruct.name)
   end
end
```

Generate a list of class and package folders:

```
dirByType('class')
class folder: @Point
class folder: @asset
class folder: @bond

dirByType('package')
package folder: +containers
package folder: +event
```

```
package folder: +mypkg
```

More About

Tips

Nested functions are not accessible to `str2func`. To construct a function handle for a nested function, you must use the function handle constructor, `@`.

Any variables and their values originally stored in a function handle when it was created are lost if you convert the function handle to a string and back again using the `func2str` and `str2func` functions.

See Also

`function_handle` | `func2str` | `functions`

Introduced before R2006a

str2mat

Form blank-padded character matrix from strings

Note: `str2mat` is not recommended. Use `char` instead.

Syntax

```
S = str2mat(T1, T2, T3, ...)
```

Description

`S = str2mat(T1, T2, T3, ...)` forms the matrix `S` containing the text strings `T1, T2, T3, ...` as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, `Ti`, can itself be a string matrix. This allows the creation of arbitrarily large string matrices. Empty strings are significant.

Examples

```
x = str2mat('36842', '39751', '38453', '90307');

whos x
  Name      Size         Bytes  Class

  x         4x5             40  char array

x(2,3)

ans =

    7
```

More About

Tips

`str2mat` differs from `strvcat` in that empty strings produce blank rows in the output. In `strvcat`, empty strings are ignored.

See Also

`char`

Introduced before R2006a

str2num

Convert string to number

Syntax

```
x = str2num('str')
[x, status] = str2num('str')
```

Description

Note `str2num` uses the `eval` function to convert the input argument. Side effects can occur if the string contains calls to functions. Using `str2double` can avoid some of these side effects.

`x = str2num('str')` converts the string `str`, which is an ASCII character representation of a numeric value, to numeric representation. `str2num` also converts string matrices to numeric matrices. If the input string does not represent a valid number or matrix, `str2num(str)` returns the empty matrix in `x`.

The input string can contain one or more numbers separated by spaces, commas, or semicolons, such as '`5`', '`10,11,12`', or '`5,10;15,20`'. In addition to numerical values and delimiters, the input string can also include a decimal point, leading + or - signs, the letter `e` or `d` preceding a power of 10 scale factor, or the letter `i` or `j` indicating a complex or imaginary number.

The following table shows several examples of valid inputs to `str2num`:

String Input	Numeric Output	Output Class
`'500'`	500	`double`
`'500 250 125 67'`	500 250 125 67	1-by-4 row vector of `double`
`'500; 250; 125; 62.5'`	500.0000 250.0000 125.0000 62.5000	4-by-1 column vector of `double`

String Input	Numeric Output	Output Class
'1 23 6 21; 53:56'	1 23 6 21 53 54 55 56	2-by-4 matrix of double
'12e-3 5.9e-3'	0.0120 0.0059	1-by-2 row vector of double
'uint16(500)'	500	16–bit unsigned integer

Note: str2num does not operate on cell arrays. To convert a cell array of strings to a numeric value, use str2double.

If the input string does not represent a valid number or matrix, str2num(str) returns the empty matrix in x.

[x, status] = str2num('str') returns the status of the conversion in logical status, where status equals logical 1 (true) if the conversion succeeds, and logical 0 (false) otherwise.

Space characters can be significant. For instance, str2num('1+2i') and str2num('1 + 2i') produce x = 1+2i, while str2num('1 +2i') produces x = [1 2i]. You can avoid these problems by using the str2double function.

Examples

Input a character string that contains a single number. The output is a scalar double:

```
A = str2num('500')
A =
   500

class(A)
ans =
   double
```

Repeat this operation, but this time using an unsigned 16–bit integer:

```
A = str2num('uint16(500)')
A =
    500
```

```
class(A)
ans =
   uint16
```

Try three different ways of specifying a row vector. Each returns the same answer:

```
str2num('2 4 6 8')              % Separate with spaces.
ans =
     2     4     6     8

str2num('2,4,6,8')              % Separate with commas.
ans =
     2     4     6     8

str2num('[2 4 6 8]')            % Enclose in brackets.
ans =
     2     4     6     8
```

Note that the first two of these commands do not need the MATLAB square bracket operator to create a matrix. The `str2num` function inserts the brackets for you if they are needed.

Use a column vector this time:

```
str2num('2; 4; 6; 8')
ans =
     2
     4
     6
     8
```

And now a 2-by-2 matrix:

```
str2num('2 4; 6 8')
ans =
     2     4
     6     8
```

See Also

`num2str` | special characters | `str2double` | `hex2num` | `sscanf` | `sparse` | `char` | `cast`

Introduced before R2006a

strcat

Concatenate strings horizontally

Syntax

```
s = strcat(s1,...,sN)
```

Description

`s = strcat(s1,...,sN)` horizontally concatenates strings `s1,...,sN`. Each input argument can be a single string, a collection of strings in a cell array, or a collection of strings in a character array.

If any input argument is a cell array, the result is a cell array of strings. Otherwise, the result is a character array.

For character array inputs, `strcat` removes trailing ASCII white-space characters: space, tab, vertical tab, newline, carriage return, and form feed. For cell array inputs, `strcat` does not remove trailing white space.

Examples

Concatenate Two Strings

```
s1 = 'Good';
s2 = 'morning';
s = strcat(s1,s2)
```

```
s =

Goodmorning
```

Concatenate Two Cell Arrays

```
s1 = {'abcde','fghi'};
s2 = {'jkl','mn'};
```

```
s = strcat(s1,s2)

s =

    'abcdejkl'    'fghimn'
```

Concatenate Two Cell Arrays with Scalar Cell Array

```
firstnames = {'Abraham'; 'George'};
lastnames = {'Lincoln'; 'Washington'};
names = strcat(lastnames, {', '}, firstnames)

names =

    'Lincoln, Abraham'
    'Washington, George'
```

Input Arguments

`s1, . . . , sN` — Input strings

character arrays | cell array of strings

Input strings, specified as cell arrays of strings or character arrays with the same number of rows. When combining nonscalar cell arrays and multirow character arrays, cell arrays must be column vectors with the same number of rows as the character arrays.

Data Types: `char` | `cell`

More About

Tips

- Strings also can be concatenated using left and right square brackets.

  ```
  s1 = 'Good ';
  s2 = 'Morning';
  s = [s1 s2]

  s =

  Good Morning
  ```

See Also

`cat` | `cellstr` | `horzcat` | `strjoin` | `vertcat`

Introduced before R2006a

strcmp

Compare strings

Syntax

```
tf = strcmp(s1,s2)
```

Description

`tf = strcmp(s1,s2)` compares strings `s1` and `s2` and returns 1 (true) if the two are identical. Otherwise, `strcmp` returns 0 (false). Strings are considered identical if the size and content of each are the same. The return result, `tf`, is of data type `logical`.

Inputs can be combinations of single strings, strings in scalar cell arrays, character arrays, and cell arrays of strings.

Examples

Compare Two Strings

Compare two different strings.

```
s1 = 'Yes';
s2 = 'No';
tf = strcmp(s1,s2)
```

```
tf =

     0
```

`strcmp` returns 0 because the two strings are not equal.

Compare two equal strings.

```
s1 = 'Yes';
s2 = 'Yes';
tf = strcmp(s1,s2)
```

```
tf =
```

```
     1
```

`strcmp` returns 1 because the two strings are equal.

Compare String and Cell Array of Strings

Compare a string, `'upon'`, to each element of a cell array of strings.

```
s1 = 'upon';
s2 = {'Once','upon';
      'a','time'};
tf = strcmp(s1,s2)
```

```
tf =

     0     1
     0     0
```

There is only one occurrence of string `s1` in array `s2`, and it occurs at element `s2(1,2)`.

Compare Two Cell Arrays of Strings

Compare each element in a cell array of strings to the corresponding element in a second cell array of strings.

```
s1 = {'Time','flies','when';
      'you''re','having','fun.'};
s2 = {'Time','drags','when';
      'you''re','anxiously','waiting.'};
tf = strcmp(s1,s2)
```

```
tf =

     1     0     1
     1     0     0
```

There are three instances of equal strings between `s1` and `s2`. These are `'Time'` at indices `(1,1)`, `'when'` at indices`(1,3)`, and `'you''re'` at indices `(2,1)`.

Input Arguments

`s1,s2` — Input strings

character array | cell array of strings

Input strings, specified as a cell array of strings, or character array with one or more rows. The order of the inputs does not affect the comparison results.

- If both s1,s2 are nonscalar cell arrays, then s1,s2 must be the same size.
- If both s1,s2 are character arrays, then s1,s2 can have different numbers of rows.
- When comparing a nonscalar cell array and a multirow character array, the cell array must be a column vector with the same number of rows as the character array.

Data Types: `char` | `cell`

Output Arguments

`tf` — True or false

1 | 0 | logical array

True or false result, returned as a 1 or 0 of data type `logical`.

- If both inputs are character arrays, `tf` is a scalar.
- If at least one input is a cell array of strings, `tf` is an array the same size as the cell array.
- If `s1` and `s2` are a string in a scalar cell and a character array with multiple rows, then `tf` is an `n`-by-1 array, where `n` is the number of rows in the character array.

More About

Tips

- Use the `strcmp` function for comparison of strings. If you use `strcmp` on numeric arrays, it always returns 0.
- For case-insensitive string comparison, use `strcmpi` instead of `strcmp`.
- The value that `strcmp` returns is not the same as the C language convention.

See Also

`regexp` | `regexpi` | `strcmpi` | `strfind` | `strncmp` | `strncmpi`

Introduced before R2006a

strcmpi

Compare strings (case insensitive)

Syntax

```
TF = strcmpi(string,string)
TF = strcmpi(string,cellstr)
TF = strcmpi(cellstr,cellstr)
```

Description

`TF = strcmpi(string,string)` compares two strings for equality, ignoring any differences in letter case. The strings are considered to be equal if the size and content of each are the same. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.

`TF = strcmpi(string,cellstr)` compares a string with each element of a cell array of strings, ignoring letter case. The function returns a logical array the same size as the `cellstr` input in which logical 1 represents equality. The order of the input arguments is not important.

`TF = strcmpi(cellstr,cellstr)` compares each element of one cell array of strings with the same element of the other, ignoring letter case. The function returns a logical array the same size as the input arrays.

Input Arguments

`string`

A single character string or n-by-1 array of strings.

`cellstr`

A cell array of strings.

Output Arguments

TF

When both inputs are character arrays, `TF` is a scalar logical value. This value is logical 1 (`true`) if the size and content of both arrays are equal, and logical 0 (`false`) if they are not.

When either or both inputs are a cell array of strings, `TF` is an array of logical ones and zeros. This array is the same size as the input cell array(s), and contains logical 1 (`true`) for those elements of the input arrays that are a match, and logical 0 (`false`) for those elements that are not.

Examples

Perform a simple case-insensitive comparison of two strings:

```
strcmpi('Yes', 'No')
ans =
     0
strcmpi('Yes', 'yes')
ans =
     1
```

Create two cell arrays of strings and call `strcmpi` to compare them:

```
A = {'Handle Graphics', 'Statistics';   ...
     '  Toolboxes', 'MathWorks'};

B = {'Handle Graphics', 'Signal Processing';    ...
     'Toolboxes', 'MATHWORKS'};

match = strcmpi(A, B)
match =
     1     0
     0     1
```

The result of comparing the two cell arrays is:

- `match{1,1}` is 1 because “Handle Graphics” in `A{1,1}` matches the same text in `B{1,1}`.

- `match{1,2}` is 0 because “Statistics” in `A{1,2}` does not match “Signal Processing” in `B{1,2}`.
- `match{2,1}` is 0 because “ Toolboxes”, in `A{2,1}` contains leading space characters that are not in `B{2,1}`.
- `match{2,2}` is 1 because even though “MathWorks” in `A{2,2}` uses different letter case than “MATHWORKS” in `B{2,2}`, `strcmpi` performs the comparison without sensitivity to letter case.

More About

Tips

- The `strcmpi` function is intended for comparison of character data. When used to compare numeric data, it returns logical `0`.
- Use `strcmp` for case-sensitive string comparisons.
- Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
- The value returned by `strcmpi` is not the same as the C language convention.
- `strcmpi` supports international character sets.

See Also

`strncmpi` | `strcmp` | `strncmp` | `strfind` | `regexpi` | `regexp`

Introduced before R2006a

stream2

Compute 2-D streamline data

Syntax

```
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)
```

Description

`XY = stream2(x,y,u,v,startx,starty)` computes streamlines from vector data `u` and `v`.

The arrays `x` and `y`, which define the coordinates for `u` and `v`, must be monotonic, but do not need to be uniformly spaced. `x` and `y` must have the same number of elements, as if produced by `meshgrid`.

`startx` and `starty` define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value `XY` contains a cell array of vertex arrays.

`XY = stream2(u,v,startx,starty)` assumes the arrays `x` and `y` are defined as `[x,y] = meshgrid(1:n,1:m)` where `[m,n] = size(u)`.

`XY = stream2(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify a value, MATLAB software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream2`.

Examples

This example draws 2-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:,:,5),y(:,:,5),u(:,:,5),v(:,:,5),sx,sy));
```

More About

- Specifying Starting Points for Stream Plots

See Also

`coneplot` | `stream3` | `streamline`

Introduced before R2006a

stream3

Compute 3-D streamline data

Syntax

```
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)
XYZ = stream3(U,V,W,startx,starty,startz)
XYZ = stream3(...,options)
```

Description

`XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)` computes streamlines from vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value `XYZ` contains a cell array of vertex arrays.

`XYZ = stream3(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)` where `[M,N,P] = size(U)`.

`XYZ = stream3(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream3`.

Examples

This example draws 3-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamline(stream3(x,y,z,u,v,w,sx,sy,sz))
view(3)
```

More About

- Specifying Starting Points for Stream Plots

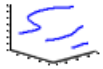
See Also

`coneplot` | `stream2` | `streamline`

Introduced before R2006a

streamline

Plot streamlines from 2-D or 3-D vector data



Syntax

```
streamline(X,Y,Z,U,V,W,startx,starty,startz)
streamline(U,V,W,startx,starty,startz)
streamline(XYZ)
streamline(X,Y,U,V,startx,starty)
streamline(U,V,startx,starty)
streamline(XY)
streamline(...,options)
streamline(axes_handle,...)
h = streamline(...)
```

Description

`streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws streamlines from 3-D vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, `startz` define the starting positions of the streamlines. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

`streamline(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)`, where `[M,N,P] = size(U)`.

`streamline(XYZ)` assumes `XYZ` is a precomputed cell array of vertex arrays (as produced by `stream3`).

`streamline(X,Y,U,V,startx,starty)` draws streamlines from 2-D vector data `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`startx` and `starty` define the starting positions of the streamlines. The output argument `h` contains a vector of line handles, one handle for each streamline.

`streamline(U,V,startx,starty)` assumes the arrays `X` and `Y` are defined as `[X,Y] = meshgrid(1:N,1:M)`, where `[M,N] = size(U)`.

`streamline(XY)` assumes `XY` is a precomputed cell array of vertex arrays (as produced by `stream2`).

`streamline(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 1000

`streamline(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of the into current axes object (`gca`).

`h = streamline(...)` returns a vector of line handles, one handle for each streamline.

Examples

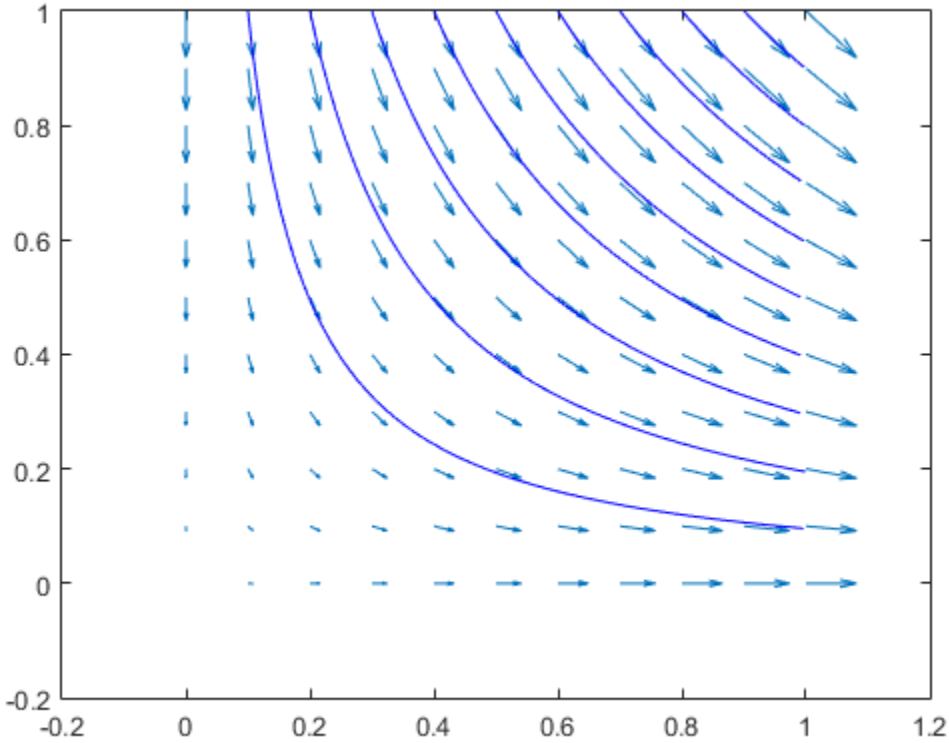
Draw Streamlines

Define arrays `x`, `y`, `u`, and `v`.

```
[x,y] = meshgrid(0:0.1:1,0:0.1:1);
u = x;
v = -y;
```

Create a quiver plot of the data. Plot streamlines that start at different points along the line $y = 1$.

```
figure
quiver(x,y,u,v)

startx = 0.1:0.1:1;
starty = ones(size(startx));
streamline(x,y,u,v,startx,starty)
```



More About

- Specifying Starting Points for Stream Plots

- Stream Line Plots of Vector Data

See Also

`coneplot` | `stream2` | `stream3` | `streamparticles` | `meshgrid`

Introduced before R2006a

streamparticles

Plot stream particles

Syntax

```
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)
streamparticles(line_handle,...)
h = streamparticles(...)
```

Description

`streamparticles(vertices)` draws stream particles of a vector field. Stream particles are usually represented by markers and can show the position and velocity of a streamline. `vertices` is a cell array of 2-D or 3-D vertices (as if produced by `stream2` or `stream3`).

`streamparticles(vertices,n)` uses `n` to determine how many stream particles to draw. The `ParticleAlignment` property controls how `n` is interpreted.

- If `ParticleAlignment` is set to `off` (the default) and `n` is greater than `1`, approximately `n` particles are drawn evenly spaced over the streamline vertices.

  If `n` is less than or equal to `1`, `n` is interpreted as a fraction of the original stream vertices; for example, if `n` is `0.2`, approximately 20% of the vertices are used.

  `n` determines the upper bound for the number of particles drawn. The actual number of particles can deviate from `n` by as much as a factor of 2.
- If `ParticleAlignment` is `on`, `n` determines the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value. The default value is `n = 1`.

`streamparticles(...,'PropertyName',PropertyValue,...)` controls the stream particles using named properties and specified values. Any unspecified properties have default values. MATLAB ignores the case of property names.

Stream Particle Properties

`Animate` — Stream particle motion [nonnegative integer]

The number of times to animate the stream particles. The default is `0`, which does not animate. `Inf` animates until you enter **Ctrl+C**.

`FrameRate` — Animation frames per second [nonnegative integer]

This property specifies the number of frames per second for the animation. `Inf`, the default, draws the animation as fast as possible. Note that the speed of the animation might be limited by the speed of the computer. In such cases, the value of `FrameRate` cannot necessarily be achieved.

`ParticleAlignment` — Align particles with streamlines [`on` | `{off}`]

Set this property to `on` to draw particles at the beginning of each streamline. This property controls how `streamparticles` interprets the argument `n` (number of stream particles).

Stream particles are primitive line objects. In addition to stream particle properties, you can specify any line property, such as `Marker`. `streamparticles` sets the following line properties when called.

Line Property	Value Set by streamparticles
`LineStyle`	`'none'`
`Marker`	`'o'`
`MarkerEdgeColor`	`'none'`
`MarkerFaceColor`	`'red'`

You can override any of these properties by specifying a property name and value as arguments to `streamparticles`. For example, this statement uses RGB values to set the `MarkerFaceColor` to medium gray:

```
streamparticles(vertices,'MarkerFaceColor',[.5 .5 .5])
```

`streamparticles(line_handle,...)` uses the line object identified by `line_handle` to draw the stream particles.

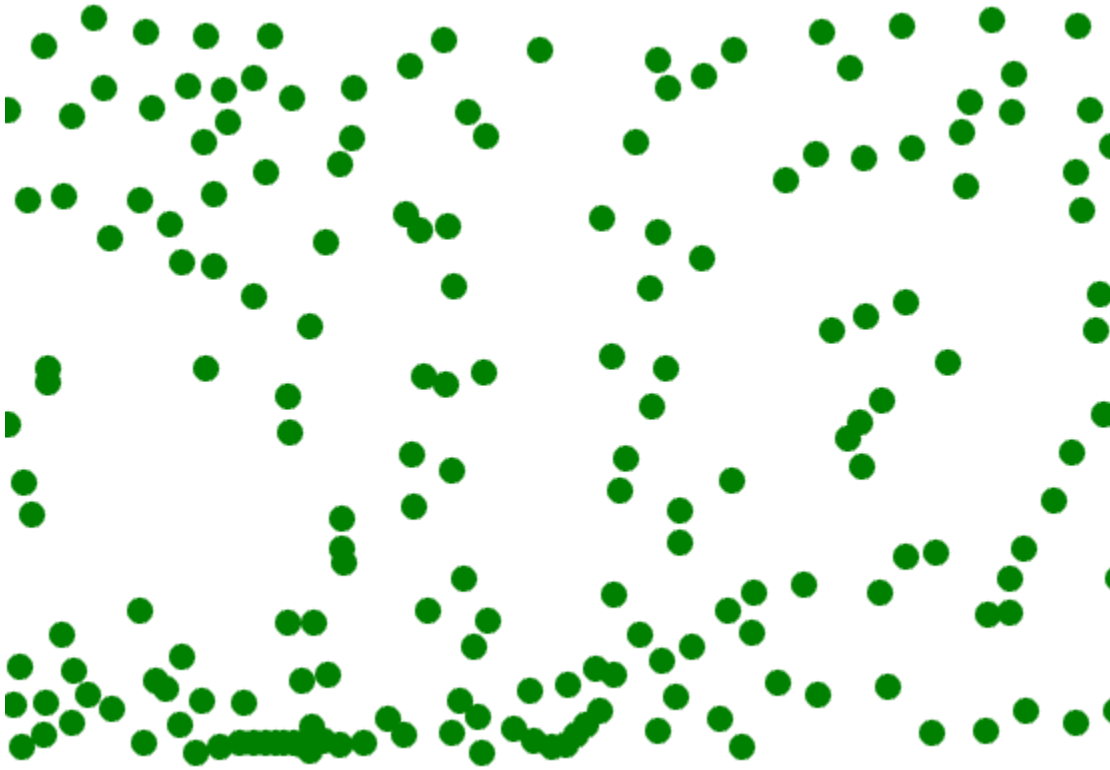
`h = streamparticles(...)` returns a vector of handles to the primitive line objects it creates. For a list of properties, see Primitive Line Properties.

Examples

Animate Flow Without Displaying Streamlines

This example uses streamlines in the `z = 5` plane to animate the flow along these lines with stream particles.

```
load wind
figure
daspect([1,1,1]);
view(2)
[verts,averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
sl = streamline([verts averts]);
axis tight manual off;
ax = gca;
ax.Position = [0,0,1,1];
set(sl,'Visible','off')
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.05);
zlim([4.9,5.1]);
streamparticles(iverts, 200, ...
    'Animate',15,'FrameRate',40, ...
    'MarkerSize',10,'MarkerFaceColor',[0 .5 0])
```



More About

- Creating Stream Particle Animations
- Specifying Starting Points for Stream Plots

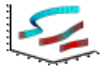
See Also

`interpstreamspeed` | `stream3` | `streamline` | `stream2`

Introduced before R2006a

streamribbon

3-D stream ribbon plot from vector volume data



Syntax

```
streamribbon(X,Y,Z,U,V,W,startx,starty,startz)
streamribbon(U,V,W,startx,starty,startz)
streamribbon(vertices,X,Y,Z,cav,speed)
streamribbon(vertices,cav,speed)
streamribbon(vertices,twistangle)
streamribbon(...,width)
streamribbon(axes_handle,...)
h = streamribbon(...)
```

Description

`streamribbon(X,Y,Z,U,V,W,startx,starty,startz)` draws stream ribbons from vector volume data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the stream ribbons at the center of the ribbons. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

`streamribbon(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamribbon(vertices,X,Y,Z,cav,speed)` assumes precomputed streamline vertices, curl angular velocity, and flow speed. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, `cav`, and `speed` are 3-D arrays.

`streamribbon(vertices,cav,speed)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(cav)`.

`streamribbon(vertices,twistangle)` uses the cell array of vectors `twistangle` for the twist of the ribbons (in radians). The size of each corresponding element of `vertices` and `twistangle` must be equal.

`streamribbon(...,width)` sets the width of the ribbons to `width`.

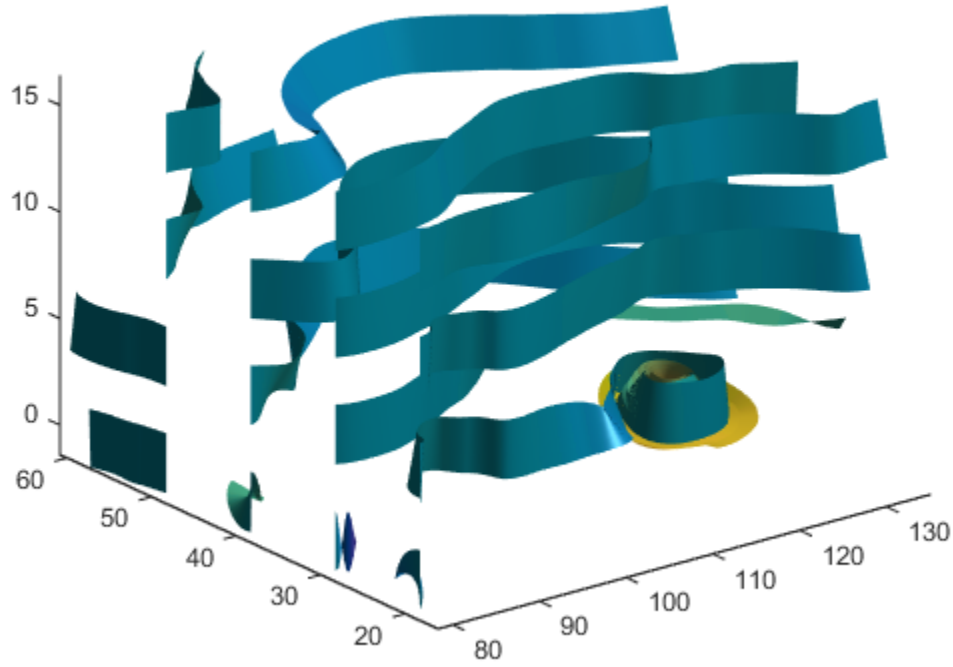
`streamribbon(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

`h = streamribbon(...)` returns a vector of handles (one per start point) to `surface` objects.

Examples

This example uses stream ribbons to indicate the flow in the `wind` data set. Inputs include the coordinates, vector field components, and starting location for the stream ribbons.

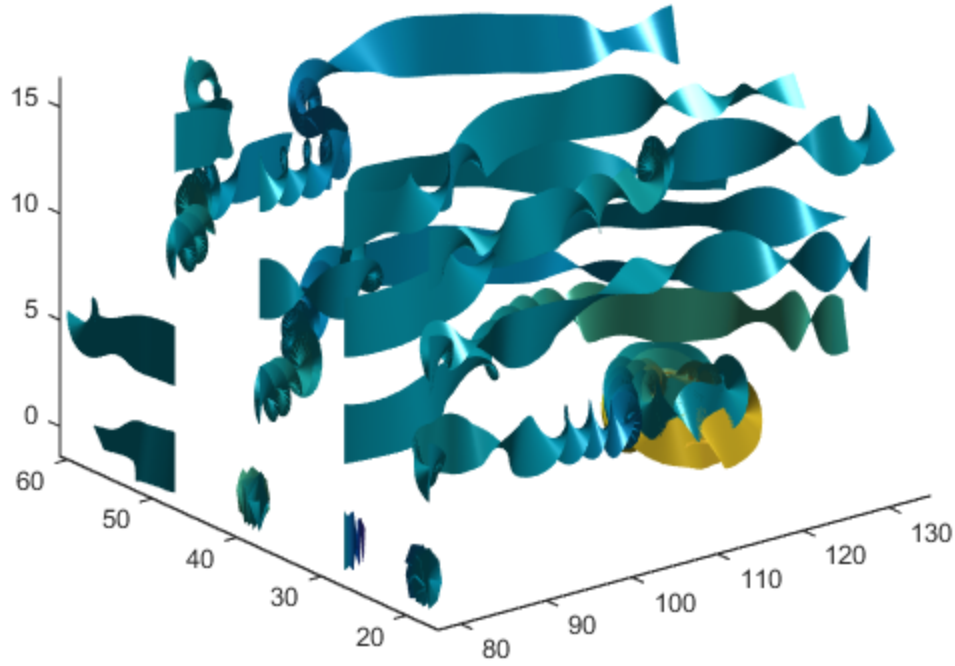
```
figure
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
% Define viewing and lighting
axis tight
shading interp;
view(3);
camlight
lighting gouraud
```



This example uses precalculated vertex data (`stream3`), curl average velocity (`curl`), and speed $\sqrt{u^2 + v^2 + w^2}$. Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to the previous example.

```
figure
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*.1;
streamribbon(verts,x,y,z,cav,spd);
% Define viewing and lighting
```

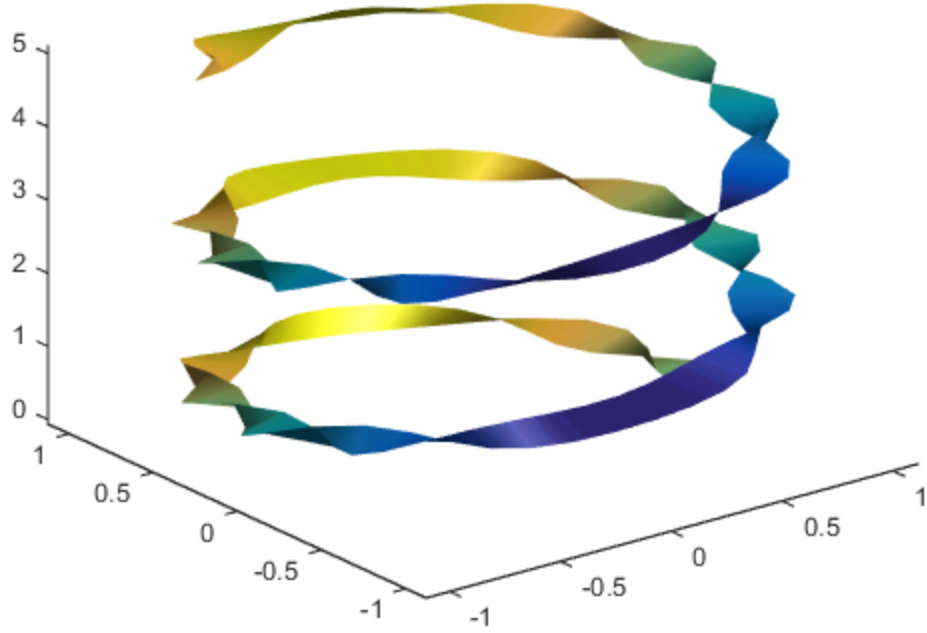
```
axis tight
shading interp
view(3)
camlight;
lighting gouraud
```



This example specifies a twist angle for the stream ribbon.

```
figure
t = 0:.15:15;
verts = {[cos(t)' sin(t)' (t/3)']};
twistangle = {cos(t)'};
streamribbon(verts,twistangle)
% Define viewing and lighting
axis tight
```

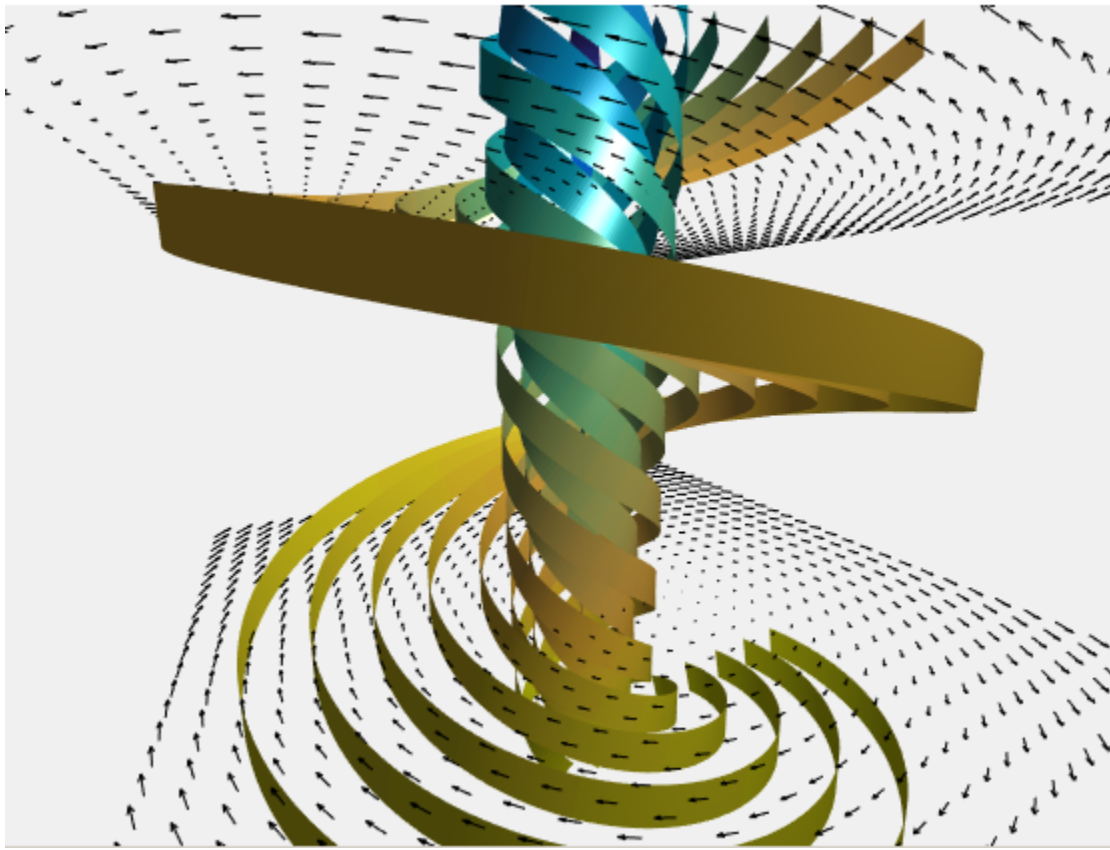
```
shading interp
view(3)
camlight
lighting gouraud
```



This example combines cone plots (`coneplot`) and stream ribbon plots in one graph.

```
figure
% Define 3-D arrays x, y, z, u, v, w
xmin = -7; xmax = 7;
ymin = -7; ymax = 7;
zmin = -7; zmax = 7;
x = linspace(xmin,xmax,30);
y = linspace(ymin,ymax,20);
z = linspace(zmin,zmax,20);
```

```
[x y z] = meshgrid(x,y,z);
u = y; v = -x; w = 0*x+1;
[cx cy cz] = meshgrid(linspace(xmin,xmax,30),...
   linspace(ymin,ymax,30),[-3 4]);
h = coneplot(x,y,z,u,v,w,cx,cy,cz,'quiver');
set(h,'Color','k');

% Plot two sets of streamribbons
[sx sy sz] = meshgrid([-1 0 1],[-1 0 1],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
[sx sy sz] = meshgrid([1:6],[0],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);

% Define viewing and lighting
shading interp
view(-30,10) ; axis off tight
camproj perspective; camva(66); camlookat;
camdolly(0,0,.5,'fixtarget')
camlight
```



More About

- "Volume Visualization"
- Displaying Curl with Stream Ribbons
- Specifying Starting Points for Stream Plots

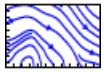
See Also

`curl` | `streamtube` | `streamline` | `stream3` | `meshgrid` | `coneplot`

Introduced before R2006a

streamslice

Plot streamlines in slice planes



Syntax

```
streamslice(X,Y,Z,U,V,W,startx,starty,startz)
streamslice(U,V,W,startx,starty,startz)
streamslice(X,Y,U,V)
streamslice(U,V)
streamslice(...,density)
streamslice(...,'arrowsmode')
streamslice(...,'method')
streamslice(axes_handle,...)
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)
```

Description

`streamslice(X,Y,Z,U,V,W,startx,starty,startz)` draws well-spaced streamlines (with direction arrows) from vector data `U`, `V`, `W` in axis aligned *x*-, *y*-, *z*-planes at the points in the vectors `startx`, `starty`, `startz`. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`. `U`, `V`, `W` must be m-by-n-by-p volume arrays.

Do not assume that the flow is parallel to the slice plane. For example, in a stream slice at a constant `z`, the `z` component of the vector field `W` is ignored when you are calculating the streamlines for that plane.

Stream slices are useful for determining where to start streamlines, stream tubes, and stream ribbons.

`streamslice(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(X,Y,U,V)` draws well-spaced streamlines (with direction arrows) from vector volume data `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`streamslice(U,V)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(...,density)` modifies the automatic spacing of the streamlines. `density` must be greater than `0`. The default value is `1`; higher values produce more streamlines on each plane. For example, `2` produces approximately twice as many streamlines, while `0.5` produces approximately half as many.

`streamslice(...,'arrowsmode')` determines if direction arrows are present or not. `arrowmode` can be

- `arrows` — Draw direction arrows on the streamlines (default).
- `noarrows` — Do not draw direction arrows.

`streamslice(...,'method')` specifies the interpolation method to use. `method` can be

- `linear` — Linear interpolation (default)
- `cubic` — Cubic interpolation
- `nearest` — Nearest-neighbor interpolation

See `interp3` for more information on interpolation methods.

`streamslice(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

`h = streamslice(...)` returns a vector of handles to the line objects created.

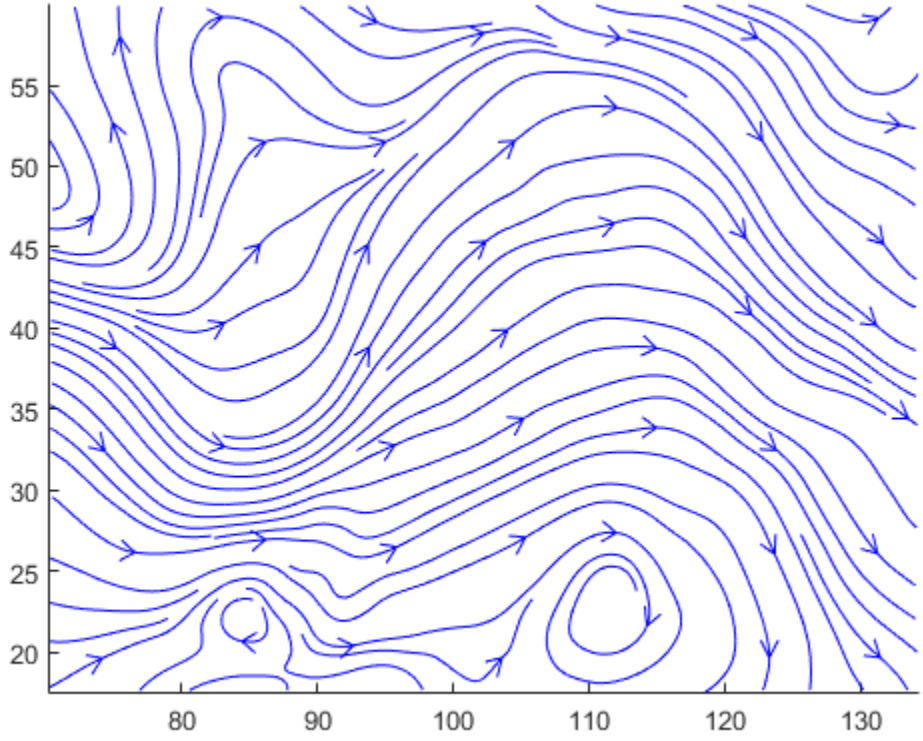
`[vertices arrowvertices] = streamslice(...)` returns two cell arrays of vertices for drawing the streamlines and the arrows. You can pass these values to any of the streamline drawing functions (`streamline`, `streamribbon`, `streamtube`).

Examples

Plot Streamlines in Slice Plane

Load the `wind` data set. Draw a stream slice at `z = 5`.

```
load wind

figure
streamslice(x,y,z,u,v,w,[],[],[5])
axis tight
```



More About

- Specifying Starting Points for Stream Plots

See Also

`contourslice` | `slice` | `streamline` | `volumebounds` | `meshgrid` | `interp3` | `interp2` | `streamribbon` | `streamtube`

Introduced before R2006a

streamtube

Create 3-D stream tube plot

Syntax

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
streamtube(axes_handle,...)
h = streamtube(...z)
```

Description

`streamtube(X,Y,Z,U,V,W,startx,starty,startz)` draws stream tubes from vector volume data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the streamlines at the center of the tubes. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The width of the tubes is proportional to the normalized divergence of the vector field.

`streamtube(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamtube(vertices,X,Y,Z,divergence)` assumes precomputed streamline vertices and divergence. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, and `divergence` are 3-D arrays.

`streamtube(vertices,divergence)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(divergence)`.

`streamtube(vertices,width)` specifies the width of the tubes in the cell array of vectors, `width`. The size of each corresponding element of `vertices` and `width` must be equal. `width` can also be a scalar, specifying a single value for the width of all stream tubes.

`streamtube(vertices)` selects the width automatically.

`streamtube(...,[scale n])` scales the width of the tubes by `scale`. The default is `scale = 1`. When the stream tubes are created, using start points or divergence, specifying `scale = 0` suppresses automatic scaling. `n` is the number of points along the circumference of the tube. The default is `n = 20`.

`streamtube(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

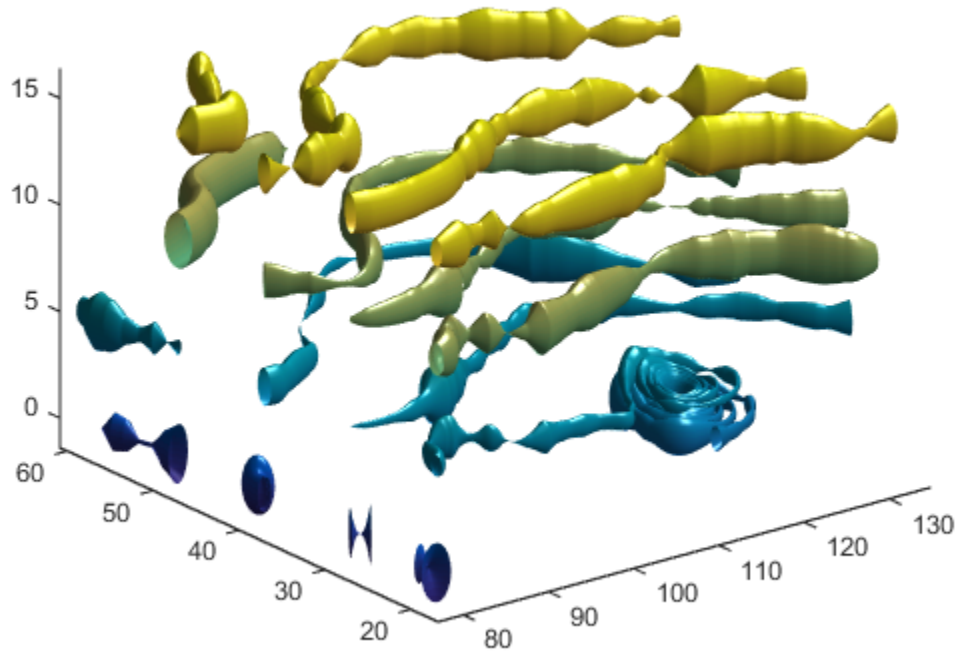
`h = streamtube(...z)` returns a vector of handles (one per start point) to `surface` objects used to draw the stream tubes.

Examples

This example uses stream tubes to indicate the flow in the `wind` data set. Inputs include the coordinates, vector field components, and starting location for the stream tubes.

```
figure
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
```

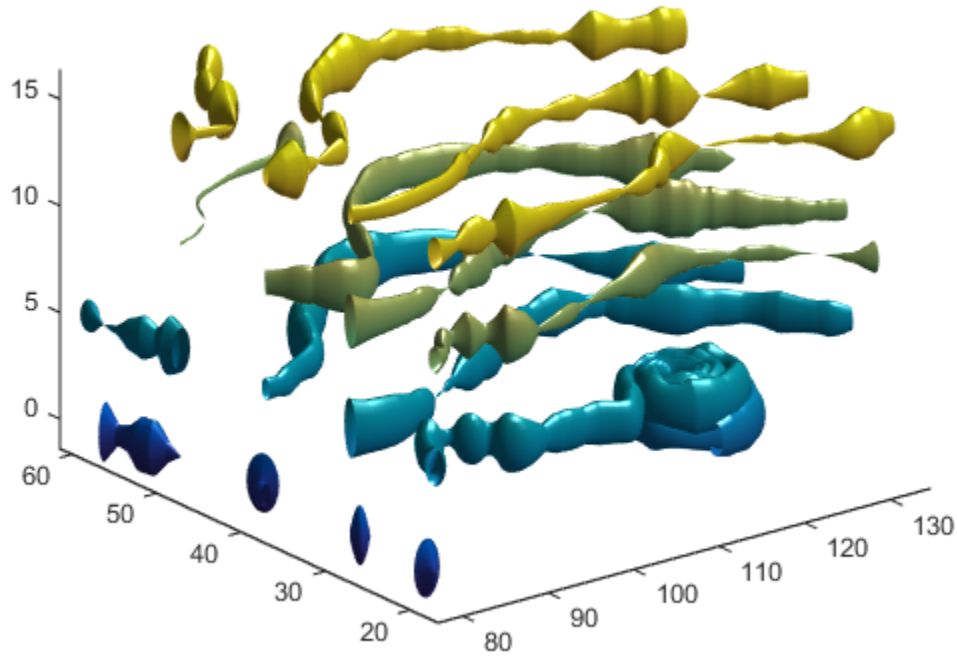
```
streamtube(x,y,z,u,v,w,sx,sy,sz);
% Define viewing and lighting
view(3)
axis tight
shading interp;
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`) and divergence (`divergence`).

```
figure
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
```

```
streamtube(verts,x,y,z,-div);
% Define viewing and lighting
view(3)
axis tight
shading interp
camlight
lighting gouraud
```



More About

- Displaying Divergence with Stream Tubes
- Specifying Starting Points for Stream Plots

See Also

`divergence` | `streamribbon` | `streamline` | `stream3` | `meshgrid` | `stream3`

Introduced before R2006a

strfind

Find one string within another

Syntax

```
k = strfind(str, pattern)
```

Description

`k = strfind(str, pattern)` searches str for occurrences of pattern. The output, k, indicates the starting index of each occurrence of pattern in str. If pattern is not found `strfind` returns an empty array, `[]`. The `strfind` function executes a case sensitive search.

- If str is a string, `strfind` returns a vector of type `double`.
- If str is a cell array, `strfind` returns a cell array of vectors of type `double`.

Examples

Find String Pattern

Define string `S` as follows:

```
S = 'Find the starting indices of the pattern string';
```

Find the pattern `in` in string `S`.

```
k = strfind(S, 'in')
```

```
k =

     2    15    19    45
```

There are four instances of the substring, `in`, found in `S`.

Find the pattern `In` in string `S`.

```
k = strfind(S, 'In')

k =

     []
```

Since `strfind` is case sensitive, the string, `In`, is not found in `S`, k is an empty array.

Find the blank spaces in string `S`.

```
k = strfind(S, ' ')

k =

     5     9    18    26    29    33    41
```

There are seven occurrences of blank spaces in the string `S`.

Find String in Cell Array of Strings

Define the cell array of strings, `cstr`, as follows:

```
cstr = {'How much wood would a woodchuck chuck';
        'if a woodchuck could chuck wood?'};
```

Find the pattern, `wood`, in cell array `cstr`.

```
idx = strfind(cstr, 'wood')

idx =

    [1x2 double]
    [1x2 double]
```

Examine the output cell array to find the instances of `wood`.

```
idx{:,:}

ans =

    10    23


ans =

     6    28
```

The pattern, `wood`, occurs at indices 10 and 23 in the first string and at indices 6 and 28 in the second string.

Input Arguments

`str` — Data to be searched

string | cell array of strings

Data to be searched, specified as a string or cell array of strings.

Data Types: `char` | `cell`

`pattern` — Search pattern

string

Search pattern, specified as a string.

Data Types: `char`

Output Arguments

`k` — Indices of occurrences of `pattern`

array

Indices of occurrences of pattern, returned as an array. If pattern is not found, then k is an empty array, `[]`.

- If str is a string, k is an array of doubles indicating the index of each occurrence of pattern.
- If str is a cell array of strings, k is a cell array with each element being an array of type `double` corresponding to the indices of each occurrence of pattern in the corresponding element of str.

More About

Tips

- The `strfind` function does not find a pattern of empty strings, `''`, within str.

See Also

`regexp` | `regexpi` | `regexprep` | `strcmp` | `strcmpi` | `strncmp` | `strncmpi` | `strrep` | `strsplit` | `strtok`

Introduced before R2006a

strings

String handling

Syntax

```
S = 'Any Characters'
S = [S1 S2 ...]
C = {S1 S2 ...}
S = strcat(S1, S2, ...)
S = char(S1, S2, ...)
S = char(X)
X = double(S)
```

Description

`S = 'Any Characters'` creates a character array, or string. The string is actually a vector that contains the numeric codes for the characters (codes 0 to 127 are ASCII). The length of `S` is the number of characters. A quotation within the string is indicated by two quotation marks.

`S = [S1 S2 ...]` concatenates character arrays `S1`, `S2`, etc. into a new character array, `S`.

`C = {S1 S2 ...}` creates a cell array of strings. Separate each row of the cell array with a semicolon (`;`).

`S = strcat(S1, S2, ...)` horizontally concatenates `S1`, `S2`, etc., which can be character arrays or cell arrays of strings. If the inputs are character arrays, `strcat` removes trailing white space. For more information, see the `strcat` reference page.

`S = char(S1, S2, ...)` vertically concatenates character arrays `S1`, `S2`, etc., padding each input string as needed so that each row contains the same number of characters.

`S = char(X)` converts an array that contains positive integers representing numeric codes into a MATLAB character array.

`X = double(S)` converts the string to its equivalent integer numeric codes.

Examples

Create a simple string that includes a single quote.

```
msg = 'You''re right!'

msg =
You're right!
```

Create the string `name` using two methods of concatenation.

```
name = ['Thomas' ' R. ' 'Lee']
name = strcat('Thomas',' R.',' Lee')
```

Create a character array of strings.

```
C = char('Hello','Goodbye','Yes','No')

C =
Hello
Goodbye
Yes
No
```

Create a cell array of strings.

```
S = {'Hello' 'Goodbye'; 'Yes' 'No'}

S =
    'Hello'    'Goodbye'
    'Yes'      'No'
```

More About

Tips

- To convert between character arrays and cell arrays of strings, use `char` and `cellstr`. Most string functions support both types.
- To determine whether `S` is a character array or cell array, call `ischar(S)` or `iscellstr(S)`.
- “Cell Arrays of Strings”

See Also

`char` | `isstrprop` | `ischar` | `cellstr` | `isletter` | `isspace` | `iscellstr` | `sprintf` | `sscanf` | `text` | `input`

strjoin

Join strings in cell array into single string

Syntax

```
str = strjoin(C)
str = strjoin(C,delimiter)
```

Description

`str = strjoin(C)` constructs the string, `str`, by linking each string in the cell array, C, with a single space.

`str = strjoin(C,delimiter)` constructs the string, `str`, by linking each string of C with the elements in delimiter.

Examples

Join List of Words with Whitespace

Join individual strings in a cell array of strings, `C`, with a single space.

```
C = {'one','two','three'};
str = strjoin(C)
```

```
str =

one two three
```

Join Cell Array of Strings

```
C = {'Newton','Gauss','Euclid','Lagrange'};
str = strjoin(C,', ')
```

```
str =
```

```
Newton, Gauss, Euclid, Lagrange
```

Join Strings with Multiple Different Delimiters

Specify multiple different delimiters in a cell array of strings. The `delimiter` cell array must have one fewer element than `C`.

```
C = {'one','two','three'};
str = strjoin(C,{' + ',' = '})
```

```
str =

one + two = three
```

Input Arguments

`C` — Input text

1-by-n cell array of strings

Input text, specified as a 1-by-n cell array of strings. Each element in the cell array must contain a single string in a single row.

Example: `{'The','rain','in','Spain'}`

Data Types: `cell`

`delimiter` — Delimiting characters

string | 1-by-n cell array of strings

Delimiting characters, specified as a single string or a 1-by-n cell array of strings.

- If `delimiter` is a single string, then `strjoin` forms `str` by inserting `delimiter` between each element of `C`. The `delimiter` input can include any of these escape sequences:

`\\`	Backslash
`\0`	Null
`\a`	Alarm
`\b`	Backspace
`\f`	Form feed

`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

- If `delimiter` is a cell array of strings, then it must contain one fewer element than `C`. Each element in the cell array must contain a single string in a single row. `strjoin` forms `str` by interleaving the elements of `delimiter` and `C`. All characters in `delimiter` are inserted as literal text, and escape sequences are not supported.

Example: `', '`

Example: `{',',' '}`

Data Types: `char` | `cell`

See Also

`cellstr` | `regexp` | `strcat` | `strsplit`

strjust

Justify character array

Syntax

```
T = strjust(S)
T = strjust(S, 'right')
T = strjust(S, 'left')
T = strjust(S, 'center')
```

Description

`T = strjust(S)` or `T = strjust(S, 'right')` returns a right-justified version of the character array `S`.

`T = strjust(S, 'left')` returns a left-justified version of `S`.

`T = strjust(S, 'center')` returns a center-justified version of `S`.

See Also

`deblank` | `strtrim`

Introduced before R2006a

strmatch

Find possible matches for string

Note: `strmatch` is not recommended. Use `strncmp` or `validatestring`, depending on your requirements, instead. `strncmp` returns a logical array indicating which array elements begin with the specified string, whereas `validatestring` returns a single string that represents the best match to the specified string. See Example 2, below.

To find an exact match for a string, use `strcmp`.

Syntax

```
x = strmatch(str, strarray)
x = strmatch(str, strarray, 'exact')
```

Description

`x = strmatch(str, strarray)` looks through the rows of the character array or cell array of strings `strarray` to find strings that begin with the text contained in `str`, and returns the matching row indices. If `strmatch` does not find `str` in `strarray`, `x` is an empty matrix (`[ ]`). Any trailing space characters in `str` or `strarray` are ignored when matching. `strmatch` is fastest when `strarray` is a character array.

`x = strmatch(str, strarray, 'exact')` compares `str` with each row of `strarray`, looking for an exact match of the entire strings. Any trailing space characters in `str` or `strarray` are ignored when matching.

Examples

Example 1

The statement

```
x = strmatch('max', char('max', 'minimax', 'maximum'))
```

returns `x = [1; 3]` since rows 1 and 3 begin with `'max'`. The statement

```
x = strmatch('max', char('max', 'minimax', 'maximum'),'exact')
```

returns `x = 1`, since only row 1 matches `'max'` exactly.

Example 2

This example shows how to replace use of the `strmatch` function with `validatestring` or `strncmp`.

To start with, use `strmatch` to return the index of those elements for which there is a match:

```
list = {'max', 'minimax', 'maximum', 'max'}
x = strmatch('max',list)
x =
     1
     3
     4
```

`validatestring` returns the string representing the best match. If multiple or no matches exist, this statement would return an error:

```
list = {'max', 'minimax', 'maximum', 'max'};
x = validatestring('max', list)
x =
   max
```

`strncmp` returns a logical array indicating which strings match the specified string:

```
list = {'max', 'minimax', 'maximum', 'max'};
x = strncmp('max', list, 3)
x =
     1     0     1     1
```

If you prefer that MATLAB return the numeric indices of `list`, use `find` as follows:

```
list = {'max', 'minimax', 'maximum', 'max'}
x = find(strncmp(list, 'max', 3))
```

If your input to `strmatch` is a character matrix, then first convert the matrix to a cell array using `cellstr`. Then, pass the output from `cellstr` to `strncmp` or `validatestring`

See Also

`strcmp` | `strcmpi` | `strncmp` | `strncmpi` | `strfind` | `regexp` | `regexpi` | `regexprep`

Introduced before R2006a

strncmp

Compare first `n` characters of strings (case sensitive)

Syntax

```
TF = strncmp(string,string,n)
TF = strncmp(string,cellstr,n)
TF = strncmp(cellstr,cellstr,n)
```

Description

`TF = strncmp(string,string,n)` compares the first `n` characters of two strings for equality. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.

`TF = strncmp(string,cellstr,n)` compares the first `n` characters of a string with the first `n` characters of each element of a cell array of strings. The function returns a logical array the same size as the `cellstr` input in which logical 1 represents equality. The order of the first two input arguments is not important.

`TF = strncmp(cellstr,cellstr,n)` compares each element of one cell array of strings with the same element of the other. `strncmp` attempts to match only the first `n` characters of these strings. The function returns a logical array the same size as either input array.

Input Arguments

`string`

Single character string or n-by-1 array of strings.

`cellstr`

Cell array of strings.

n

Maximum number of characters to compare. Must be a scalar, integer-valued double.

Output Arguments

TF

When both inputs are character arrays, `TF` is a scalar logical value. This value is logical 1 (`true`) if the size and content of both arrays are equal, and logical 0 (`false`) if they are not.

When either or both inputs are a cell array of strings, `TF` is an array of logical ones and zeros. This array is the same size as the input cell array(s), and contains logical 1 (`true`) for those elements of the input arrays that are a match, and logical 0 (`false`) for those elements that are not.

Examples

Before trying the `strncmp` function, use `strcmp` to perform a simple comparison of the two input strings. Because only the first 13 characters are the same, `strcmp` returns logical 0:

```
strcmp('Kansas City, KS', 'Kansas City, MO')
ans =
     0
```

Do the comparison again, but this time using `strncmp` and specifying the number of characters to compare:

```
chars2compare = length('Kansas City, KS') - 2
ans =
    13
strncmp('Kansas City, KS', 'Kansas City, MO', chars2compare)
ans =
     1
```

From a list of 10 MATLAB functions, find those that apply to using a camera:

```
function_list = {'calendar' 'case' 'camdolly' 'circshift' ...
                 'caxis' 'camtarget' 'cast' 'camorbit' ...
```

```
               'callib' 'cart2sph'};

strncmp(function_list, 'cam', 3)
ans =
    0    0    1    0    0    1    0    1    0    0

function_list{strncmp(function_list, 'cam', 3)}
ans =
   camdolly
ans =
   camtarget
ans =
   camorbit
```

Create two 5-by-10 string arrays `str1` and `str2` that are equal, except for the element at row 4, column 3. Using linear indexing, this is element 14:

```
str1 = ['AAAAAAAAAA'; 'BBBBBBBBBB'; 'CCCCCCCCCC'; ...
        'DDDDDDDDDD'; 'EEEEEEEEEE']
str1 =
    AAAAAAAAAA
    BBBBBBBBBB
    CCCCCCCCCC
    DDDDDDDDDD
    EEEEEEEEEE

str2 = str1;
str2(4,3) = '-'
str2 =
    AAAAAAAAAA
    BBBBBBBBBB
    CCCCCCCCCC
    DD-DDDDDDD
    EEEEEEEEEE
```

Because MATLAB compares the arrays in linear order (that is, column by column rather than row by row), `strncmp` finds only the first 13 elements to be the same:

```
str1   A B C D E A B C D E A B C D E
str2   A B C D E A B C D E A B C - E
                                 |
                              element 14

strncmp(str1, str2, 13)
ans =
     1
```

```
strncmp(str1, str2, 14)
ans =
     0
```

More About

Tips

- The `strncmp` function is intended for comparison of character data. When used to compare numeric data, it returns logical `0`.
- Use `strncmpi` for case-insensitive string comparisons.
- Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
- The value returned by `strncmp` is not the same as the C language convention.
- `strncmp` supports international character sets.

See Also

`strcmp` | `strncmpi` | `strcmpi` | `strfind` | `regexp` | `regexpi`

Introduced before R2006a

strncmpi

Compare first `n` characters of strings (case insensitive)

Syntax

```
TF = strncmpi(string,string,n)
TF = strncmpi(string,cellstr,n)
TF = strncmpi(cellstr,cellstr,n)
```

Description

`TF = strncmpi(string,string,n)` compares the first `n` characters of two strings for equality, ignoring any differences in letter case. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.

`TF = strncmpi(string,cellstr,n)` compares the first `n` characters of a string with the first `n` characters of each element of a cell array of strings, ignoring letter case. The function returns a logical array the same size as the `cellstr` input in which logical 1 represents equality. The order of the input arguments is not important.

`TF = strncmpi(cellstr,cellstr,n)` compares each element of one cell array of strings with the same element of the other, ignoring letter case. `strncmpi` attempts to match only the first `n` characters of these strings. The function returns a logical array the same size as either input array.

Input Arguments

`string`

A single character string or n-by-1 array of strings.

`cellstr`

A cell array of strings.

`n`

Maximum number of characters to compare. Must be a scalar, integer-valued double.

Output Arguments

TF

When both inputs are character arrays, `TF` is a scalar logical value. This value is logical 1 (`true`) if the size and content of both arrays are equal, and logical 0 (`false`) if they are not.

When either or both inputs are a cell array of strings, `TF` is an array of logical ones and zeros. This array is the same size as the input cell array(s), and contains logical 1 (`true`) for those elements of the input arrays that are a match, and logical 0 (`false`) for those elements that are not.

Examples

From a list of 10 MATLAB functions, find those that apply to using a camera. Do the comparison without sensitivity to letter case:

```
function_list = {'calendar' 'case' 'camdolly' 'circshift' ...
                 'caxis' 'Camtarget' 'cast' 'camorbit' ...
                 'callib' 'cart2sph'};

strncmpi(function_list, 'CAM', 3)
ans =
    0    0    1    0    0    1    0    1    0    0

function_list{strncmpi(function_list, 'CAM', 3)}
ans =
   camdolly
ans =
   Camtarget
ans =
   camorbit
```

More About

Tips

- The `strncmpi` function is intended for comparison of character data. When used to compare numeric data, it returns logical `0`.
- Use `strncmp` for case-sensitive string comparisons.

- Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
- The value returned by `strncmpi` is not the same as the C language convention.
- `strncmpi` supports international character sets.

See Also

`strcmpi` | `strncmp` | `strcmp` | `strfind` | `regexpi` | `regexp`

Introduced before R2006a

strread

Read formatted data from string

Note `strread` is not recommended. Use `textscan` instead.

Syntax

```
A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)
```

Description

`A = strread('str')` reads numeric data from input string `str` into a 1-by-`N` vector `A`, where `N` equals the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See “Example 1” on page 1-7811 below.

`[A, B, ...] = strread('str')` reads numeric data from the string input `str` into scalar output variables `A`, `B`, and so on. The number of output variables must equal the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See “Example 2” on page 1-7811 below.

`[A, B, ...] = strread('str', 'format')` reads data from `str` into variables `A`, `B`, and so on using the specified `format`. The number of output variables `A`, `B`, etc. must be equal to the number of format specifiers (e.g., `%s` or `%d`) in the `format` argument. You can read all of the data in `str` to a single output variable as long as you use only one format specifier in the command. See “Example 4” on page 1-7811 and “Example 5” on page 1-7812 below.

The table Formats for strread lists the valid format specifiers. More information on using formats is available under “Formats” on page 1-7814 in the “Tips” on page 1-7814 section below.

`[A, B, ...] = strread('str', 'format', N)` reads data from `str` reusing the `format` string `N` times, where `N` is an integer greater than zero. If `N` is -1, `strread` reads the entire string. When `str` contains only numeric data, you can set `format` to the empty string (`''`). See "Example 3" on page 1-7811 below.

`[A, B, ...] = strread('str', 'format', N, param, value, ...)` customizes `strread` using `param/value` pairs, as listed in the table Parameters and Values for strread below. When `str` contains only numeric data, you can set `format` to the empty string (`''`). The `N` argument is optional and may be omitted entirely. See "Example 7" on page 1-7813 below.

Formats for strread

Format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a string that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space separated string.	Cell array of strings
`%q`	Read a double quoted string, ignoring the quotes.	Cell array of strings
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings
`%[^...]`	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
`%*...`	Ignore the characters following `*`. See "Example 8" on page 1-7813 below.	No output

Format	Action	Output
`%w...`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

Parameters and Values for strread

param	value		Action
`whitespace`	Any from the list below:		Treats vector of characters, `*`, as white space. Default is `\b \r\n\t`.
	`\b`	Backspace	
	`\n`	New line	
	`\r`	Carriage return	
	`\t`	Horizontal tab	
	`\\`	Backslash	
	`%%`	Percent sign	
	`''`	Single quotation mark	
`delimiter`	Delimiter character		Specifies delimiter character. Default is one or more whitespace characters.
`expchars`	Exponent characters		Default is `eEdD`.
`bufsize`	Positive integer		Specifies the maximum string length, in bytes. Default is `4095`.
`commentstyle`	`matlab`		Ignores characters after `%`.
`commentstyle`	`shell`		Ignores characters after `#`.
`commentstyle`	`c`		Ignores characters between `/*` and `*/`.
`commentstyle`	`c++`		Ignores characters after `//`.
`emptyvalue`	Value to return for empty numeric fields in delimited files		Default is NaN.

Examples

Example 1

Read numeric data into a 1-by-5 vector:

```
a = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100    8.2400    3.5700    6.2400    9.2700
```

Example 2

Read numeric data into separate scalar variables:

```
[a b c d e] = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100
b =
    8.2400
c =
    3.5700
d =
    6.2400
e =
    9.2700
```

Example 3

Read the only first three numbers in the string, also formatting as floating point:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%4.2f', 3)

a =
    0.4100
    8.2400
    3.5700
```

Example 4

Truncate the data to one decimal digit by specifying format `%3.1f`. The second specifier, `%*1d`, tells `strread` not to read in the remaining decimal digit:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%3.1f %*1d')

a =
    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
```

Example 5

Read six numbers into two variables, reusing the format specifiers:

```
[a b] = strread('0.41 8.24 3.57 6.24 9.27 3.29', '%f %f')

a =
    0.4100
    3.5700
    9.2700
b =
    8.2400
    6.2400
    3.2900
```

Example 6

Read string and numeric data to two output variables. Ignore commas in the input string:

```
str = 'Section 4, Page 7, Line 26';

[name value] = strread(str, '%s %d,')
name =
    'Section'
    'Page'
    'Line'
value =
     4
     7
    26
```

Example 7

Read the string used in the last example, but this time delimiting with commas instead of spaces:

```
str = 'Section 4, Page 7, Line 26';

[a b c] = strread(str, '%s %s %s', 'delimiter', ',')
a =
    'Section 4'
b =
    'Page 7'
c =
    'Line 26'
```

Example 8

Read selected portions of the input string:

```
str = '<table border=5 width="100%" cellspacing=0>';

[border width space] = strread(str, ...
   '%*s%*s %c %*s "%4s" %*s %c', 'delimiter', '= ')
border =
     5
width =
    '100%'
space =
     0
```

Example 9

Read the string into two vectors, restricting the `Answer` values to `T` and `F`. Also note that two delimiters (comma and space) are used here:

```
str = 'Answer_1: T, Answer_2: F, Answer_3: F';

[a b] = strread(str, '%s %[TF]', 'delimiter', ', ')
a =
    'Answer_1:'
    'Answer_2:'
    'Answer_3:'
b =
```

```
    'T'
    'F'
    'F'
```

More About

Tips

If you terminate the input string with a newline character (`\n`), `strread` returns arrays of equal size by padding arrays of lesser size with the `emptyvalue` character:

```
[A,B,C] = strread(sprintf('5,7,1,9\n'),'%d%d%d', ...
              'delimiter', ',', 'emptyvalue',NaN)
A =
     5
     9
B =
     7
   NaN
C =
     1
   NaN
```

If you remove the `\n` from the input string of this example, array `A` continues to be a 2-by-1 array, but `B` and `C` are now 1-by-1.

Delimiters

If your data uses a character other than a space as a delimiter, you must use the `strread` parameter `'delimiter'` to specify the delimiter. For example, if the string `str` used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer] = strread(str,'%s %s %f ...
    %d %s','delimiter',';')
```

Formats

The `format` string determines the number and types of return arguments. The number of return arguments must match the number of conversion specifiers in the `format` string.

The `strread` function continues reading `str` until the entire string is read. If there are fewer format specifiers than there are entities in `str`, `strread` reapplies the format specifiers, starting over at the beginning. See “Example 5” on page 1-7812 below.

The `format` string supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine. White-space characters in the `format` string are ignored.

Preserving White-Space

If you want to preserve leading and trailing spaces in a string, use the `whitespace` parameter as shown here:

```
str = '   An  example      of preserving     spaces      ';

strread(str, '%s', 'whitespace', '')
ans =
    '   An  example      of preserving     spaces      '
```

See Also

`textscan` | `sscanf`

Introduced before R2006a

strrep

Find and replace substring

Syntax

```
modifiedStr = strrep(origStr, oldSubstr, newSubstr)
```

Description

`modifiedStr = strrep(origStr, oldSubstr, newSubstr)` replaces all occurrences of the string `oldSubstr` within string `origStr` with the string `newSubstr`.

Examples

Replace text in a character array:

```
claim = 'This is a good example.';
new_claim = strrep(claim, 'good', 'great')
```

MATLAB returns:

```
new_claim =
This is a great example.
```

Replace text in a cell array:

```
c_files = {'c:\cookies.m'; ...
           'c:\candy.m';   ...
           'c:\calories.m'};
d_files = strrep(c_files, 'c:', 'd:')
```

MATLAB returns:

```
d_files =
    'd:\cookies.m'
    'd:\candy.m'
```

```
    'd:\calories.m'
```

Replace text in a cell array with values in a second cell array:

```
missing_info = {'Start: __'; ...
                'End: __'};

dates = {'01/01/2001'; ...
         '12/12/2002'};

complete = strrep(missing_info, '__', dates)
```

MATLAB returns:

```
complete =
    'Start: 01/01/2001'
    'End: 12/12/2002'
```

Compare the use of `strrep` and `regexprep` to replace a string with a repeated pattern:

```
repeats = 'abc 2 def 22 ghi 222 jkl 2222';
indices = strfind(repeats, '22')

using_strrep = strrep(repeats, '22', '*')
using_regexprep = regexprep(repeats, '22', '*')
```

MATLAB returns:

```
indices =
    11    18    19    26    27    28

using_strrep =
abc 2 def * ghi ** jkl ***

using_regexprep =
abc 2 def * ghi *2 jkl **
```

More About

Tips

- `strrep` accepts input combinations of single strings, strings in scalar cells, and same-sized cell arrays of strings. If any inputs are cell arrays, `strrep` returns a cell array.

- The `strrep` function does not find empty strings for replacement. That is, when *`origStr`* and *`oldSubstr`* both contain the empty string (`''`), `strrep` does not replace `''` with the contents of *`newSubstr`*.
- Before replacing strings, `strrep` finds all instances of *`oldSubstr`* in *`origStr`*, like the `strfind` function. For overlapping patterns, `strrep` performs multiple replacements. See the final example in the Examples section.

See Also

`strfind` | `regexprep`

Introduced before R2006a

strsplit

Split string at specified delimiter

Syntax

```
C = strsplit(str)
C = strsplit(str,delimiter)
C = strsplit(str,delimiter,Name,Value)

[C,matches] = strsplit( ___ )
```

Description

`C = strsplit(str)` splits the string, `str`, at whitespace into the cell array of strings, `C`. A whitespace character is equivalent to any sequence in the set `{' ','\f','\n','\r','\t','\v'}`.

`C = strsplit(str,delimiter)` splits `str` at the delimiters specified by `delimiter`.

`C = strsplit(str,delimiter,Name,Value)` specifies additional delimiter options using one or more name-value pair arguments.

`[C,matches] = strsplit( ___ )` additionally returns a cell array of strings, `matches`, using any of the input arguments in the previous syntaxes. `matches` contains all occurrences of delimiters upon which `strsplit` splits `str`.

Examples

Split String on Whitespace

```
str = 'The rain in Spain.';
C = strsplit(str)
```

```
C =

    'The'    'rain'    'in'    'Spain.'
```

`C` is a cell array containing 4 strings.

Split String of Values on Specific Delimiter

Split a string of comma-separated values.

```
data = '1.21, 1.985, 1.955, 2.015, 1.885';
C = strsplit(data,', ')
```

```
C =

    '1.21'    '1.985'    '1.955'    '2.015'    '1.885'
```

Split a string of values, `data`, which contains the units `m/s` with an arbitrary number of white-space on either side of the text. The regular expression, `\s*`, matches any white-space character appearing zero or more times.

```
data = '1.21m/s1.985m/s 1.955 m/s2.015 m/s 1.885m/s';
[C,matches] = strsplit(data,'\s*m/s\s*',...
    'DelimiterType','RegularExpression')
```

```
C =

    '1.21'    '1.985'    '1.955'    '2.015'    '1.885'    ''


matches =

    'm/s'    'm/s '    ' m/s'    ' m/s '    'm/s'
```

In this case, the last string in `C` is empty. This empty string is the string that follows the last matched delimiter.

Split Path String on File Separator

```
myPath = 'C:\work\matlab';
C = strsplit(myPath,'\')
```

```
C =

    'C:'    'work'    'matlab'
```

Split Text String with Multiple Delimiters

Split a string on `' '` and `'ain'`, treating multiple delimiters as one. Specify multiple delimiters in a cell array of strings.

```
str = 'The rain in Spain stays mainly in the plain.';
[C,matches] = strsplit(str,{' ','ain'},'CollapseDelimiters',true)
```

```
C =

    'The'    'r'    'in'    'Sp'    'stays'    'm'    'ly'    'in'    'the'    'pl'


matches =

    ' '    'ain '    ' '    'ain '    ' '    'ain'    ' '    ' '    ' '    'ain'
```

Split the same string on whitespace and on `'ain'`, using regular expressions and treating multiple delimiters separately.

```
[C,matches] = strsplit(str,{'\s','ain'},'CollapseDelimiters',...
    false, 'DelimiterType','RegularExpression')
```

```
C =

    'The'    'r'    ''    'in'    'Sp'    ''    'stays'    'm'    'ly'    'in'    'the

matches =

    ' '    'ain'    ' '    ' '    'ain'    ' '    ' '    'ain'    ' '    ' '    ' '
```

In this case, `strsplit` treats the two delimiters separately, so empty strings appear in output `C` between the consecutively matched delimiters.

Split Text with Multiple, Overlapping Delimiters

Split text on the strings `', '` and `', and '`.

```
str = 'bacon, lettuce, and tomato';
[C,matches] = strsplit(str,{', ',', and '})
```

```
C =

    'bacon'    'lettuce'    'and tomato'


matches =

    ', '    ', '
```

Because the command lists `', '` first and `', and '` contains `', '`, the `strsplit` function splits `str` on the first delimiter and never proceeds to the second delimiter.

If you reverse the order of delimiters, `', and '` takes priority.

```
str = 'bacon, lettuce, and tomato';
[C,matches] = strsplit(str,{', and ',', '})
```

```
C =

    'bacon'    'lettuce'    'tomato'


matches =

    ', '    ', and '
```

Input Arguments

`str` — Input text

string

Input text, specified as a string.

Data Types: `char`

`delimiter` — Delimiting characters

string | 1-by-n cell array of strings

Delimiting characters, specified as a single string or a 1-by-n cell array of strings. Strings specified in delimiter do not appear in the string fragments of the output C.

Specify multiple delimiters in a cell array of strings. Each element of the cell array must contain a single string in a single row. The `strsplit` function splits str on the elements of delimiter in the order in which they appear in the cell array.

`delimiter` can include the following escape sequences:

`\\`	Backslash
`\0`	Null
`\a`	Alarm

`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

Example: `','`

Example: `{'-',','}`

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DelimiterType','RegularExpression'` instructs `strsplit` to treat `delimiter` as a regular expression.

`'CollapseDelimiters'` — Multiple delimiter handling

`1 (true)` (default) | `0 (false)`

Multiple delimiter handling, specified as the comma-separated pair consisting of `'CollapseDelimiters'` and either `true` or `false`. If `true`, then consecutive delimiters in `str` are treated as one. If `false`, then consecutive delimiters are treated as separate delimiters, resulting in empty string `''` elements between matched delimiters.

Example: `'CollapseDelimiters',true`

`'DelimiterType'` — Delimiter type

`'Simple'` (default) | `'RegularExpression'`

Delimiter type, specified as the comma-separated pair consisting of `'DelimiterType'` and one of the following strings.

`'Simple'`	Except for escape sequences, `strsplit` treats `delimiter` as a literal string.

`'RegularExpression'`	`strsplit` treats `delimiter` as a regular expression.

In both cases, `delimiter` can include escape sequences.

Output Arguments

`C` — Parts of original string

cell array of strings

Parts of the original string, returned as a cell array of strings. C always contains one more element than matches contains. Consequently, if the original string, `str`, ends with a delimiter, then the last cell in `C` contains an empty string.

`matches` — Identified delimiters

cell array of strings

Identified delimiters, returned as a cell array of strings. matches always contains one fewer element than output C contains.

More About

- "Regular Expressions"

See Also

`regexp` | `strfind` | `strjoin`

strtok

Selected parts of string

Syntax

```
token = strtok(str)
token = strtok(str, delimiter)
[token, remain] = strtok(str, ...)
```

Description

`token = strtok(str)` parses input string `str` from left to right, returning part or all of that string in `token`. Using the white-space character as a delimiter, the `token` output begins at the start of `str`, skipping any delimiters that might appear at the start, and includes all characters up to either the next delimiter or the end of the string. White-space characters include space (ASCII 32), tab (ASCII 9), and carriage return (ASCII 13).

The `str` argument can be a string of characters enclosed in single quotation marks, a cell array of strings each enclosed in single quotation marks, or a variable representing either of the two. If `str` is a cell array of `N` strings, then `token` is a cell array of `N` tokens, with `token{1}` derived from `str{1}`, `token{2}` from `str{2}`, and so on.

`token = strtok(str, delimiter)` is the same as the above syntax except that you specify the delimiting character(s) yourself using the `delimiter` character vector input. White-space characters are not considered to be delimiters when using this syntax unless you include them in the `delimiter` argument. If the `delimiter` input specifies more than one character, MATLAB treats each character as a separate delimiter; it does not treat the multiple characters as a delimiting string. The number and order of characters in the `delimiter` argument is unimportant. Do not use escape sequences as delimiters. For example, use `char(9)` rather than `'\t'` for tab.

`[token, remain] = strtok(str, ...)` returns in `remain` that part of `str`, if any, that follows `token`. The delimiter is included in `remain`. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned in `token`, and `remain` is an empty string (`''`). If `str` is a cell array of strings, `token` is a cell array of tokens and `remain` is a cell array of string remainders.

Examples

Example 1

This example uses the default white-space delimiter. Note that space characters at the start of the string are not included in the `token` output, but the space character that follows `token` is included in `remain`:

```
s = '  This is a simple example.';
[token, remain] = strtok(s)

token =
This
remain =
 is a simple example.
```

Example 2

Take a string of HTML code and break it down into segments delimited by the < and > characters. Write a `while` loop to parse the string and print each segment:

```
s = sprintf('%s%s%s%s', ...
'<ul class=continued><li class=continued>', ...
'<pre><a name="13474"></a>token = strtok', ...
'(''str'', delimiter)<a name="13475"></a>', ...
'token = strtok(''str'')');

remain = s;

while true
   [str, remain] = strtok(remain, '<>');
   if isempty(str),  break;  end
   disp(sprintf('%s', str))
end
```

Here is the output:

```
ul class=continued
li class=continued
pre
a name="13474"
/a
token = strtok('str', delimiter)
```

```
a name="13475"
/a
token = strtok('str')
```

Example 3

Using `strtok` on a cell array of strings returns a cell array of strings in `token` and a character array in `remain`:

```
s = {'all in good time'; ...
     'my dog has fleas'; ...
     'leave no stone unturned'};

remain = s;

for k = 1:4
   [token, remain] = strtok(remain);
   token
end
```

Here is the output:

```
token =
    'all'
    'my'
    'leave'
token =
    'in'
    'dog'
    'no'
token =
    'good'
    'has'
    'stone'
token =
    'time'
    'fleas'
    'unturned'
```

See Also

`strfind` | `strcmp` | `strncmp` | `textscan` | `strsplit`

Introduced before R2006a

strtrim

Remove leading and trailing white space from string

Syntax

```
S = strtrim(str)
C = strtrim(cstr)
```

Description

`S = strtrim(str)` returns a copy of string `str` with all leading and trailing white-space characters removed. A white-space character is one for which the `isspace` function returns logical `1` (`true`).

`C = strtrim(cstr)` returns a copy of the cell array of strings `cstr` with all leading and trailing white-space characters removed from each string in the cell array.

Examples

Remove the leading white-space characters (spaces and tabs) from `str`:

```
str = sprintf('  \t   Remove    leading white-space')
str =
        Remove    leading white-space

str = strtrim(str)
str =
Remove    leading white-space
```

Remove leading and trailing white-space from the cell array of strings:

```
cstr = {'     Trim leading white-space';
        'Trim trailing white-space     '};

cstr = strtrim(cstr)
cstr =
    'Trim leading white-space'
```

```
    'Trim trailing white-space'
```

See Also

isspace | cellstr | deblank | strjust

Introduced before R2006a

struct

Create structure array

Syntax

```
s = struct
s = struct(field,value)
s = struct(field1,value1,...,fieldN,valueN)
s = struct([])

s = struct(obj)
```

Description

`s = struct` creates a scalar (1-by-1) structure with no fields.

`s = struct(field,value)` creates a structure array with the specified field and values. The `value` input argument can be any data type, such as a numeric, logical, character, or cell array.

- If `value` is *not* a cell array, then `s` is a scalar structure, where `s.(field) = value`.
- If `value` is a cell array, then `s` is a structure array with the same dimensions as `value`. Each element of `s` contains the corresponding element of `value`. For example, `s = struct('f',{'a','b'})` returns `s(1).f = 'a'` and `s(2).f = 'b'`.
- If `value` is an empty cell array `{}`, then `s` is an empty (0-by-0) structure.

`s = struct(field1,value1,...,fieldN,valueN)` creates a structure array with multiple fields. Any nonscalar cell arrays in the set `value1,...,valueN` must have the same dimensions.

- If none of the `value` inputs is a cell array, or if all `value` inputs that are cell arrays are scalars, then `s` is a scalar structure.
- If any of the `value` inputs is a nonscalar cell array, then `s` has the same dimensions as the nonscalar cell array. For any `value` that is a scalar cell array or an array of any other data type, `struct` inserts the contents of `value` in the relevant field for all elements of `s`.

- If any `value` input is an empty cell array, `{}`, then output `s` is an empty (0-by-0) structure. To specify an empty field and keep the values of the other fields, use `[]` as a `value` input instead.

`s = struct([])` creates an empty (0-by-0) structure with no fields.

`s = struct(obj)` creates a scalar structure with field names and values that correspond to properties of `obj`. The `struct` function does not convert `obj`, but rather creates `s` as a new structure. This structure does not retain the class information, so private, protected, and hidden properties become public fields in `s`. The `struct` function issues a warning when you use this syntax.

Examples

Structure with One Field

Create a nonscalar structure with one field.

```
field = 'f';
value = {'some text';
         [10, 20, 30];
         magic(5)};
s = struct(field,value)
```

```
s =
3x1 struct array with fields:
    f
```

View the contents of each element.

```
s.f
```

```
ans =
some text

ans =
    10    20    30

ans =
    17    24     1     8    15
    23     5     7    14    16
```

```
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

When you access a field of a nonscalar structure, such as `s.f`, MATLAB returns a comma-separated list. In this case, `s.f` is equivalent to `s(1).f, s(2).f, s(3).f`.

Structure with Multiple Fields

Create a nonscalar structure with several fields.

```
field1 = 'f1';  value1 = zeros(1,10);
field2 = 'f2';  value2 = {'a', 'b'};
field3 = 'f3';  value3 = {pi, pi.^2};
field4 = 'f4';  value4 = {'fourth'};

s = struct(field1,value1,field2,value2,field3,value3,field4,value4)

s =
1x2 struct array with fields:
    f1
    f2
    f3
    f4
```

The cell arrays for `value2` and `value3` are 1-by-2, so `s` is also 1-by-2. Because `value1` is a numeric array and not a cell array, both `s(1).f1` and `s(2).f1` have the same contents. Similarly, because the cell array for `value4` has a single element, `s(1).f4` and `s(2).f4` have the same contents.

```
s(1)

ans =
    f1: [0 0 0 0 0 0 0 0 0 0]
    f2: 'a'
    f3: 3.1416
    f4: 'fourth'

s(2)

ans =
    f1: [0 0 0 0 0 0 0 0 0 0]
    f2: 'b'
    f3: 9.8696
```

```
    f4: 'fourth'
```

Structure with Empty Field

Create a structure with an empty field. Use `[]` to specify the value of the empty field.

```
s = struct('f1','a','f2',[])

s =

    f1: 'a'
    f2: []
```

Fields That Contain Cell Arrays

Create a structure with a field that contains a cell array.

```
field = 'mycell';
value = {{'a','b','c'}};
s = struct(field,value)

s =
    mycell: {'a'  'b'  'c'}
```

Empty Structure

Create an empty structure with several fields.

```
s = struct('a',{},'b',{},'c',{})

s =
0x0 struct array with fields:
    a
    b
    c
```

Assign a value to a field in an empty structure.

```
s(1).a = 'a'

s =

    a: 'a'
    b: []
```

```
    c: []
```

Nested Structure

Create a nested structure. `a` is a structure with a field which contains another structure.

```
a.b = struct('c',{},'d',{})
```

```
a =
    b: [0x0 struct]
```

View the names of the fields of `a.b`.

```
fieldnames(a.b)
```

```
ans =
    'c'
    'd'
```

- "Create a Structure Array"
- "Access Data in a Structure Array"
- "Generate Field Names from Variables"

Input Arguments

`field` — Field name

string

Field name, specified as a string. Valid field names begin with a letter, and can contain letters, digits, and underscores. The maximum length of a field name is the value that the `namelengthmax` function returns.

`value` — Values within structure field

cell array | scalar | vector | multidimensional array

Values within a structure field, specified as a cell array or as a scalar, vector, or multidimensional array of any other data type.

If none of the `value` inputs is a cell array, or if all `value` inputs that are cell arrays are scalars, then output `s` is a scalar structure. Otherwise, `value` inputs that are nonscalar cell arrays must have the same dimensions, and output `s` also has those dimensions. For

any `value` that is a scalar cell array or an array of any other data type, `struct` inserts the contents of `value` in the relevant field for all elements of `s`.

If any `value` input is an empty cell array, `{}`, then output `s` is an empty structure array. To specify an empty field and keep the values of the other fields, use `[]` as a `value` input instead.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

`obj` — Object

object of any nonfundamental class

Object. `struct` copies the properties of `obj` to the fields of a new scalar structure. `obj` cannot be an object of a fundamental data type, such as `double` or `char`. See "Fundamental MATLAB Classes" for the list of fundamental data types.

See Also

`cell2struct` | `fieldnames` | `isfield` | `isstruct` | `orderfields` | `rmfield` | `struct2cell` | `struct2table` | `substruct` | `table` | `table2struct`

Introduced before R2006a

struct2cell

Convert structure to cell array

Syntax

```
c = struct2cell(s)
```

Description

`c = struct2cell(s)` converts the `m`-by-`n` structure `s` (with `p` fields) into a `p`-by-`m`-by-`n` cell array `c`.

If structure `s` is multidimensional, cell array `c` has size `[p size(s)]`.

Examples

The commands

```
clear s, s.category = 'tree';
s.height = 37.4; s.name = 'birch';
```

create the structure

```
s =
    category: 'tree'
      height: 37.4000
        name: 'birch'
```

Converting the structure to a cell array,

```
c = struct2cell(s)

c =
    'tree'
    [37.4000]
    'birch'
```

More About

Tips

- Use `fieldnames` to obtain structure field names in the same order as `struct2cell` returns structure values.
- dynamic field names

See Also

`cell2struct` | `struct2table` | `table2cell` | `cell` | `iscell` | `struct` | `isstruct` | `fieldnames`

Introduced before R2006a

struct2table

Convert structure array to table

Syntax

```
T = struct2table(S)
T = struct2table(S,Name,Value)
```

Description

`T = struct2table(S)` converts the structure array, `S`, to a table, `T`. Each field of `S` becomes a variable in `T`.

`T = struct2table(S,Name,Value)` creates a table from a structure array, `S`, with additional options specified by one or more Name,Value pair arguments.

For example, you can specify row names to include in the table.

Examples

Convert Scalar Structure to Table

Convert a scalar structure to a table using the default options.

Create a structure array, `S`.

```
S.Name = {'CLARK';'BROWN';'MARTIN'};
S.Gender = {'M';'F';'M'};
S.SystolicBP = [124;122;130];
S.DiastolicBP = [93;80;92];

S
```

```
S =

           Name: {3x1 cell}
         Gender: {3x1 cell}
```

```
     SystolicBP: [3x1 double]
    DiastolicBP: [3x1 double]
```

The scalar structure, `S`, has four fields, each with three rows.

Convert the structure array to a table.

```
T = struct2table(S)
```

```
T =

      Name       Gender    SystolicBP    DiastolicBP
    _________    ______    __________    ___________

    'CLARK'      'M'       124           93
    'BROWN'      'F'       122           80
    'MARTIN'     'M'       130           92
```

The structure field names in `S` become the variable names in the output table. The size of `T` is 3-by-4.

Change `Name` from a variable to row names by modifying the table property, `T.Properties.RowNames`, and then deleting the variable `Name`.

```
T.Properties.RowNames = T.Name;
T.Name = [];
```

```
T
```

```
T =

                Gender    SystolicBP    DiastolicBP
                ______    __________    ___________

    CLARK       'M'       124           93
    BROWN       'F'       122           80
    MARTIN      'M'       130           92
```

Convert Nonscalar Structure Array to Table

Create a nonscalar structure array, `S`.

```
S(1,1).Name = 'CLARK';
S(1,1).Gender = 'M';
S(1,1).SystolicBP = 124;
```

```
S(1,1).DiastolicBP = 93;

S(2,1).Name = 'BROWN';
S(2,1).Gender = 'F';
S(2,1).SystolicBP = 122;
S(2,1).DiastolicBP = 80;

S(3,1).Name = 'MARTIN';
S(3,1).Gender = 'M';
S(3,1).SystolicBP = 130;
S(3,1).DiastolicBP = 92;

S
```

```
S =

3x1 struct array with fields:

    Name
    Gender
    SystolicBP
    DiastolicBP
```

`S` is a 3-by-1 structure array with four fields.

Convert the structure array to a table.

```
T = struct2table(S)
```

```
T =

      Name       Gender    SystolicBP    DiastolicBP
    _________    _______    __________    ___________

    'CLARK'      'M'        124           93
    'BROWN'      'F'        122           80
    'MARTIN'     'M'        130           92
```

The structure field names in `S` become the variable names in the output table. The size of `T` is 3-by-4.

Treat Scalar Stucture As Array

Use `'AsArray',true` to create a table from a scalar structure whose fields have different numbers of rows.

Create a scalar structure, `S`, with fields `name`, `billing`, and `test`.

```
S.name = 'John Doe';
S.billing = 127.00;
S.test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
S
```

```
S =

       name: 'John Doe'
    billing: 127
       test: [3x3 double]
```

The fields have a different number of rows. Therefore, you cannot use `struct2table(S)`, which uses `'AsArray',false` by default.

Treat the scalar structure as an array and convert it to a table.

```
T = struct2table(S,'AsArray',true)
```

```
T =

       name       billing        test
    __________    _______    _____________

    'John Doe'    127        [3x3 double]
```

`T` contains one row.

- "Access Data in a Table"

Input Arguments

S — Structure array

structure array

Structure array, specified as a scalar structure array.

- If `S` is a scalar structure with `n` fields, all of which have `m` rows, then `T` is an `m`-by-`n` table.
- If `S` is a nonscalar `m`-by-`1` structure array with `n` fields, then `T` is an `m`-by-`n` table.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `RowNames','{'row1','row2','row3'}` uses the row names, `row1`, `row2`, and `row3` for the table, `T`.

`'RowNames'` — Row names for `T`

`{}` (default) | cell array of nonempty, distinct strings

Row names for `T`, specified as the comma-separated pair consisting of `'RowNames'` and a cell array of nonempty, distinct strings.

`'AsArray'` — Indicator for how to treat scalar structure

`false` (default) | `true` | `0` | `1`

Indicator for how to treat scalar structure, specified as the comma-separated pair consisting of `'AsArray'` and either `false`, `true`, `0`, or `1`.

`true`	`struct2table` converts `S` to a table with one row and `n` variables. The variables can be different sizes.
`false`	`struct2table` converts a scalar structure array with `n` fields into an `m`-by-`n` table. Each field must have `m` rows. This is the default behavior

Output Arguments

`T` — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

See Also

`array2table` | `cell2table` | `table` | `table2struct`

structfun

Apply function to each field of scalar structure

Syntax

```
[A1,...,An] = structfun(func,S)
[A1,...,An] = structfun(func,S,Name,Value)
```

Description

`[A1,...,An] = structfun(func,S)` applies the function specified by function handle `func` to each field of scalar structure `S`. Output arrays `A1,...,An`, where `n` is the number of outputs from function `func`, contain the outputs from the function calls.

`[A1,...,An] = structfun(func,S,Name,Value)` calls function `func` with additional options specified by one or more Name,Value pair arguments. Possible values for `Name` are `'UniformOutput'` or `'ErrorHandler'`.

Input Arguments

`func`

Handle to a function that accepts a single input argument and returns `n` output arguments.

If function `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

`S`

Scalar structure.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'UniformOutput'`

Logical value, as follows:

`true (1)`	Indicates that for all inputs, each output from function `func` is a cell array or a scalar value that is always of the same type and size. The `structfun` function combines the outputs in arrays `A1,...,An`, where `n` is the number of function outputs. Each output array is of the same type as the individual function outputs.
`false (0)`	Requests that the `structfun` function combine the outputs into scalar structures `A1,...,An`, with the same fields as input structure `S`. The outputs of function `func` can be of any size or type.

Default: `true`

`'ErrorHandler'`

Handle to a function that catches any errors that occur when MATLAB attempts to execute function `func`. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Linear index corresponding to the element of the input cell array at the time of the error.

- The set of input arguments to function `func` at the time of the error.

Output Arguments

`A1,...,An`

Arrays that collect the `n` outputs from function `func`.

If `UniformOutput` is `true` (the default):

- The individual outputs from function `func` must be scalar values (numeric, logical, character, or structure) or cell arrays.
- The class of a particular output argument must be the same for each input. The class of the corresponding output array is the same as the class of the individual outputs from function `func`.
- Each array `A` is a column vector whose length equals the number of fields in `S`. The `structfun` function applies function `func` to the fields of `S` in the same order as that returned by the `fieldnames` function.

If `UniformOutput` is `false`, each array `A` is a scalar structure with the same fields as input `S`.

Examples

Create a scalar structure, and count the number of characters in each field.

```
s.f1 = 'Sunday';
s.f2 = 'Monday';
s.f3 = 'Tuesday';
s.f4 = 'Wednesday';
s.f5 = 'Thursday';
s.f6 = 'Friday';
s.f7 = 'Saturday';

lengths = structfun(@numel, s)
```

Shorten the text in each field of `s`, created in the previous example. Because the output is nonscalar, set `UniformOutput` to `false`.

```
shortNames = structfun(@(x) ( x(1:3) ), s, 'UniformOutput', false)
```

The syntax `@(x)` creates an anonymous function.

Define and call a custom error handling function.

```
function result = errorfun(errorinfo, field)
   warning(errorinfo.identifier, errorinfo.message);
   result = NaN;
end
```

```
mystruct.f1 = 'text';
myresult = structfun(@(x) x^2, mystruct, 'ErrorHandler', @errorfun)
```

See Also

cellfun | arrayfun | function_handle | cell2mat | spfun

Introduced before R2006a

strvcat

Concatenate strings vertically

Note: `strvcat` is not recommended. Use `char` instead. Unlike `strvcat`, the `char` function does not ignore empty strings.

Syntax

```
S = strvcat(t1, t2, t3, ...)
S = strvcat(c)
```

Description

`S = strvcat(t1, t2, t3, ...)` forms the character array `S` containing the text strings (or string matrices) `t1,t2,t3,...` as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

`S = strvcat(c)` when `c` is a cell array of strings, passes each element of `c` as an input to `strvcat`. Empty strings in the input are ignored.

Examples

The command `strvcat('Hello','Yes')` is the same as `['Hello';'Yes ']`, except that `strvcat` performs the padding automatically.

```
t1 = 'first'; t2 = 'string'; t3 = 'matrix'; t4 = 'second';

S1 = strvcat(t1, t2, t3)        S2 = strvcat(t4, t2, t3)

S1 =                            S2 =

first                           second
string                          string
```

```
matrix                          matrix

S3 = strvcat(S1, S2)

S3 =
first
string
matrix
second
string
matrix
```

More About

Tips

If each text parameter, `ti`, is itself a character array, `strvcat` appends them vertically to create arbitrarily large string matrices.

See Also

`strcat` | `cat` | `vertcat` | `special character` | `horzcat` | `int2str` | `mat2str` | `num2str` | `strings`

Introduced before R2006a

sub2ind

Convert subscripts to linear indices

Syntax

```
linearInd = sub2ind(matrixSize, rowSub, colSub)
linearInd = sub2ind(arraySize, dim1Sub, dim2Sub, dim3Sub, ...)
```

Description

`linearInd = sub2ind(matrixSize, rowSub, colSub)` returns the linear index equivalents to the row and column subscripts `rowSub` and `colSub` for a matrix of size `matrixSize`. The `matrixSize` input is a 2-element vector that specifies the number of rows and columns in the matrix as [`nRows`, `nCols`]. The `rowSub` and `colSub` inputs are positive, whole number scalars or vectors that specify one or more row-column subscript pairs for the matrix. Example 3 demonstrates the use of vectors for the `rowSub` and `colSub` inputs.

`linearInd = sub2ind(arraySize, dim1Sub, dim2Sub, dim3Sub, ...)` returns the linear index equivalents to the specified subscripts for each dimension of an N-dimensional array of size `arraySize`. The `arraySize` input is an n-element vector that specifies the number of dimensions in the array. The `dimNSub` inputs are positive, whole number scalars or vectors that specify one or more row-column subscripts for the matrix.

All subscript inputs can be `single`, `double`, or any integer type. The `linearInd` output is always of class `double`.

If needed, `sub2ind` assumes that unspecified trailing subscripts are 1. See Example 2, below.

Examples

Example 1

This example converts the subscripts (2, 1, 2) for three-dimensional array `A` to a single linear index. Start by creating a 3-by-4-by-2 array `A`:

```
rng(0,'twister');   % Initialize random number generator.
A = rand(3, 4, 2)

A(:,:,1) =
    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
A(:,:,2) =
    0.9572    0.1419    0.7922    0.0357
    0.4854    0.4218    0.9595    0.8491
    0.8003    0.9157    0.6557    0.9340
```

Find the linear index corresponding to (2, 1, 2):

```
linearInd = sub2ind(size(A), 2, 1, 2)
linearInd =
    14
```

Make sure that these agree:

```
A(2, 1, 2)            A(14)
ans =                 and =
     0.4854               0.4854
```

Example 2

Using the 3-dimensional array `A` defined in the previous example, specify only 2 of the 3 subscript arguments in the call to `sub2ind`. The third subscript argument defaults to 1.

The command

```
linearInd = sub2ind(size(A), 2, 4)
ans =
    11
```

is the same as

```
linearInd = sub2ind(size(A), 2, 4, 1)
ans =
    11
```

Example 3

Using the same 3-dimensional input array `A` as in Example 1, accomplish the work of five separate `sub2ind` commands with just one.

Replace the following commands:

```
sub2ind(size(A), 3, 3, 2);
sub2ind(size(A), 2, 4, 1);
sub2ind(size(A), 3, 1, 2);
sub2ind(size(A), 1, 3, 2);
sub2ind(size(A), 2, 4, 1);
```

with a single command:

```
sub2ind(size(A), [3 2 3 1 2], [3 4 1 3 4], [2 1 2 2 1])
ans =
    21    11    15    19    11
```

Verify that these linear indices access the same array elements as their subscripted counterparts:

```
[A(3,3,2),   A(2,4,1),   A(3,1,2),   A(1,3,2),   A(2,4,1)]
ans =
    0.6557    0.1576    0.8003    0.7922    0.1576

A([21, 11, 15, 19, 11])
ans =
    0.6557    0.1576    0.8003    0.7922    0.1576
```

See Also

`ind2sub` | `find` | `size`

Introduced before R2006a

subplot

Create axes in tiled positions

Syntax

```
subplot(m,n,p)
subplot(m,n,p,'replace')
subplot(m,n,p,'align')
subplot('Position',positionVector)
subplot(m,n,p,ax)
subplot(___,Name,Value)

h = subplot(___)

subplot(h)
```

Description

`subplot(m,n,p)` divides the current figure into an `m`-by-`n` grid and creates an axes for a subplot in the position specified by `p`. MATLAB numbers its subplots by row, such that the first subplot is the first column of the first row, the second subplot is the second column of the first row, and so on. If the axes already exists, then the command `subplot(m,n,p)` makes the subplot in position `p` the current axes.

`subplot(m,n,p,'replace')` deletes any existing axes in position `p` and creates a new axes.

`subplot(m,n,p,'align')` creates a new axes so that the plot boxes are aligned. This is the default behavior.

`subplot('Position',positionVector)` creates a new axes at the position specified by `positionVector`. The `positionVector` is a four-element vector of the form `[left,bottom,width,height]`, such that the entries are normalized values between 0.0 to 1.0. If the position vector specifies an axes that overlaps any previous axes, then the new axes replaces the existing ones.

`subplot(m,n,p,ax)` converts the existing axes, `ax`, into a subplot axes in the same figure.

`subplot(___,Name,Value)` specifies properties for the axes using any of the input argument combinations in the previous syntaxes and one or more `Name,Value` pair arguments.

`h = subplot(___)` returns the axes object created by the `subplot` function.

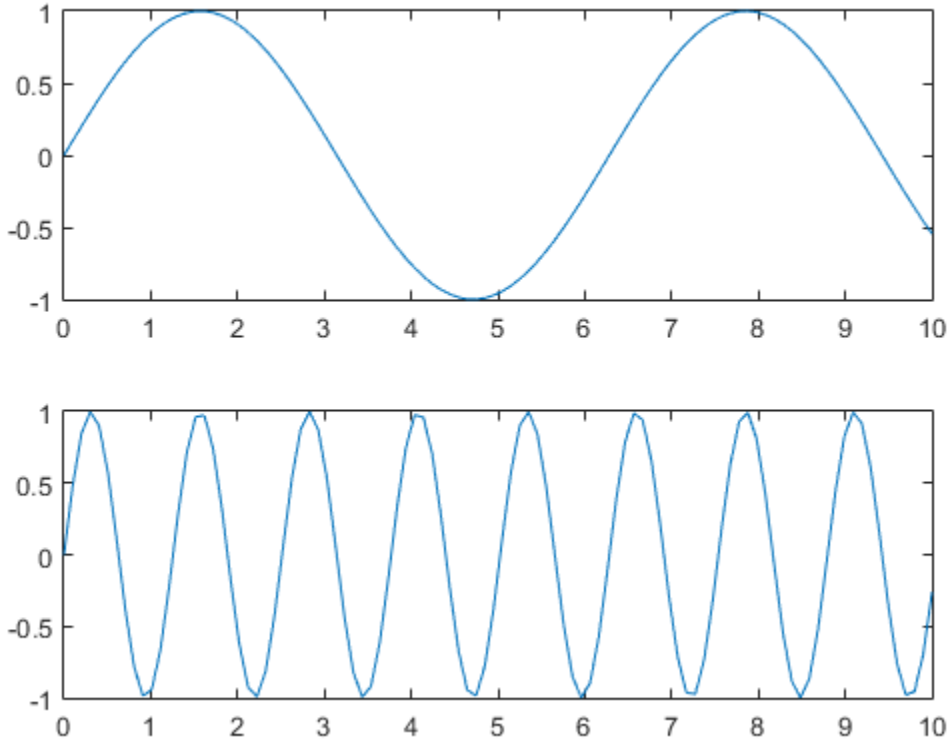
`subplot(h)` makes the axes with handle `h` the current axes of its figure, but does not make its figure the current figure if it is not already the current figure.

Examples

Upper and Lower Subplots

Create a figure with two stacked subplots. Plot a sine wave in each axes.

```
x = linspace(0,10);
y1 = sin(x);
y2 = sin(5*x);

figure
subplot(2,1,1);
plot(x,y1)

subplot(2,1,2);
plot(x,y2)
```



Quadrant of Subplots

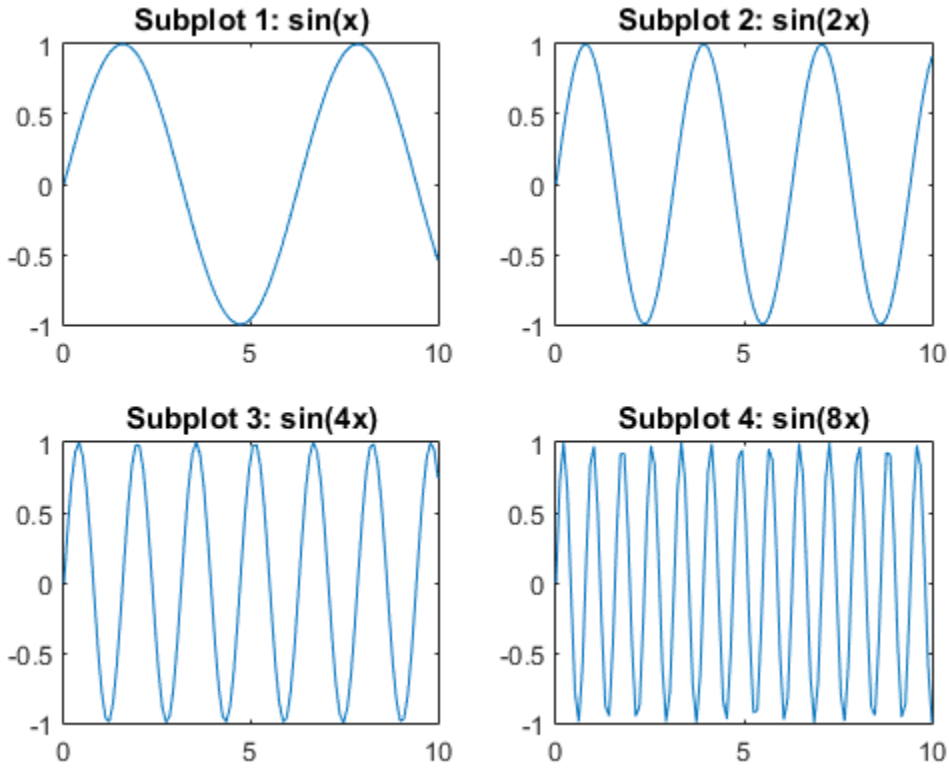
Create a figure divided into four subplots.

Define the data.

```
x = linspace(0,10);
y1 = sin(x);
y2 = sin(2*x);
y3 = sin(4*x);
y4 = sin(8*x);
```

Plot the four sine waves and title each subplot.

```
figure
subplot(2,2,1)
plot(x,y1)
title('Subplot 1: sin(x)')

subplot(2,2,2)
plot(x,y2)
title('Subplot 2: sin(2x)')

subplot(2,2,3)
plot(x,y3)
title('Subplot 3: sin(4x)')

subplot(2,2,4)
plot(x,y4)
title('Subplot 4: sin(8x)')
```



Subplots with Different Sizes

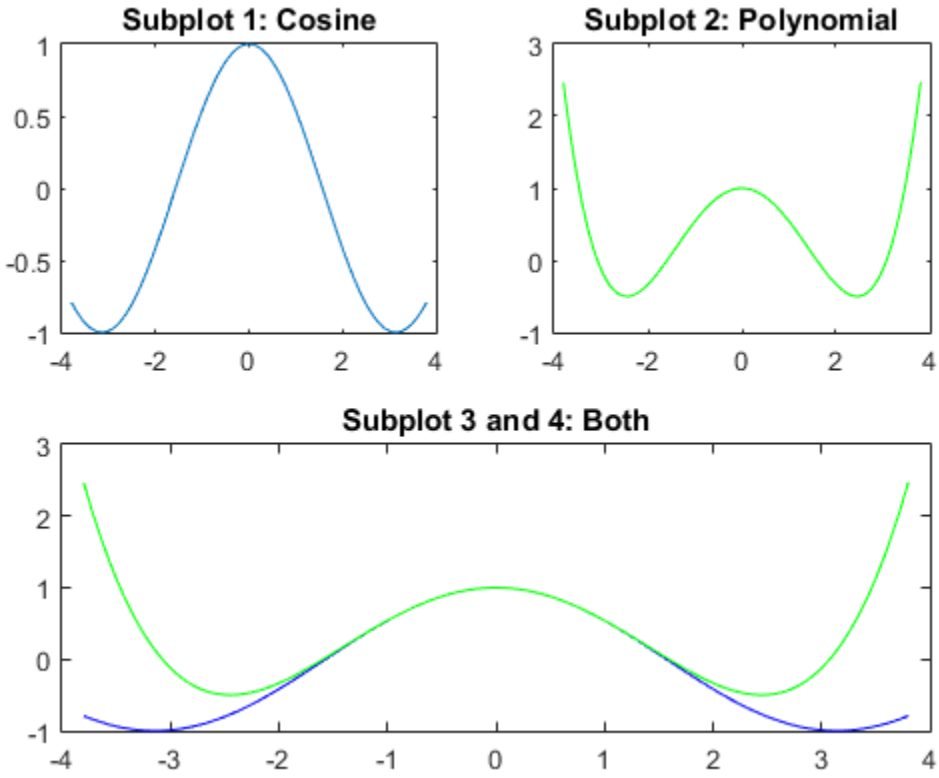
Create a figure containing subplots with different sizes.

Define a vector of sine values and a vector of polynomial values.

```
x = linspace(-3.8,3.8);
y_cos = cos(x);
y_poly = 1 - x.^2./2 + x.^4./24;
```

Plot the sine wave in the first subplot and the polynomial in the second subplot. Create a third subplot that spans the lower half of the figure and plot both vectors together. Add titles to each subplot.

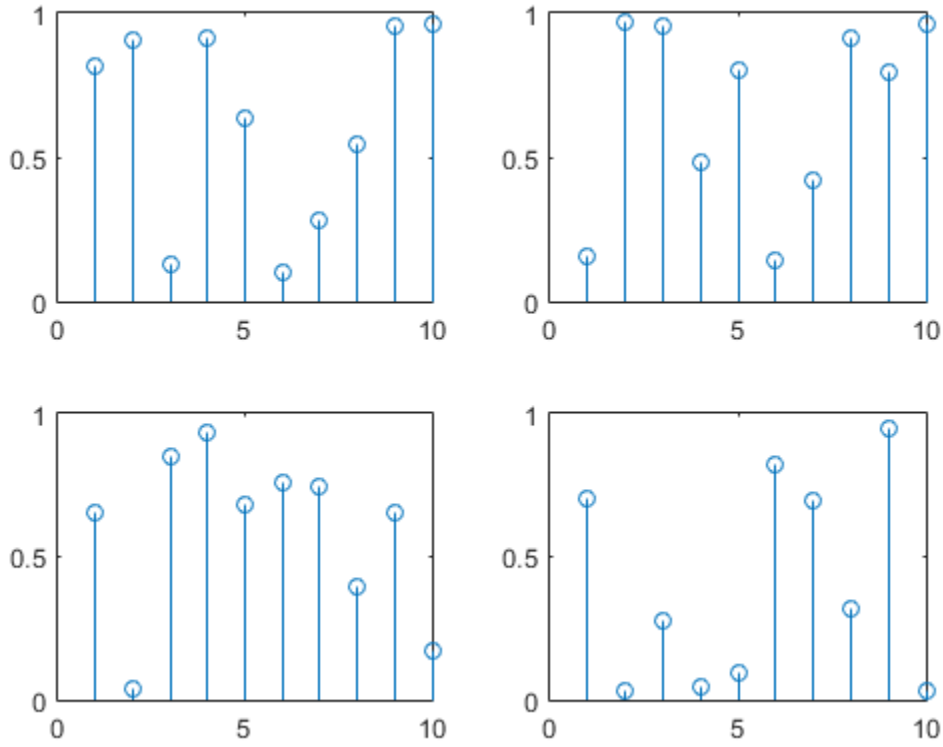
```
figure
subplot(2,2,1);
plot(x,y_cos);
title('Subplot 1: Cosine')

subplot(2,2,2);
plot(x,y_poly,'g');
title('Subplot 2: Polynomial')

subplot(2,2,[3,4]);
plot(x,y_cos,'b',x,y_poly,'g');
title('Subplot 3 and 4: Both')
```



Replace Subplot with Empty Axes

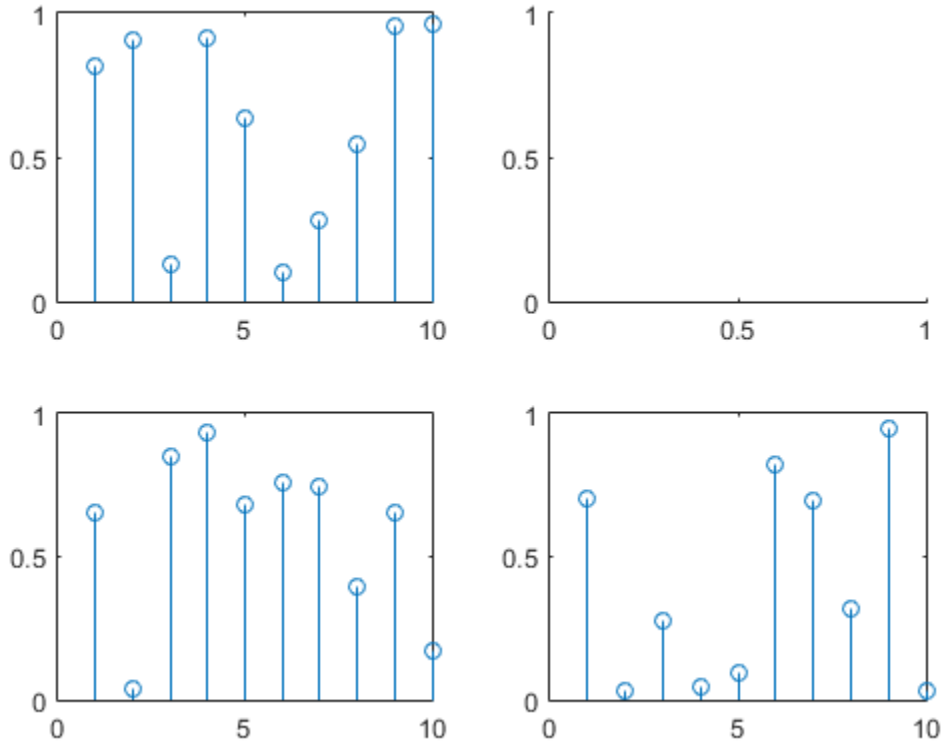
Initialize the random number generator. Use a loop to create a figure with four stem plots of random data.

```
rng default
figure
for k = 1:4
    data = rand(1,10);
    subplot(2,2,k)
    stem(data)
end
```



Replace the second subplot with an empty axes.

```
subplot(2,2,2,'replace')
```



Subplots at Specified Positions

Create a figure with two subplots that are not aligned.

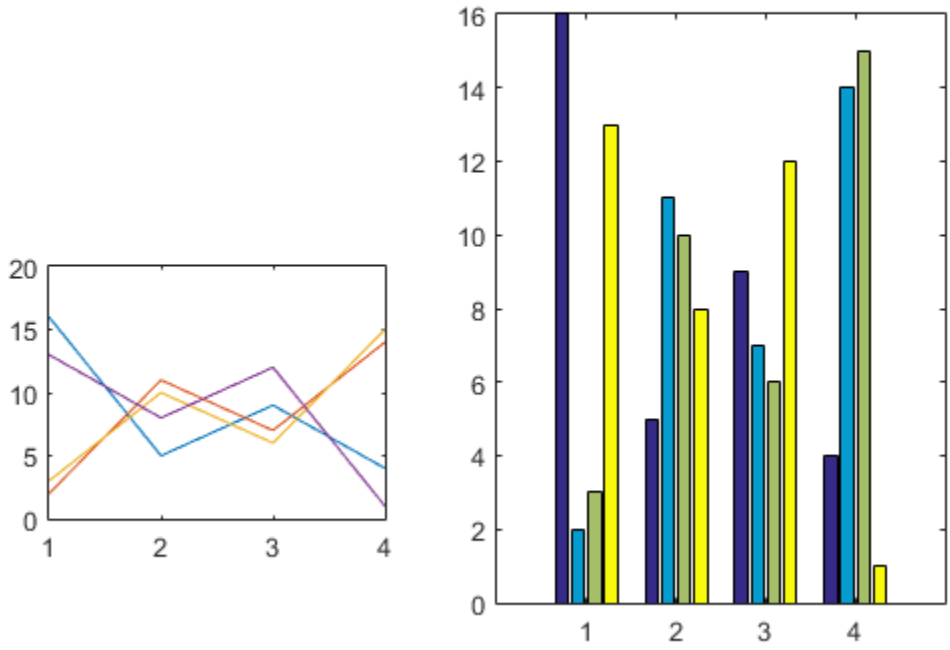
Define `y` as data from the `magic` function.

```
y = magic(4);
```

Plot `y` in one subplot. Create a bar graph of `y` in a second subplot. Specify a custom position for each subplot.

```
figure
positionVector1 = [0.1, 0.2, 0.3, 0.3];
```

```
subplot('Position',positionVector1)
plot(y)

positionVector2 = [0.5, 0.1, 0.4, 0.7];
subplot('Position',positionVector2)
bar(y)
```

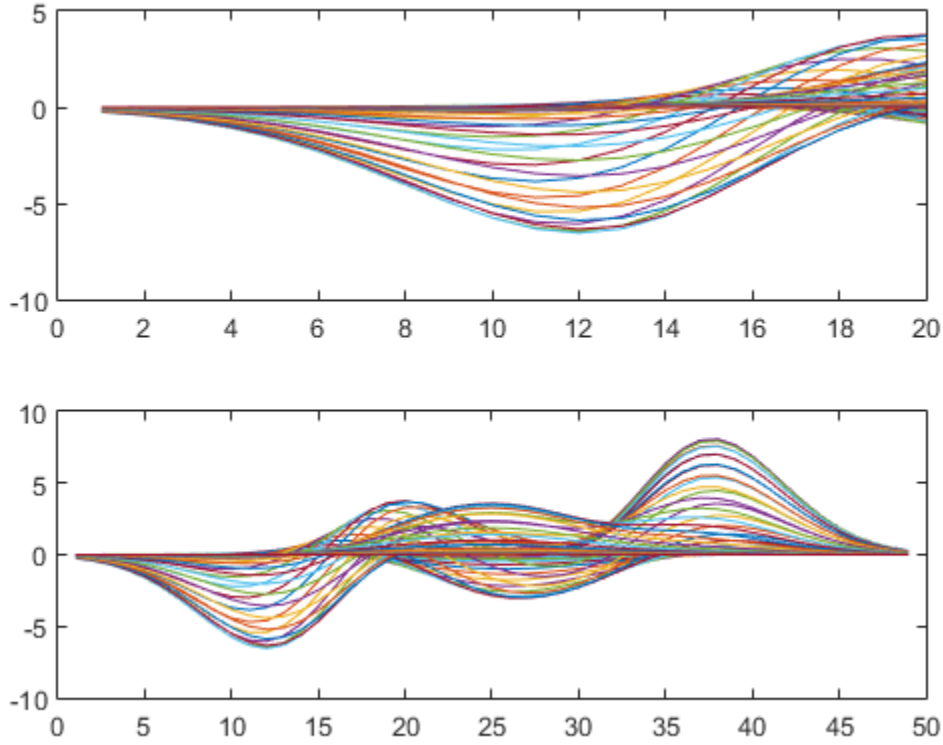


Return Subplot Axes Handle

Create a figure with two subplots. Plot the first 20 rows of the `peaks` function in the upper subplot. Plot the entire data set in the lower subplot.

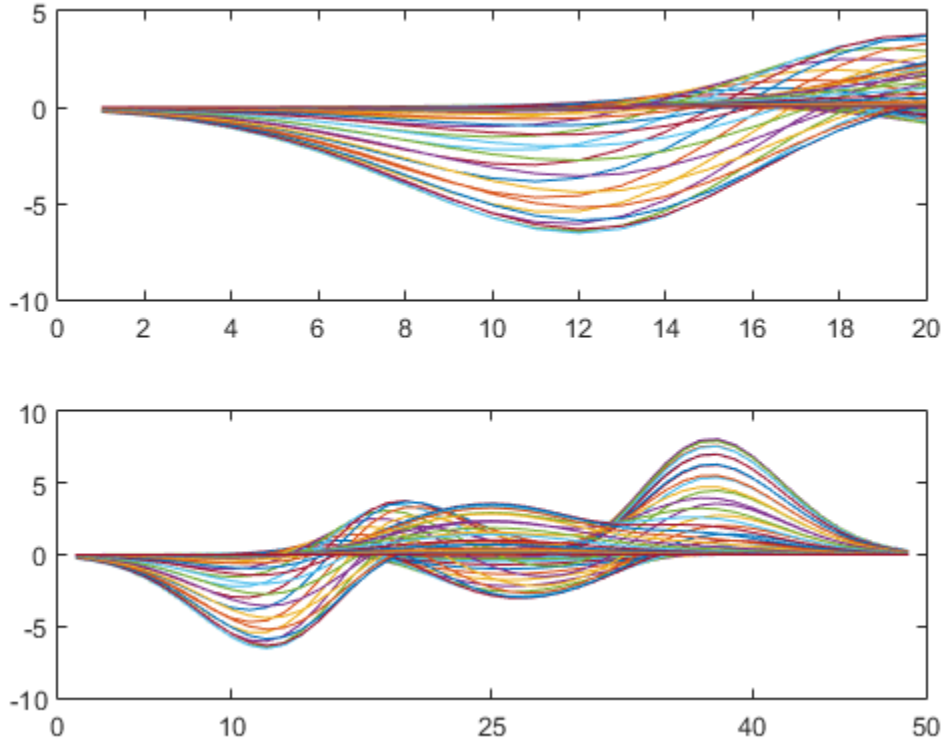
```
ax1 = subplot(2,1,1);
```

```
Z = peaks;
plot(ax1,Z(1:20,:))

ax2 = subplot(2,1,2);
plot(ax2,Z)
```



Change the tick marks for the lower subplot. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
ax2.XTick = [0,10,25,40,50];
```



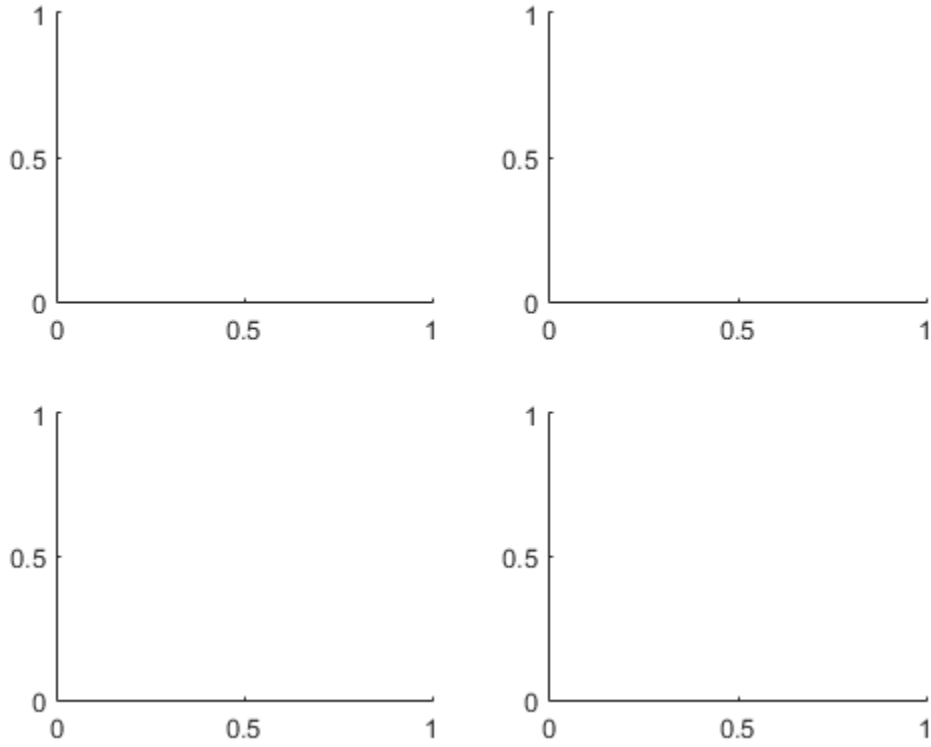
Some plotting functions set axes properties. Execute plotting functions before specifying axes properties to avoid overriding existing axes property settings.

Make Subplot Axes the Current Axes

Make a subplot the current axes using its axes handle.

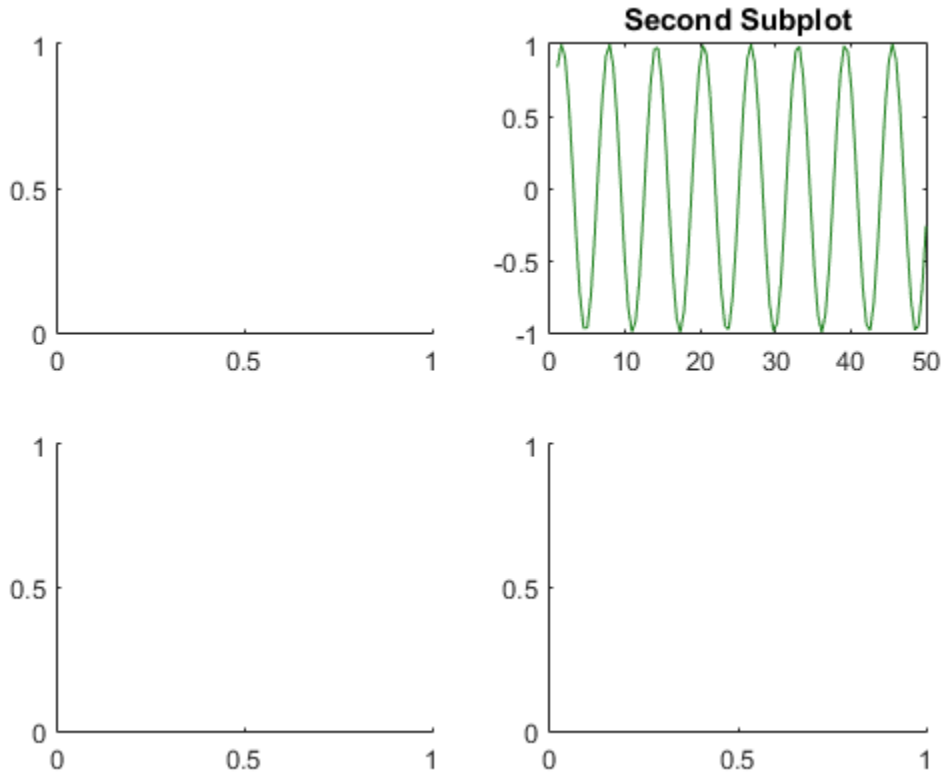
Create a figure with multiple subplots. Store the subplot axes handles in vector `h`.

```
figure;
for k = 1:4
    h(k) = subplot(2,2,k);
end
```



Make the second subplot the current axes. Plot a sine wave and change the axis limits.

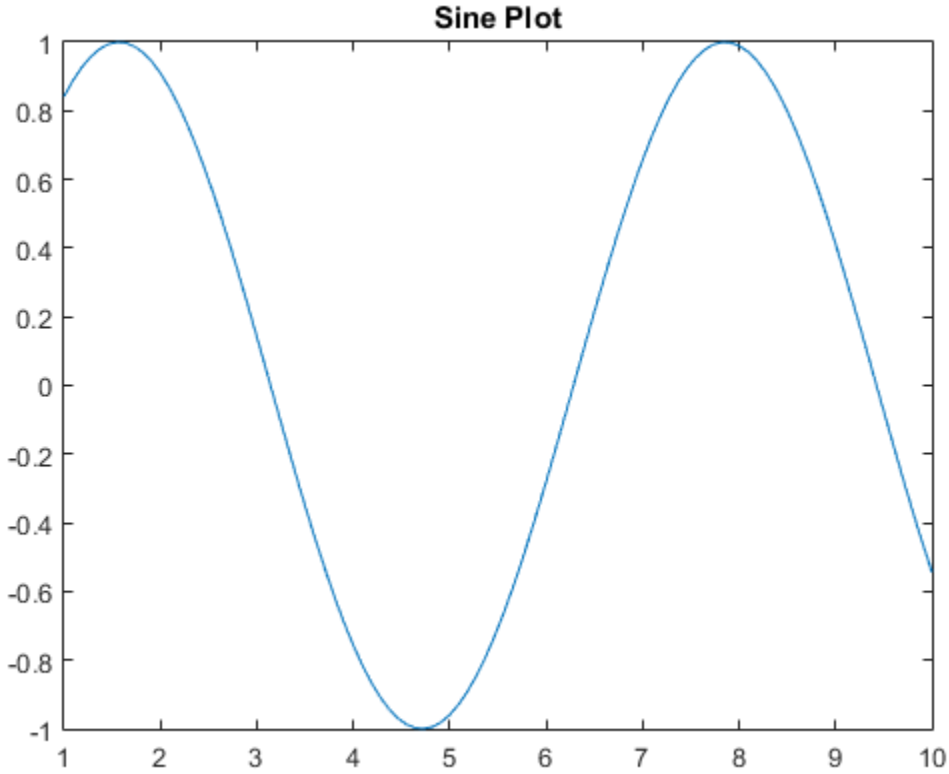
```
x = linspace(1,50);
y = sin(x);

subplot(h(2))
plot(x,y,'Color',[0.1, 0.5, 0.1])
title('Second Subplot')
axis([0,50,-1,1])
```



Convert Existing Axes to Subplot

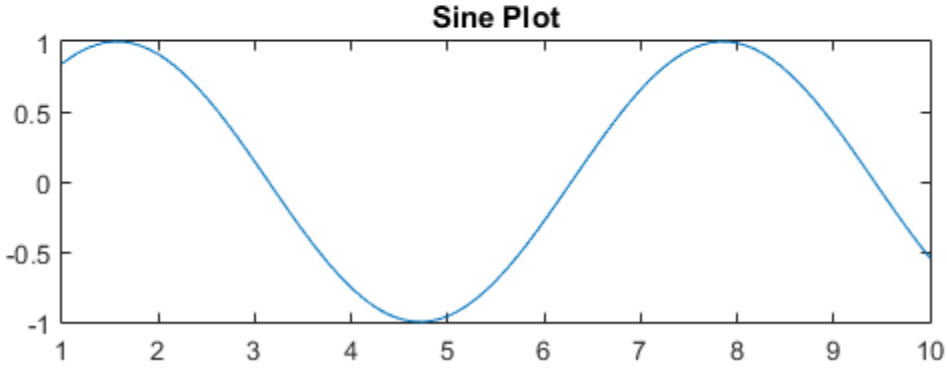
Create a plot with a title.

```
x = linspace(1,10);
y = sin(x);
plot(x,y)
title('Sine Plot')
```



Get the current axes object using `gca`. Convert the current axes so that it is the lower subplot of the figure.

```
ax1 = gca;
subplot(2,1,2,ax1)
```



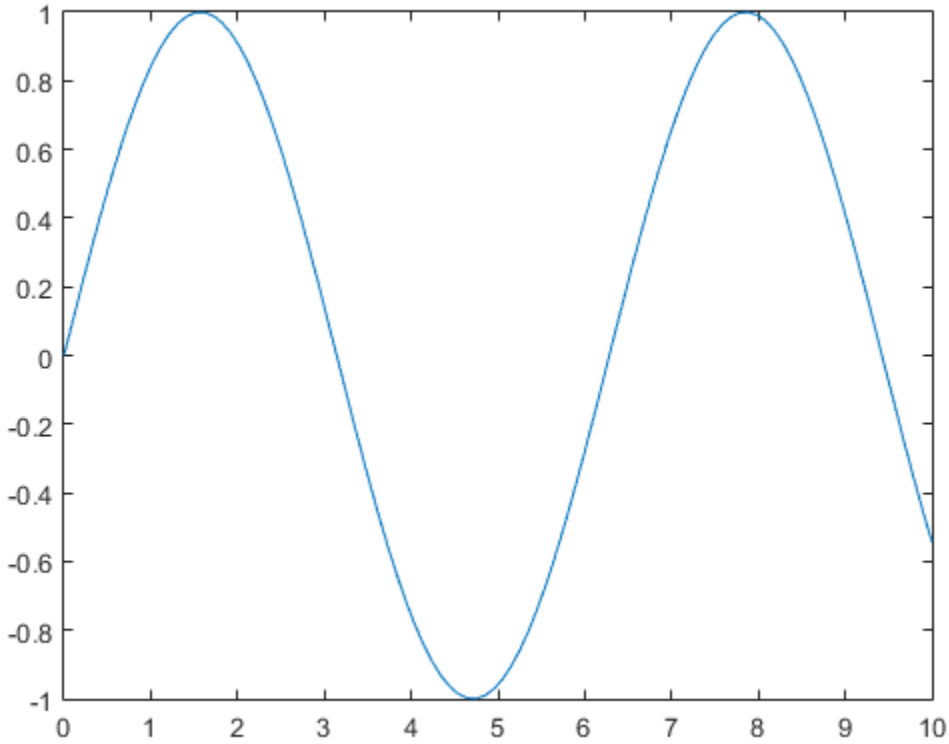
The `subplot` function uses the figure in which the original axes existed.

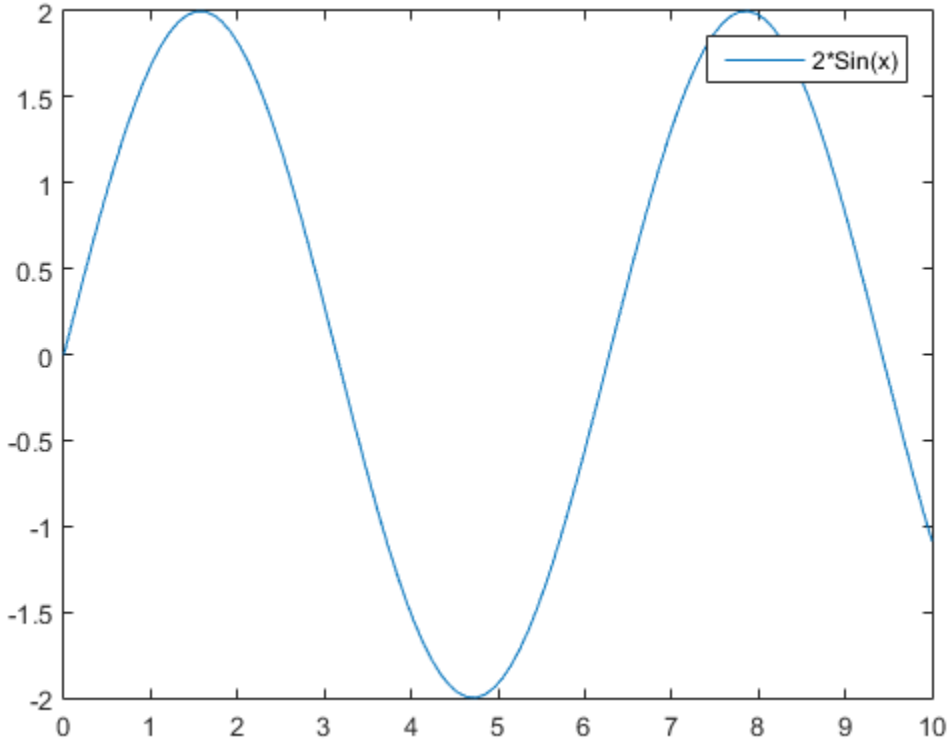
Combine Two Axes Into Single Figure With Subplots

Create two plots in two different figures and store the axes objects.

```
x = linspace(0,10);
y1 = sin(x);
figure
plot(x,y1)
ax1 = gca;

y2 = 2*sin(x);
```

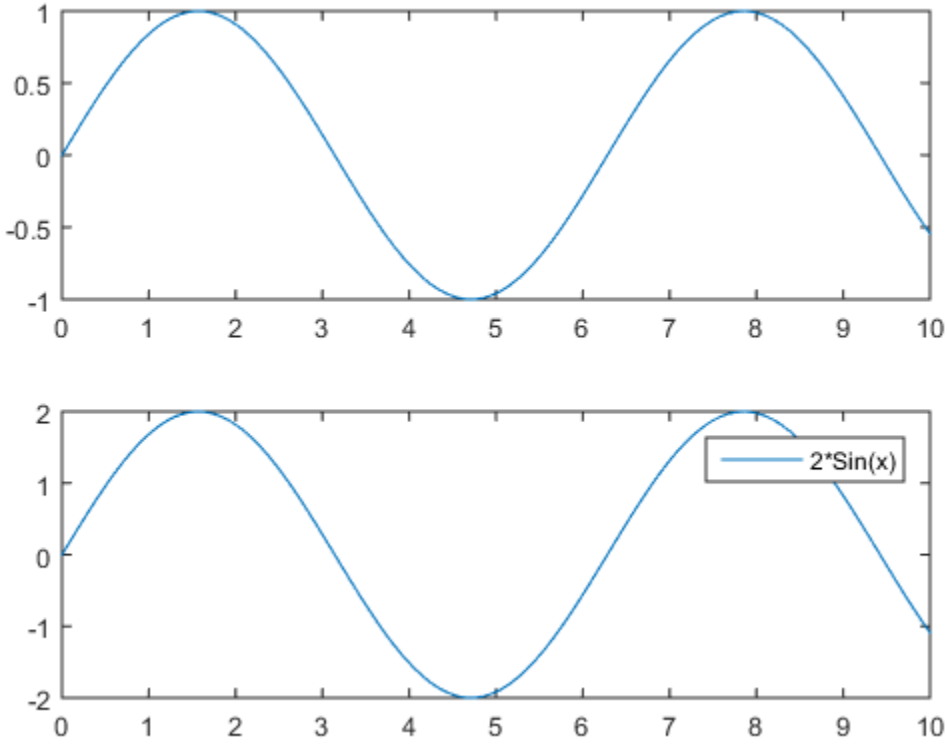
```
figure
plot(x,y2)
leg = legend('2*Sin(x)');
ax2 = gca;
```





Create copies of the two axes objects using `copyobj`. Set the parent of the copied axes to a third figure. Since legends and colorbars objects do not get copied with the associated axes, copy the legend with the axes.

```
f3 = figure;
ax1_copy = copyobj(ax1,f3);
subplot(2,1,1,ax1_copy)

copies = copyobj([ax2,leg],f3);
ax2_copy = copies(1);
subplot(2,1,2,ax2_copy)
```



Input Arguments

`m` — Number of grid rows

1 (default) | positive integer

Number of grid rows, specified as a positive integer.

Data Types: `single` | `double`

`n` — Number of grid columns

1 (default) | positive integer

Number of grid columns, specified as a positive integer.

Data Types: `single | double`

`p` — Grid position for new axes

positive integer | vector

Grid position for the new axes, specified as a positive integer or a vector of positive integers.

- If `p` is a positive integer, then `subplot` creates a new axes in grid position `p`.
- If `p` is a vector of positive integers, then `subplot` creates a new axes that spans the grid positions listed in `p`. For example, `subplot(2,3,[2,5])` creates one axes spanning positions 2 and 5. Use `subplot(2,3,[2,6])` to create one axes spanning positions 2, 3, 5, and 6.

Data Types: `single | double`

`positionVector` — Normalized position for new axes

four-element vector of values between 0.0 and 1.0

Normalized position for the new axes, specified as a four-element vector of values between 0.0 and 1.0 with the form `[left,bottom,width,height]`. The values are normalized with respect to the interior of the figure. The first two elements specify the position of the bottom-left corner of the axes in relation to the bottom-left corner of the figure. The last two elements specify the width and height of the new axes. Use this syntax to position an axes that does not align with grid positions.

Example: [0.1, 0.1, 0.35, 0.35]

Data Types: `single | double`

`ax` — Existing axes to convert to subplot

axes object

Existing axes to convert to a subplot axes, specified as an axes object. Use `gca` to refer to the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

You can set any axes properties for a subplot. Some plotting functions override existing axes property settings. Execute plotting functions before specifying axes properties to avoid overriding them. For a list of axes properties see Axes Properties.

Example: `'XGrid','on'`

Output Arguments

h — Axes object

scalar

Axes object, returned as a scalar. This is a unique identifier, which you can use to query and modify the properties of a specific axes.

More About

Tips

- `subplot(111)` is an exception and not identical in behavior to `subplot(1,1,1)`. For reasons of backwards compatibility, `subplot(111)` is a special case of subplot that does not immediately create an axes, but sets up the figure so that the next graphics command executes `clf reset`. The next graphics command deletes all the figure children and creates a new axes in the default position. `subplot(111)` does not return a handle and an error occurs if code specifies a return argument.
- If a new subplot axes overlaps an existing axes, then MATLAB deletes the existing axes. To overlay a new axes on top of existing subplots, use the `axes` command. For example, `subplot('Position',[.35 .35 .3 .3])` deletes any underlying subplots, but `axes('Position',[.35 .35 .3 .3])` positions a new axes in the middle of the figure without deleting any underlying axes.
- When using a script to create subplots, MATLAB does not finalize the `Position` property value until either a `drawnow` command is issued or MATLAB returns to await a user command. The `Position` property value for a subplot is subject to change until the script either refreshes the plot or exits.

See Also

Functions

`axes` | `cla` | `clf` | `figure` | `gca`

Properties
Axes Properties

Introduced before R2006a

subsasgn

Subscripted assignment

Syntax

```
A = subsasgn(A, S, B)
```

Description

`A = subsasgn(A, S, B)` is called by MATLAB for the syntax `A(i) = B`, `A{i} = B`, or `A.i = B` when `A` is an object.

MATLAB uses the built-in `subsasgn` function to interpret indexed assignment statements. Modify the indexed assignment behavior of classes by overloading `subsasgn` in the class.

If `A` is a fundamental class (see "Fundamental MATLAB Classes"), then an indexed reference to `A` calls the built-in `subsasgn` function. It does not call a `subsasgn` method that you have overloaded for that class. Therefore, if `A` is an array of class `double`, and there is an `@double/subsasgn` method on your MATLAB path, the statement `A(I) = B` calls the MATLAB built-in `subsasgn` function.

Input Arguments

A

Object

Default:

S

`struct` array with two fields, `type` and `subs`.

- `type` is a string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.

- `subs` is a cell array or string containing the actual subscripts.

Default:

`B`

Assignment value (right-hand side)

Default:

Output Arguments

`A`

Result of evaluating assignment.

Examples

See how MATLAB calls `subsasgn` for the expression:

```
A(1:2,:) = B;
```

The syntax `A(1:2,:) = B` calls `A = subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type = '()'` and `S.subs = {1:2,':'}`. The string `':'` indicates a colon used as a subscript.

See how MATLAB calls `subsasgn` for the expression:

```
A{1:2} = B;
```

The syntax `A{1:2} = B` calls `A = subsasgn(A,S,B)` where `S.type = '{}'` and `S.subs = {[1 2]}`.

See how MATLAB calls `subsasgn` for the expression:

```
A.field = B;
```

The syntax `A.field = B` calls `A = subsasgn(A,S,B)` where `S.type = '.'` and `S.subs = 'field'`.

See how MATLAB calls `subsasgn` for the expression:

```
A(1,2).name(3:5)=B;
```

Simple calls combine in a straightforward way for more complicated indexing expressions. In such cases, `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)=B` calls `A=subsasgn(A,S,B)` where `S` is a 3-by-1 structure array with the following values:

```
S(1).type = '()'       S(2).type = '.'        S(3).type = '()'
S(1).subs = {1,2}      S(2).subs = 'name'     S(3).subs = {[3 4 5]}
```

More About

Tips

Within a class's own methods, MATLAB calls the built-in `subsasgn`, not the class defined `subsasgn`. This behavior enables to use the default `subsasgn` behavior when defining specialized indexing for your class. See "Built-In subsref and subsasgn Called In Methods" for more information.

Algorithms

In the assignment `A(J,K,...) = B(M,N,...)`, subscripts `J`, `K`, `M`, `N`, and so on, can be scalar, vector, or arrays, when all the following are true:

- The number of subscripts specified for `B`, excluding trailing subscripts equal to 1, does not exceed the value returned by `ndims(B)`.
- The number of nonscalar subscripts specified for `A` equals the number of nonscalar subscripts specified for `B`. For example, `A(5,1:4,1,2) = B(5:8)` is valid because both sides of the equation use one nonscalar subscript.
- The order and length of all nonscalar subscripts specified for `A` matches the order and length of nonscalar subscripts specified for `B`. For example, `A(1:4, 3, 3:9) = B(5:8, 1:7)` is valid because both sides of the equation (ignoring the one scalar subscript `3`) use a 4-element subscript followed by a 7-element subscript.

  See `numel` for information concerning the use of `numel` with regards to the overloaded `subsasgn` function.

  For information on the syntax to use when overloading `subsasgn`, see "Overloading numel, subsref, and subsasgn"

- “Overloading numel, subsref, and subsasgn”

See Also

`subsref` | `substruct`

Introduced before R2006a

subsindex

Subscript indexing with object

Syntax

```
ind = subsindex(A)
```

Description

`ind = subsindex(A)` called by MATLAB for the expression `X(A)` when `A` is an object, unless such an expression results in a call to an overloaded `subsref` or `subsasgn` method for `X`. `subsindex` must return the value of the object as a zero-based integer index. (`ind` must contain integer values in the range `0` to `prod(size(X))-1`.) Call `subsindex` directly from an overloaded `subsref` or `subsasgn` method.

MATLAB invokes `subsindex` separately on all the subscripts in an expression, such as `X(A,B)`.

See Also

`subsasgn` | `subsref`

Tutorials

-
- “Objects In Index Expressions”

Introduced before R2006a

subspace

Angle between two subspaces

Syntax

```
theta = subspace(A,B)
```

Description

`theta = subspace(A,B)` finds the angle between two subspaces specified by the columns of `A` and `B`. If `A` and `B` are column vectors of unit length, this is the same as `acos(abs(A'*B))`.

Examples

Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

```
H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);
```

Note that matrices `A` and `B` are different sizes — `A` has three columns and `B` four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

```
theta = subspace(A,B)
theta =
    1.5708
```

That `A` and `B` are orthogonal is shown by the fact that `theta` is equal to $\pi/2$.

```
theta - pi/2
ans =
     0
```

More About

Tips

If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations `A`, and a second realization of the experiment described by `B`, `subspace(A,B)` gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Introduced before R2006a

subsref

Redefine subscripted reference for objects

Syntax

```
B = subsref(A,S)
```

Description

`B = subsref(A,S)` is called by MATLAB for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object. `S` is a `struct` array with two fields, `type` and `subs`.

The `type` field is string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields. The subs field is a cell array or a string containing the actual subscripts.

`B` is the result of the indexed expression.

MATLAB uses the built-in `subsref` function to interpret indexed references to objects. To modify the indexed reference behavior of objects, overload `subsref` in the class.

If `A` is a fundamental class (see "Fundamental MATLAB Classes"), then an indexed reference to `A` calls the built-in `subsref` function. It does not call a `subsref` method that you have overloaded for that class. Therefore, if `A` is an array of class `double`, and there is an `@double/subsref` method on your MATLAB path, the statement `A(I)` calls the MATLAB built-in `subsref` function.

Examples

See how MATLAB calls `subsref` for the expression:

```
A(1:2,:)
```

The syntax `A(1:2,:)` calls `B = subsref(A,S)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs={1:2,':'}`. The string `':'` indicates a colon used as a subscript.

See how MATLAB calls `subsref` for the expression:

```
A{1:2}
```

The syntax `A{1:2}` calls `B = subsref(A,S)` where `S.type='{}'` and `S.subs={[1 2]}`.

See how MATLAB calls `subsref` for the expression:

```
A.field
```

The syntax `A.field` calls `B = subsref(A,S)` where `S.type='.'` and `S.subs='field'`.

See how MATLAB calls `subsref` for the expression:

```
A(1,2).name(3:5)
```

Simple calls combine in a straightforward way for more complicated indexing expressions. In such cases, `length(S)` is the number of subscript levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is a 3-by-1 structure array with the following values:

```
S(1).type='()'          S(2).type='.'           S(3).type='()'
S(1).subs={1,2}         S(2).subs='name'        S(3).subs={[3 4 5]}
```

More About

Tips

Within a class's own methods, MATLAB calls the built-in `subsref`, not the class defined `subsref`. This behavior enables to use the default `subsref` behavior when defining specialized indexing for your class. See “Built-In subsref and subsasgn Called In Methods” for more information.

For information on the syntax to use when overloading `subsref`, see “Overloading numel, subsref, and subsasgn”

- “Overloading numel, subsref, and subsasgn”

See Also

`numel` | `subsasgn` | `substruct`

Introduced before R2006a

substruct

Create structure argument for `subsasgn` or `subsref`

Syntax

```
S = substruct(type1, subs1, type2, subs2, ...)
```

Description

`S = substruct(type1, subs1, type2, subs2, ...)` creates a structure with the fields required by an overloaded `subsref` or `subsasgn` method. Each `type` string must be one of `'.'`, `'()'`, or `'{}'`. The corresponding `subs` argument must be either a field name (for the `'.'` type) or a cell array containing the index vectors (for the `'()'` or `'{}'` types).

Output Arguments

S

`struct` with these fields:

- `type`: one of `'.'`, `'()'`, or `'{}'`
- `subs`: subscript values (field name or cell array of index vectors)

Examples

Call `subsref` with arguments equivalent to the syntax:

```
B = A(3,5).field;
```

where `A` is an object of a class that implements a `subsref` method

Use `substruct` to form the input `struct`, `S`:

```
S = substruct('()',{3,5},'.','field');
```

Call the class method:

```
B = subsref(A,S);
```

The `struct` created by `substruct` in this example contains:

```
S(1)

ans =

    type: '()'
    subs: {[3]  [5]}

S(2)

ans =

    type: '.'
    subs: 'field'
```

See Also

`subsasgn` | `subsref`

Introduced before R2006a

subvolume

Extract subset of volume data set

Syntax

```
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)
[Nx,Ny,Nz,Nv] = subvolume(V,limits)
Nv = subvolume(...)
```

Description

`[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)` extracts a subset of the volume data set `V` using the specified axis-aligned `limits`. `limits = [xmin,xmax,ymin, ymax,zmin,zmax]` (Any `NaN`s in the limits indicate that the volume should not be cropped along that axis.)

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The subvolume is returned in `NV` and the coordinates of the subvolume are given in `NX`, `NY`, and `NZ`.

`[Nx,Ny,Nz,Nv] = subvolume(V,limits)` assumes the arrays `X`, `Y`, and `Z` are defined as

```
[X,Y,Z] = meshgrid(1:N,1:M,1:P)
```

where `[M,N,P] = size(V)`.

`Nv = subvolume(...)` returns only the subvolume.

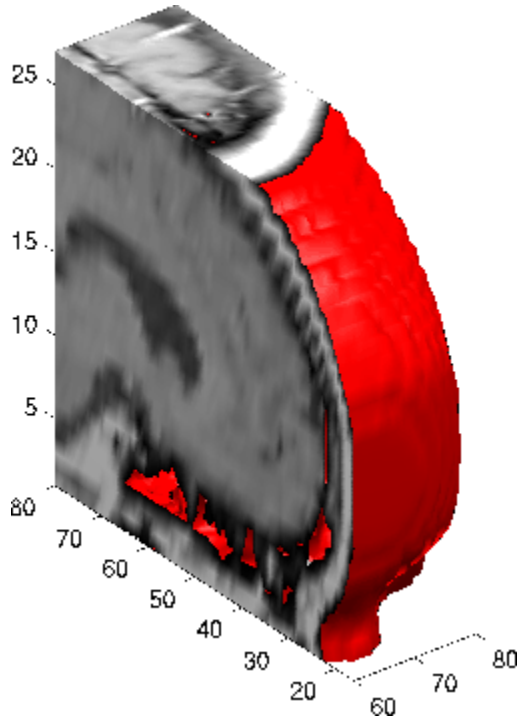
Examples

This example uses a data set that is a collection of MRI slices of a human skull. The data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then a subset of the data is extracted (`subvolume`).

- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).
- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).
- Adding lights to the right and left of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = subvolume(D,[60,80,nan,80,nan,nan]);
p1 = patch(isosurface(x,y,z,D, 5),...
     'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
     'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight right; camlight left; lighting gouraud
```



See Also

`isocaps` | `isonormals` | `isosurface` | `reducepatch` | `reducevolume` | `smooth3`

Introduced before R2006a

sum

Sum of array elements

Syntax

```
S = sum(A)
S = sum(A,dim)
S = sum( ___ ,outtype)
S = sum( ___ ,nanflag)
```

Description

`S = sum(A)` returns the sum of the elements of A along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `sum(A)` returns the sum of the elements.
- If `A` is a matrix, then `sum(A)` returns a row vector containing the sum of each column.
- If `A` is a multidimensional array, then `sum(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes `1` while the sizes of all other dimensions remain the same.

`S = sum(A,dim)` returns the sum along dimension `dim`. For example, if `A` is a matrix, then `sum(A,2)` is a column vector containing the sum of each row.

`S = sum( ___ ,outtype)` returns the sum with a specified data type, using any of the input arguments in the previous syntaxes. `outtype` can be `'default'`, `'double'`, or `'native'`.

`S = sum( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `sum(A,'includenan')` includes all `NaN` values in the calculation while `sum(A,'omitnan')` ignores them.

Examples

Sum of Vector Elements

Create a vector and compute the sum of its elements.

```
A = 1:10;
S = sum(A)

S =

    55
```

Sum of Matrix Columns

Create a matrix and compute the sum of the elements in each column.

```
A = [1 3 2; 4 2 5; 6 1 4]

A =

     1     3     2
     4     2     5
     6     1     4

S = sum(A)

S =

    11     6    11
```

Sum of Matrix Rows

Create a matrix and compute the sum of the elements in each row.

```
A = [1 3 2; 4 2 5; 6 1 4]

A =

     1     3     2
     4     2     5
     6     1     4

S = sum(A,2)
```

```
S =

     6
    11
    11
```

Sum of 3-D Array

Create a 4-by-2-by-3 array of ones and compute the sum along the third dimension.

```
A = ones(4,2,3);
S = sum(A,3)
```

```
S =

     3     3
     3     3
     3     3
     3     3
```

Integer Output Type Saturation

Create a vector of 8-bit integers and compute the `int8` sum of its elements by specifying the output type as `native`.

```
A = int8(1:20);
S = sum(A,'native')
```

```
S =

  127
```

Compute the sum using the default `double` output type (equivalent to `S = sum(A,'default')`).

```
S = sum(A)
```

```
S =
```

```
   210
```

The two sums differ because the `int8` output is saturated while the `double` output reflects the expected sum of elements.

Sum Excluding `NaN`

Create a vector and compute its sum, excluding `NaN` values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
S = sum(A,'omitnan')
```

```
S =

    3.3250
```

If you do not specify `'omitnan'`, then `sum(A)` returns `NaN`.

Input Arguments

`A` — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If `A` is a scalar, then `sum(A)` returns `A`.
- If `A` is an empty 0-by-0 matrix, then `sum(A)` returns `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `duration`
Complex Number Support: Yes

`dim` — Dimension to operate along

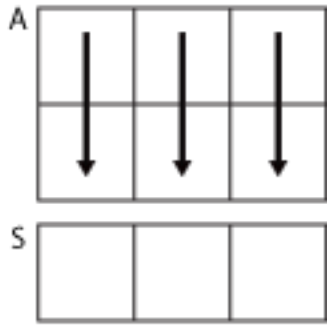
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(S,dim)` is `1`, while the sizes of all other dimensions remain the same.

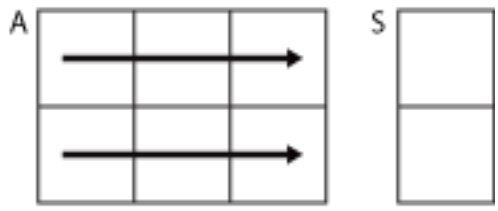
Consider a two-dimensional input array, `A`:

- `sum(A,1)` operates on successive elements in the columns of `A` and returns a row vector of the sums of each column.



sum(A,1)

- `sum(A,2)` operates on successive elements in the rows of `A` and returns a column vector of the sums of each row.



sum(A,2)

`sum` returns `A` when `dim` is greater than `ndims(A)` or when `size(A,dim)` is `1`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`outtype` — Output data type

`'default'` (default) | `'double'` | `'native'`

Output data type, specified as `'default'`, `'double'`, or `'native'`. These options also specify the data type in which the operation is performed.

`outtype`	Output data type
`'default'`	`double`, unless the input data type is `single` or `duration`, in which case, the output is `'native'`
`'double'`	`double`, unless the data type is `duration`, in which case, `'double'` is not supported
`'native'`	same data type as the input, unless the input data type is `char`, in which case, `'native'` is not supported

Data Types: `char`

`nanflag` — `NaN` condition

'`includenan`' (default) | '`omitnan`'

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values when computing the sum, resulting in `NaN`.
- `'omitnan'` — Ignore all `NaN` values in the input.

The `sum` function does not support the `nanflag` option for `duration` arrays.

Data Types: `char`

See Also

`cumsum` | `diff` | `mean` | `prod`

summary

Print summary of table or categorical array

Syntax

```
summary(T)
```

```
summary(A)
summary(A,dim)
```

Description

`summary(T)` prints a summary of the table, `T`. The table summary displays the table description from `T.Properties.Description` followed by a summary of the table variables.

`summary(A)` prints a summary of the categorical array, `A`.

- If `A` is a vector, then `sumamry(A)` displays the category names along with the number of elements in each category (the category counts). Furthermore, the summary contains the number of elements that are undefined.
- If `A` is a matrix, then `summary` treats the columns of `A` as vectors and displays the category counts for each column of `A`.
- If `A` is a multidimensional array, then `summary` acts along the first array dimension whose size does not equal `1`.

`summary(A,dim)` prints the category counts of the categorical array, `A`, along dimension dim.

For example, you can display the counts of each row in a categorical array using `counts(A,2)`.

Examples

Summary of Table

Create a table.

```
load patients
BloodPressure = [Systolic Diastolic];
T = table(Gender,Age,Smoker,BloodPressure,'RowNames',LastName);
```

Add descriptions and units to table `T`.

```
T.Properties.Description = 'Simulated patient data';
T.Properties.VariableUnits =  {''  'Yrs' ''  'mm Hg'};
T.Properties.VariableDescriptions{4} = 'Systolic/Diastolic';
```

Print a summary of table `T`.

```
format compact

summary(T)
```

```
Description:  Simulated patient data
Variables:
    Gender: 100x1 cell string
    Age: 100x1 double
        Units:  Yrs
        Values:
            min       25
            median    39
            max       50
    Smoker: 100x1 logical
        Values:
            true     34
            false    66
    BloodPressure: 100x2 double
        Units:  mm Hg
        Description:  Systolic/Diastolic
        Values:
                      BloodPressure_1    BloodPressure_2
                      _______________    _______________
            min       109                  68
            median    122                81.5
            max       138                  99
```

`summary` displays the minimum, median, and maximum values for each column of the variable `BloodPressure`.

Summary of Categorical Vector

Create a 1-by-5 categorical vector.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'})
```

```
A =

     plane      car      train      car      plane
```

A has three categories, `car`, `plane`, and `train`.

Print a summary of A.

```
summary(A)
```

```
     car      plane      train
       2          2          1
```

`car` appears in two elements of A, `plane` appears in two elements, and `train` appears in one element.

Since A is a row vector, `summary` lists the occurrences of each category horizontally.

Summary of Each Column in Categorical Array

Create a 3-by-2 categorical array, A, from a numeric array.

```
X = [1 3; 2 1; 3 1; 4 2];
valueset = 1:3;
catnames = {'red' 'green' 'blue'};

A = categorical(X,valueset,catnames)
```

```
A =

     red              blue
     green            red
     blue             red
     <undefined>      green
```

A has three categories, `red`, `green`, and `blue`. The value, `4`, was not included in the `valueset` input to the `categorical` function. Therefore, the corresponding element, `A(4,1)`, does not have a corresponding category and is undefined.

Print a summary of A.

```
summary(A)
```

```
     red              1      2
     green            1      1
```

```
blue              1       1
<undefined>       1       0
```

`red` appears in one element in the first column of `A` and two in the second column.

`green` appears in one element in the first column of `A` and none in the second column.

`blue` appears in one element in the first column of `A` and one in the second column.

`A` contains only one undefined element. It occurs in the first column.

Category Counts of Each Row in Categorical Array

Create a 3-by-2 categorical array, `A`, from a numeric array.

```
A = categorical([1 3; 2 1; 3 1],1:3,{'red' 'green' 'blue'})
```

```
A =

     red         blue
     green       red
     blue        red
```

`A` has three categories, `red`, `green`, and `blue`.

Print a summary of `A` along the second dimension.

```
summary(A,2)
```

```
     red       green       blue
       1           0          1
       1           1          0
       1           0          1
```

`red` appears in one element in the first row of `A`, one in the second row, and one in the third row.

`green` appears in only one element. It occurs in the second row of `A`.

`blue` appears in one element in the first row of `A` and one in the third column.

Input Arguments

`T` — Input table

table

Input table, specified as a table.

`A` — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

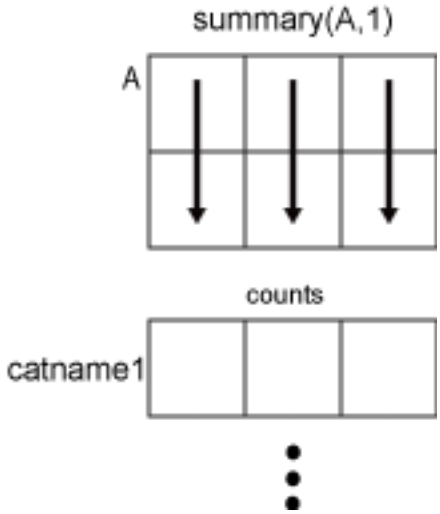
`dim` — Dimension of `A` to operate along

positive integer scalar

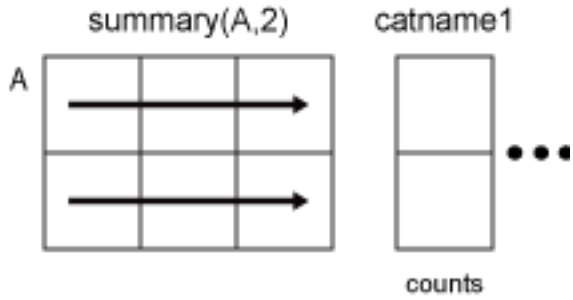
Dimension of `A` to operate to along, specified as a positive integer scalar. If no value is specified, the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional categorical array, `A`:

If `dim = 1`, then `summary(A,1)` displays the category counts for each column of `A`.



If `dim = 2`, then `sumary(A,2)` returns the category counts of each row of `A`.



If `dim` is greater than `ndims(A)`, then`summary(A)` returns an array the same size as `A` for each category. `summary` returns `1` for elements in the corresponding category and `0` otherwise.

More About

Table Summary

The table summary displays the table description from `T.Properties.Description` followed by information on the variables of T.

The summary contains the following information on the variables:

- Name: Size and Data Type — variable name from `T.Properties.VariableNames`, the size of the variable, and the data type of the variable.
- Units — variable's units from `T.Properties.VariableUnits`.
- Description — variable's description from `T.Properties.VariableDescriptions`.
- Values — only included for numeric variables, logical variables, or categorical variables.
  - numeric variables — minimum, median, and maximum values.
  - logical variables — number of values that are `true` and the number of values that are `false`.
  - categorical variables — number of elements from each category.

See Also

`categorical` | `categories` | `countcats` | `table`

superclasses

Superclass names

Syntax

```
superclasses('ClassName')
superclasses(obj)
s = superclasses(...)
```

Description

`superclasses('ClassName')` displays the names of all visible superclasses of the MATLAB class with the name *`ClassName`*. Visible classes have a `Hidden` attribute value of `false` (the default).

`superclasses(obj)` `obj` is an instance of a MATLAB class. `obj` can be either a scalar object or an array of objects.

`s = superclasses(...)` returns the superclass names in a cell array of strings.

Examples

Get the name of the `matlab.mixin.SetGet` class superclass:

```
superclasses('matlab.mixin.SetGet')

Superclasses for class matlab.mixin.SetGet:

    handle
```

See Also

`properties` | `methods` | `events` | `classdef`

superiorto

Establish superior class relationship

Syntax

```
superiorto('class1', 'class2', ...)
```

Description

`superiorto('class1', 'class2', ...)` establishes that the class invoking this function in its constructor has higher precedence than the classes in the argument list.

The `superiorto` function establishes a precedence that determines which object method MATLAB calls. Use this function only from a constructor that calls the `class` function to create an object. For classes defined with `classdef` statements, see "Class Precedence".

Examples

Show function dispatching:

`a` is an object of class `class_a`, `b` is an object of class `class_b`, and `c` is an object of class `class_c`. The constructor method for `class_c` contains the statement `superiorto('class_a')`. Then, either of the following two statements:

```
e = fun(a,c);
e = fun(c,a);
```

invokes `class_c/fun`.

If you call a function with two objects having an unspecified relationship, MATLAB considers the two objects to have equal precedence. In this case, MATLAB calls the left-most object method. So `fun(b,c)` calls `class_b/fun`, while `fun(c,b)` calls `class_c/fun`.

See Also

`inferiorto`

Introduced before R2006a

support

Open MathWorks Technical Support Web page

Note: `support` will be removed in a future release.

Syntax

```
support
```

Description

`support` opens the MathWorks Technical Support Web page, http://www.mathworks.com/support, in a Web browser.

This Web page contains resources including

- A search engine, including an option for solutions to common problems
- Information about installation and licensing
- A patch archive for bug fixes you can download
- Other useful resources

See Also

`doc` | `web`

Introduced before R2006a

Support Package Installer

Install support for third-party hardware or software

Description

Use Support Package Installer to install support packages. The support packages add support for specific third-party hardware or software to specific MathWorks products.

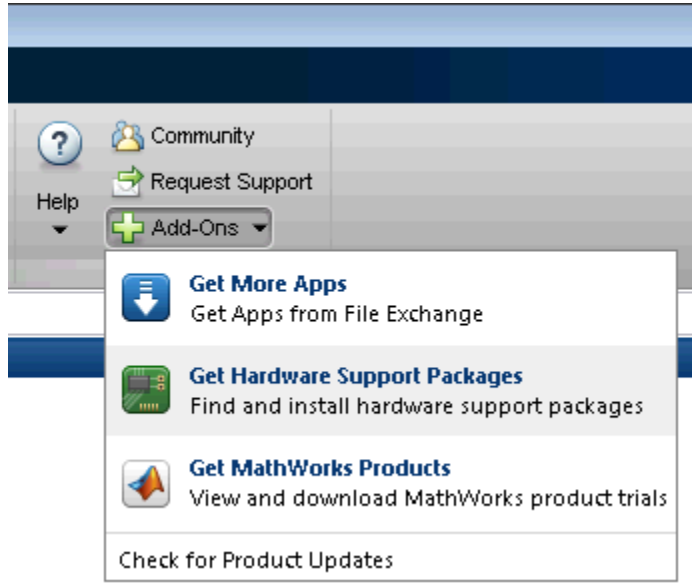
Support Package Installer can:

- Display a list of available, installable, installed, or updatable support packages.
- Install, update, download, or uninstall a support package.
- Update the firmware on specific third-party hardware.

If the support package installs third-party software, Support Package Installer displays a list of the software and licenses for you to review before continuing with the installation.

Open the Support Package Installer App

- On the MATLAB toolstrip, click **Add-Ons > Get Hardware Support Packages**.



- In the MATLAB Command Window, enter `supportPackageInstaller`.
- Double-click a support package installation file (`*.mlpkginstall`).

See Also

Functions

`matlabshared.supportpkg.checkForUpdate` | `matlabshared.supportpkg.getInstalled` | `supportPackageInstaller` | `targetupdater`

supportPackageInstaller

Install support for third-party hardware or software

Syntax

```
supportPackageInstaller
```

Description

The `supportPackageInstaller` function opens *Support Package Installer*.

Use Support Package Installer to install support packages. The support packages add support for specific third-party hardware or software to specific MathWorks products.

Support Package Installer can:

- Display a list of available, installable, installed, or updatable support packages.
- Install, update, download, or uninstall a support package.
- Update the firmware on specific third-party hardware.

If the support package installs third-party software, Support Package Installer displays a list of the software and licenses for you to review before continuing with the installation.

Examples

Use the supportPackageInstaller Function

Enter the function in the MATLAB Command Window:

```
supportPackageInstaller
```

This action starts the Support Package Installer.

More About

- "Support Package Installation"

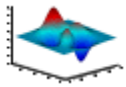
- Hardware Support Catalog

See Also

`matlabshared.supportpkg.checkForUpdate` | `matlabshared.supportpkg.getInstalled` | `targetupdater`

surf

3-D shaded surface plot



Syntax

```
surf(Z)
surf(Z,C)
surf(X,Y,Z)
surf(X,Y,Z,C)
surf(...,'PropertyName',PropertyValue)
surf(axes_handles,...)
h = surf(...)
```

Description

`surf(Z)` creates a three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data, as well as surface height, so color is proportional to surface height.

`surf(Z,C)` plots the height of `Z`, a single-valued function defined over a geometrically rectangular grid, and uses matrix `C`, assumed to be the same size as `Z`, to color the surface. See Coloring Mesh and Surface Plots for information on defining `C`.

`surf(X,Y,Z)` uses `Z` for the color data and surface height. `X` and `Y` are vectors or matrices defining the `x` and `y` components of a surface. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, the vertices of the surface faces are *(X(j), Y(i), Z(i,j))* triples. To create X and Y matrices for arbitrary domains, use the `meshgrid` function.

`surf(X,Y,Z,C)` uses `C` to define color. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surf(...,'PropertyName',PropertyValue)` specifies surface properties along with the data. For a list or properties, see Chart Surface Properties.

`surf(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

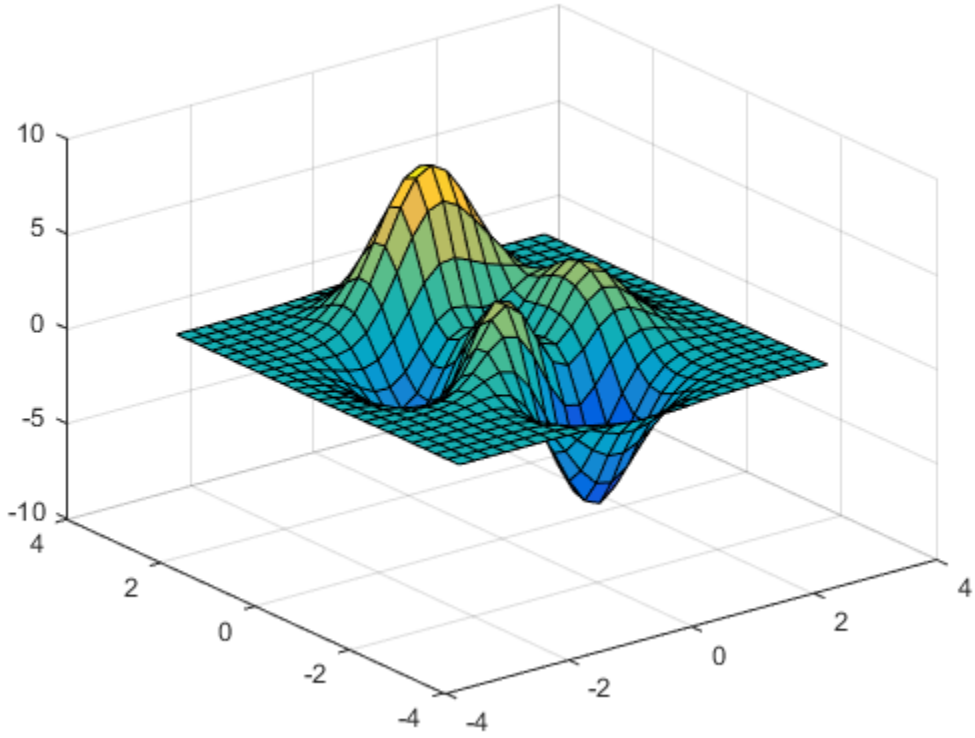
`h = surf(...)` returns a handle to a chart surface graphics object.

Examples

Surface Plot of Peaks Function

Use the `peaks` function to define `X`, `Y`, and `Z` as 25-by-25 matrices. Then, create a surface plot.

```
[X,Y,Z] = peaks(25);

figure
surf(X,Y,Z);
```



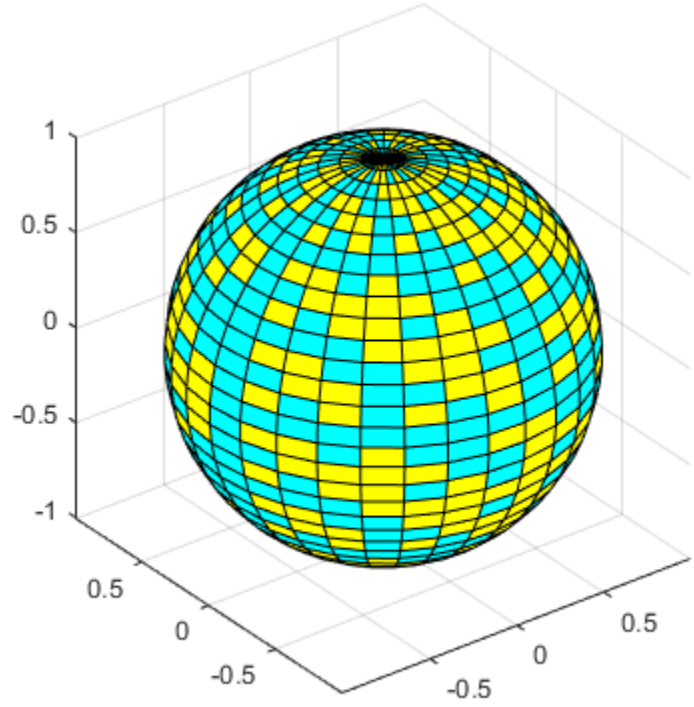
`surf` creates the surface plot from corresponding values in `X`, `Y`, and `Z`. If you do not define the color data `C`, then `surf` uses `Z` to determine the color, so color is proportional to surface height.

Sphere With Two Colors

Create a sphere and color it using the pattern from a Hadamard matrix, which is a matrix that contains the values `1` and `-1`. Change the colors used in the plot by setting the colormap to an array of two RGB triplet values.

```
k = 5;
n = 2^k-1;
[x,y,z] = sphere(n);
c = hadamard(2^k);
```

```
figure
surf(x,y,z,c);
colormap([1  1  0; 0  1  1])
axis equal
```



More About

Tips

`surf` does not accept complex inputs.

Algorithms

Consider a parametric surface parameterized by two independent variables, `i` and `j`, which vary continuously over a rectangle; for example, $1 \le i \le m$ and $1 \le j \le n$. The three functions `x(i,j)`, `y(i,j)`, and `z(i,j)` specify the surface. When `i` and `j` are integer values, they define a rectangular grid with integer grid points. The functions `x(i,j)`, `y(i,j)`, and `z(i,j)` become three `m`-by-`n` matrices, `X`, `Y`, and `Z`. Surface color is a fourth function, `c(i,j)`, denoted by matrix `C`.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
       i-1,j
         |
i,j-1 - i,j - i,j+1
         |
       i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, `[X(:) Y(:) Z(:)]` returns a list of triples specifying points in 3-D space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors. The four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

You can specify surface color in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of `x` and `y`. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The `shading` function sets the shading. If the shading is `interp`, `C` must be the same size as `X`, `Y`, and `Z`; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is `faceted` (the default) or `flat`, `C(i,j)` specifies the constant color in the surface patch:

```
 (i,j)   -   (i,j+1)
   |   C(i,j)  |
(i+1,j)  -  (i+1,j+1)
```

In this case, `C` can be the same size as `X`, `Y`, and `Z` and its last row and column are ignored. Alternatively, its row and column dimensions can be one less than those of `X`, `Y`, and `Z`.

The `surf` function specifies the viewpoint using `view(3)`.

The range of `X`, `Y`, and `Z` or the current setting of `XLimMode`, `YLimMode`, and `ZLimMode` axes properties determines the axis labels. You can also set these properties using `axis` function.

The range of `C` or the current setting of the `CLim` and `CLimMode` axes properties determines the color scaling. You can also set the properties using `caxis` function. The scaled color values are indices into the current colormap.

- "Representing Data as a Surface"
- "Surface Plots of Nonuniformly Sampled Data"
- Coloring Mesh and Surface Plots

See Also

Functions

`axis` | `colormap` | `griddata` | `imagesc` | `mesh` | `meshgrid` | `pcolor` | `shading` | `surfc` | `trisurf` | `view`

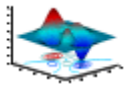
Properties

Chart Surface Properties

Introduced before R2006a

surfc

Contour plot under a 3-D shaded surface plot



Syntax

```
surfc(Z)
surfc(Z,C)
surfc(X,Y,Z)
surfc(X,Y,Z,C)
surfc(...,'PropertyName',PropertyValue)
surfc(axes_handles,...)
h = surfc(...)
```

Description

`surfc(Z)` creates a contour plot under the three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data, as well as surface height, so color is proportional to surface height.

`surfc(Z,C)` plots the height of `Z`, a single-valued function defined over a geometrically rectangular grid, and uses matrix `C`, assumed to be the same size as `Z`, to color the surface.

`surfc(X,Y,Z)` uses `Z` for the color data and surface height. `X` and `Y` are vectors or matrices defining the `x` and `y` components of a surface. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, the vertices of the surface faces are *(X(j), Y(i), Z(i,j))* triples. To create X and Y matrices for arbitrary domains, use the `meshgrid` function.

`surfc(X,Y,Z,C)` uses `C` to define color. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surfc(...,'PropertyName',PropertyValue)` specifies surface properties along with the data.

`surfc(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

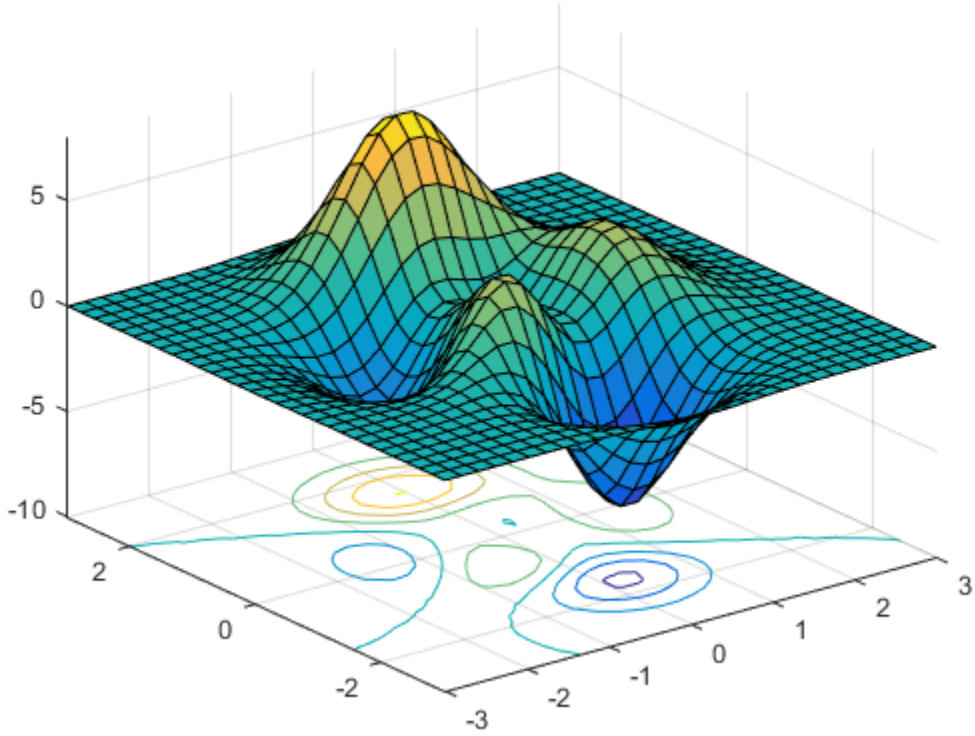
`h = surfc(...)` returns handles to a chart surface and a contour object.

Examples

Display Contour Plot Under Surface Plot

Display a contour plot under a surface plot of the `peaks` function.

```
[X,Y,Z] = peaks(30);
figure
surfc(X,Y,Z)
```



More About

Tips

`surfc` does not accept complex inputs.

Algorithms

Consider a parametric surface parameterized by two independent variables, `i` and `j`, which vary continuously over a rectangle; for example, $1 \le i \le m$ and $1 \le j \le n$. The three functions `x(i,j)`, `y(i,j)`, and `z(i,j)` specify the surface. When `i` and `j` are integer

values, they define a rectangular grid with integer grid points. The functions `x(i,j)`, `y(i,j)`, and `z(i,j)` become three `m`-by-`n` matrices, `X`, `Y`, and `Z`. Surface color is a fourth function, `c(i,j)`, denoted by matrix `C`.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
       i-1,j
         |
i,j-1 - i,j - i,j+1
         |
       i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, `[X(:) Y(:) Z(:)]` returns a list of triples specifying points in 3-D space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors. The four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

You can specify surface color in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of `x` and `y`. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The `shading` function sets the shading. If the shading is `interp`, `C` must be the same size as `X`, `Y`, and `Z`; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is `faceted` (the default) or `flat`, `C(i,j)` specifies the constant color in the surface patch:

```
 (i,j)   -   (i,j+1)
   |   C(i,j)  |
(i+1,j)  -  (i+1,j+1)
```

In this case, `C` can be the same size as `X`, `Y`, and `Z` and its last row and column are ignored. Alternatively, its row and column dimensions can be one less than those of `X`, `Y`, and `Z`.

The `surfc` function specifies the viewpoint using `view(3)`.

The range of `X`, `Y`, and `Z` or the current setting of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties (also set by the `axis` function) determines the axis labels.

The range of `C` or the current setting of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function) determines the color scaling. The scaled color values are used as indices into the current colormap.

- Representing a Matrix as a Surface
- Coloring Mesh and Surface Plots

See Also

Functions

`axis` | `caxis` | `colormap` | `contour` | `delaunay` | `imagesc` | `mesh` | `meshgrid` | `pcolor` | `shading` | `surf` | `trisurf` | `view`

Properties

Chart Surface Properties | Contour Properties

Introduced before R2006a

surf2patch

Convert surface data to patch data

Syntax

```
fvc = surf2patch(Z)
fvc = surf2patch(Z,C)
fvc = surf2patch(X,Y,Z)
fvc = surf2patch(X,Y,Z,C)
fvc = surf2patch(...,'triangles')
[f,v,c] = surf2patch(...)
```

Description

```
fvc = surf2patch(h)
```

converts the geometry and color data from the `surface` object identified by the handle `h` into patch format and returns the face, vertex, and color data in the struct `fvc`. You can pass this struct directly to the `patch` command.

`fvc = surf2patch(Z)` calculates the patch data from the surface's `ZData` matrix `Z`.

`fvc = surf2patch(Z,C)` calculates the patch data from the surface's `ZData` and `CData` matrices `Z` and `C`.

`fvc = surf2patch(X,Y,Z)` calculates the patch data from the surface's `XData`, `YData`, and `ZData` matrices `X`, `Y`, and `Z`.

`fvc = surf2patch(X,Y,Z,C)` calculates the patch data from the surface's `XData`, `YData`, `ZData`, and `CData` matrices `X`, `Y`, `Z`, and `C`.

`fvc = surf2patch(...,'triangles')` creates triangular faces instead of the quadrilaterals that compose surfaces.

`[f,v,c] = surf2patch(...)` returns the face, vertex, and color data in the three arrays `f`, `v`, and `c` instead of a struct.

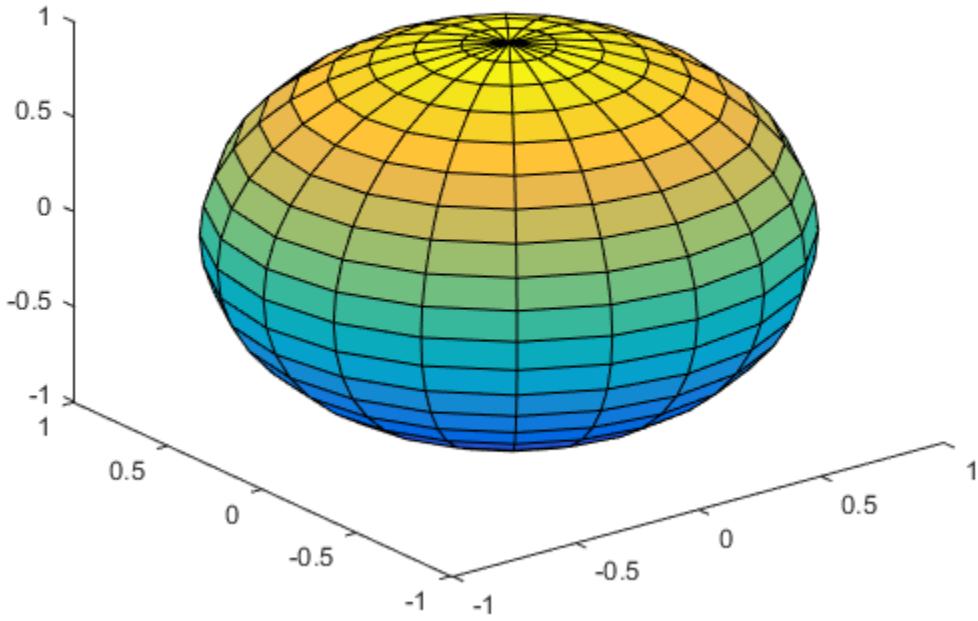
Examples

Calculate Patch Data from Surface Data

Use the `sphere` command to generate the `XData`, `YData`, and `ZData` of a surface. Then, calculate the patch data. Pass the `ZData` (`z`) to `surf2patch` as both the third and fourth arguments - the third argument is the `ZData` and the fourth argument is taken as the `CData`. You must do this since the `patch` command does not automatically use the z-coordinate data for the color data, as does the surface command.

Since patch is a low-level command, you must set the view and shading to produce the same results produced by the `surf` command.

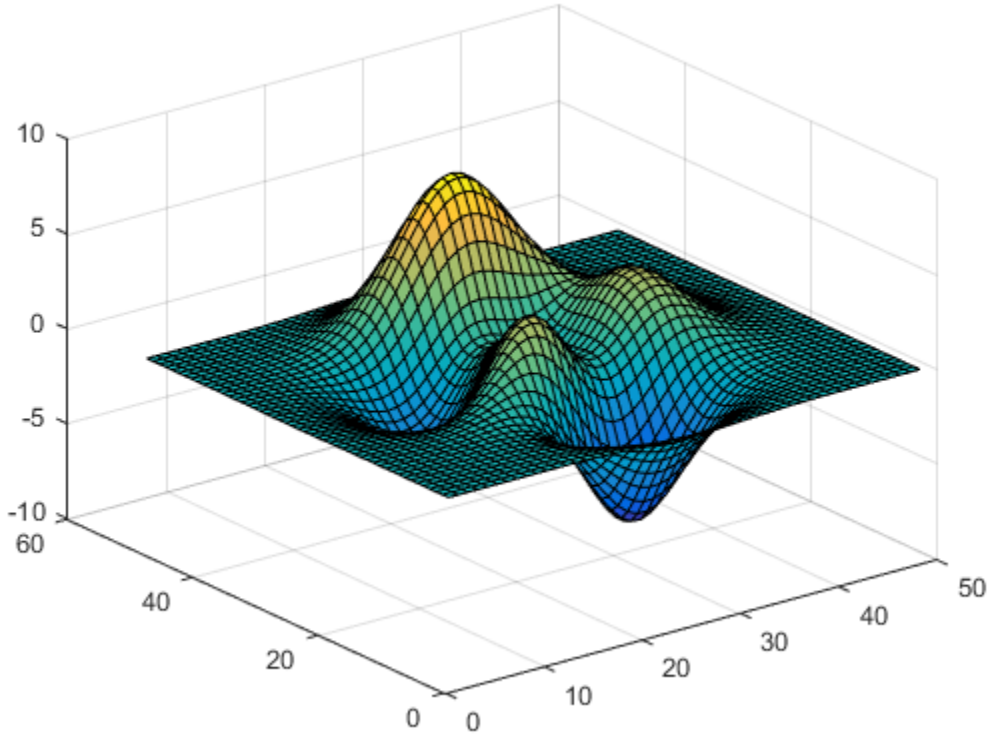
```
[x,y,z] = sphere;
figure
patch(surf2patch(x,y,z,z));
shading faceted;
view(3)
```



Calculate Patch Data Using Surface Object

Calculate face, vertex, and color data from a surface whose handle has been passed as an argument.

```
figure
s = surf(peaks);
patch(surf2patch(s));
delete(s)
shading faceted;
view(3)
```



See Also

`patch` | `reducepatch` | `shrinkfaces` | `surface` | `surf`

Introduced before R2006a

surface

Create surface object

Syntax

```
surface(Z)
surface(Z,C)
surface(X,Y,Z)
surface(X,Y,Z,C)
surface(x,y,Z)
surface(...'PropertyName',PropertyValue,...)
h = surface(...)
```

Properties

For a list of properties, see Primitive Surface Properties.

Description

`surface` is the low-level function for creating surface graphics objects. Surfaces are plots of matrix data created using the row and column indices of each element as the *x*- and *y*-coordinates and the value of each element as the *z*-coordinate.

`surface(Z)` plots the surface specified by the matrix `Z`. Here, `Z` is a single-valued function, defined over a geometrically rectangular grid.

`surface(Z,C)` plots the surface specified by `Z` and colors it according to the data in `C` (see "Examples").

`surface(X,Y,Z)` uses `C = Z`, so color is proportional to surface height above the *x*-*y* plane.

`surface(X,Y,Z,C)` plots the parametric surface specified by `X`, `Y`, and `Z`, with color specified by `C`.

`surface(x,y,Z)`, surface(x,y,Z,C) replaces the first two matrix arguments with vectors and must have `length(x) = n` and `length(y) = m` where `[m,n] = size(Z)`. In this case, the vertices of the surface facets are the triples `(x(j),y(i),Z(i,j))`. Note that `x` corresponds to the columns of `Z` and `y` corresponds to the rows of `Z`. For a complete discussion of parametric surfaces, see the `surf` function.

`surface(...'PropertyName',PropertyValue,...)` follows the `X`, `Y`, `Z`, and `C` arguments with property name/property value pairs to specify additional surface properties. For a description of the properties, see Primitive Surface Properties.

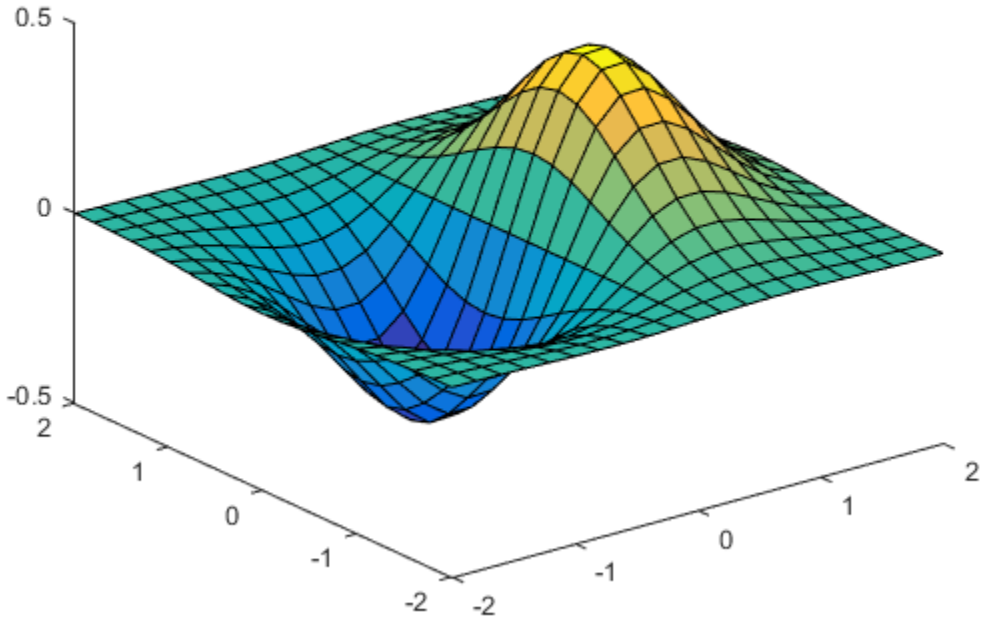
`h = surface(...)` returns a handle to the created primitive surface object.

Examples

Create Surface Plot

Plot the function $z = xe^{-x^2-y^2}$ on the domain $-2 \le x \le 2$ and $-2 \le y \le 2$. Use `meshgrid` to define `X` and `Y`. Then, define `Z` and create a surface plot. Change the view of the plot using `view`.

```
[X,Y] = meshgrid(-2:0.2:2,-2:0.2:2);
Z = X.*exp(-X.^2 - Y.^2);
figure
surface(X,Y,Z)
view(3)
```



`surface` creates the plot from corresponding values in `X`, `Y`, and `Z`. If you do not define the color data `C`, then `surface` uses `Z` to determine the color, so color is proportional to surface height.

Display Image Along Surface Plot

Use the `peaks` function to define `XD`, `YD`, and `ZD` as 25-by-25 matrices.

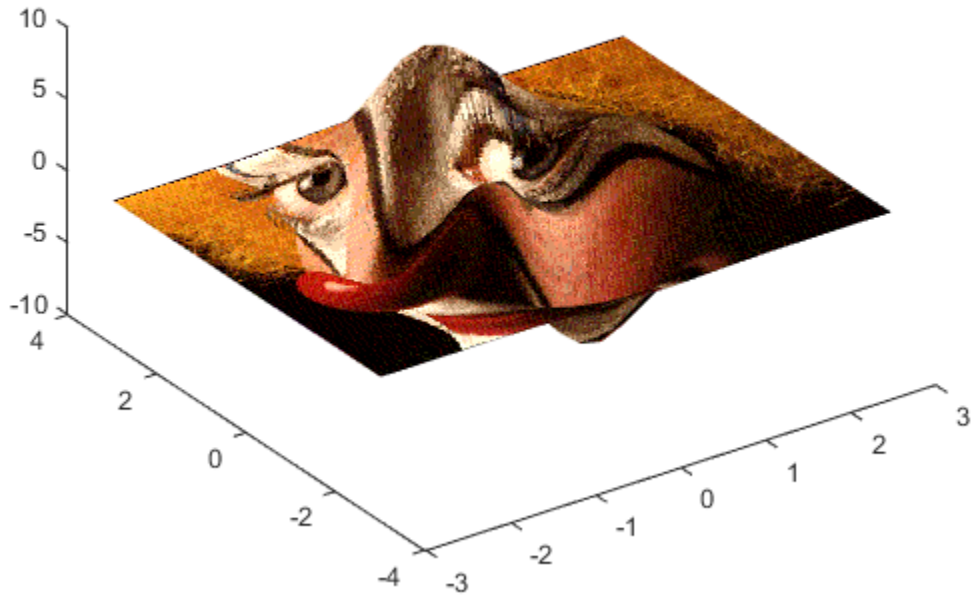
```
[XD,YD,ZD] = peaks(25);
```

Load the `clown` data set to get the image data `X` and its associated colormap, `map`. Flip `X` using the `flipud` function and define the flipped image as the color data for the surface, `C`.

```
load clown
C = flipud(X);
```

Create a surface plot and display the image along the surface. Since the surface data `ZD` and the color data `C` have different dimensions, you must set the surface `FaceColor` to `'texturemap'`.

```
figure
surface(XD,YD,ZD,C,...
    'FaceColor','texturemap',...
    'EdgeColor','none',...
    'CDataMapping','direct')
colormap(map)
view(-35,45)
```



The clown data is typically viewed with the `image` function, which uses `'ij'` axis numbering. This example reverses the image data in the vertical direction using `flipud`.

Tutorials

For examples, see Representing a Matrix as a Surface.

More About

Tips

`surface` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds the surface object to the current axes.

If you do not specify separate color data (`C`), MATLAB uses the matrix (`Z`) to determine the coloring of the surface. In this case, color is proportional to values of `Z`. You can specify a separate matrix to color the surface independently of the data defining the area of the surface.

You can specify properties as property name/property value pairs or using dot notation.

`surface` provides convenience forms that allow you to omit the property name for the `XData`, `YData`, `ZData`, and `CData` properties. For example,

```
surface('XData',X,'YData',Y,'ZData',Z,'CData',C)
```

is equivalent to

```
surface(X,Y,Z,C)
```

When you specify only a single matrix input argument,

```
surface(Z)
```

MATLAB assigns the data properties as if you specified

```
surface('XData',[1:size(Z,2)],...
   'YData',[1:size(Z,1)],...
   'ZData',Z,...
   'CData',Z)
```

The `axis`, `caxis`, `colormap`, `hold`, `shading`, and `view` commands set graphics properties that affect surfaces. You can also set and query surface property values after creating them using dot notation.

See Also

Functions

`ColorSpec` | `patch` | `pcolor` | `surf`

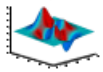
Properties

Primitive Surface Properties

Introduced before R2006a

surfl

Surface plot with colormap-based lighting



Syntax

```
surfl(Z)
surfl(...,'light')
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)
```

Description

The surfl function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surfl(Z)` and `surfl(X,Y,Z)` create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. `X`, `Y`, and `Z` are vectors or matrices that define the *x*, *y*, and *z* components of a surface.

`surfl(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surfl(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surfl(...,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from a surface to a light source. `s = [sx sy sz]` or `s = [azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`surfl(X,Y,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient. `k = [ka kd ks shine]` and defaults to `[.55,.6,.4,10]`.

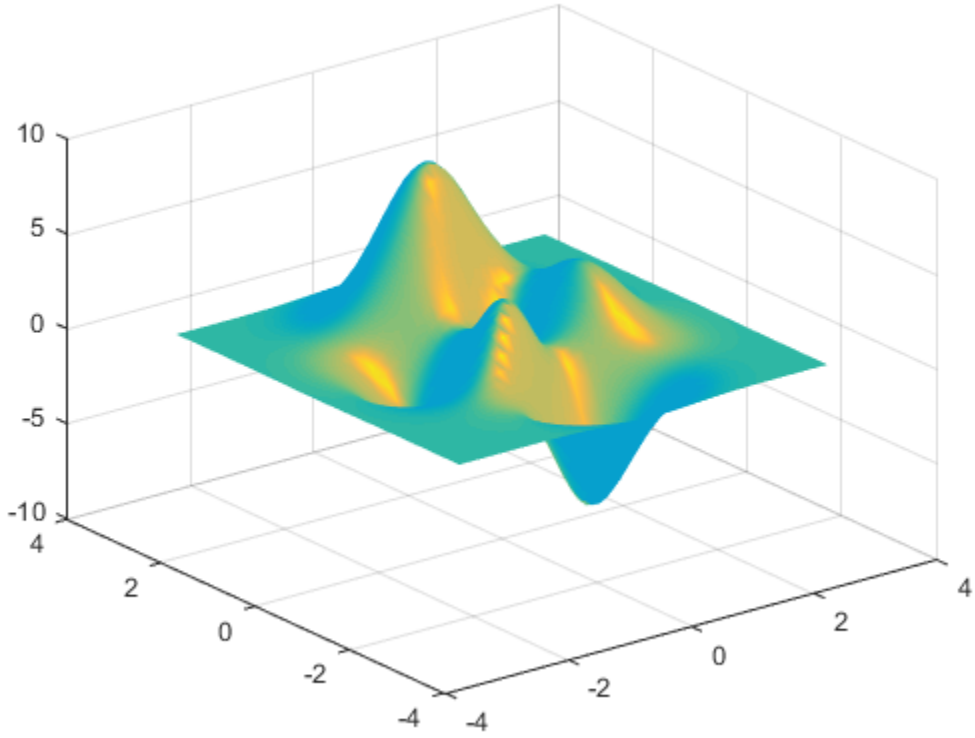
`h = surfl(...)` returns a handle to a surface graphics object. If you specify the `'light'` option, `h` contains the handle to the surface and the light objects.

Examples

Create Surface Plot With Colormap-Based Lighting

Create a surface plot of the `peaks` function using colormap-based lighting. Set the `shading` to `interp` to interpolate the colors across lines and faces.

```
[x,y] = meshgrid(-3:1/8:3);
z = peaks(x,y);
surfl(x,y,z)
shading interp
```



More About

Tips

`surfl` does not accept complex inputs.

For smoother color transitions, use colormaps that have linear intensity variations (e.g., `gray`, `copper`, `bone`, `pink`).

The ordering of points in the `X`, `Y`, and `Z` matrices defines the inside and outside of parametric surfaces. If you want the opposite side of the surface to reflect the light

source, use `surfl(X',Y',Z')`. Because of the way surface normal vectors are computed, `surfl` requires matrices that are at least 3-by-3.

See Also

`colormap` | `light` | `shading`

Introduced before R2006a

surfnorm

Compute and display 3-D surface normals

Syntax

```
surfnorm(Z)
surfnorm(X,Y,Z)
surfnorm(axes_handle, ___ )
surfnorm( ___ ,Name,Value)
[Nx,Ny,Nz] = surfnorm( ___ )
```

Description

`surfnorm(Z)` plots a surface of the matrix Z with `surf` and displays its surface normals as radiating vectors.

`surfnorm(X,Y,Z)` plots a surface and its surface normals from the vectors or matrices X, Y, and matrix Z. `X`, `Y`, and `Z` must be the same size.

`surfnorm(axes_handle, ___ )` plots into `axes_handle` instead of `gca` and it can include any of the input arguments in previous syntaxes.

`surfnorm( ___ ,Name,Value)` can be used to set the value of the specified Chart Surface Properties properties.

`[Nx,Ny,Nz] = surfnorm( ___ )` returns the components of the 3-D surface normals for the surface without plotting the surface or surface normals.

Input Arguments

Z

2–D array of real numbers representing a surface

Default:

`X`

2–D array of real numbers that defines the *x* component of the surface grid

`Y`

2–D array of real numbers that defines the *y* component of the surface grid

`axes_handle`

Handle to the target axes in which to plot the surface

If you do not specify `axes_handle`, MATLAB uses current axes.

`Name,Value`

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Property names and values of the surface object

See Chart Surface Properties for description of property names and values.

Output Arguments

`[Nx,Ny,Nz]`

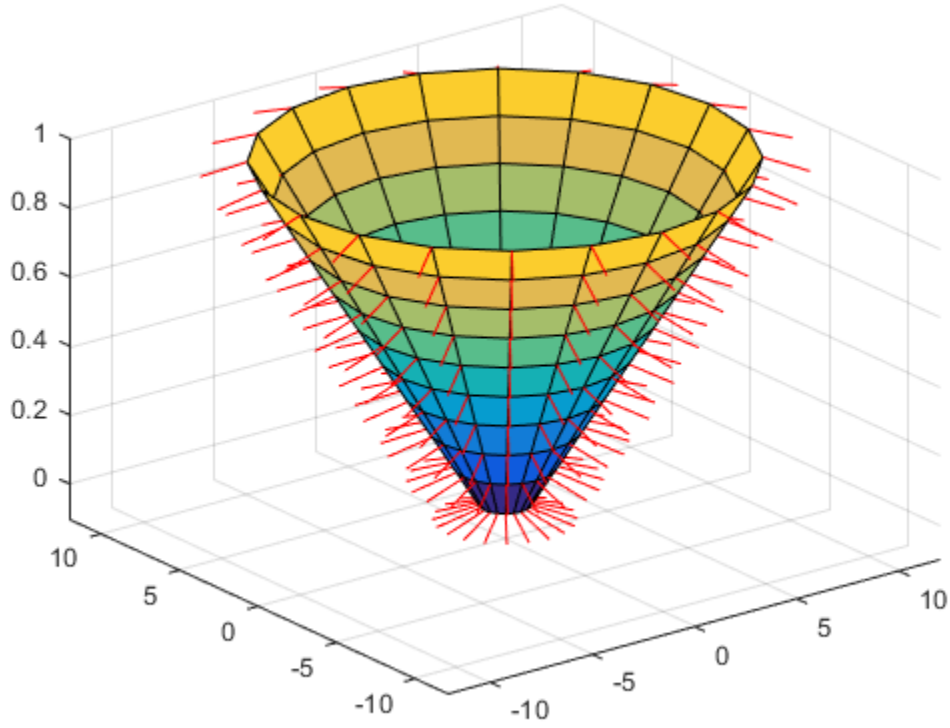
x, *y*, and *z* components of the three-dimensional surface normals for the surface

Examples

Display 3-D Surface Normals for Cone Plot

Compute and plot the normal vectors for a truncated cone. Set the axis limits using the `axis` function.

```
[x,y,z] = cylinder(1:10);
figure
surfnorm(x,y,z)
axis([-12 12 -12 12 -0.1 1])
```



Use Computed Surface Normals for Lighting

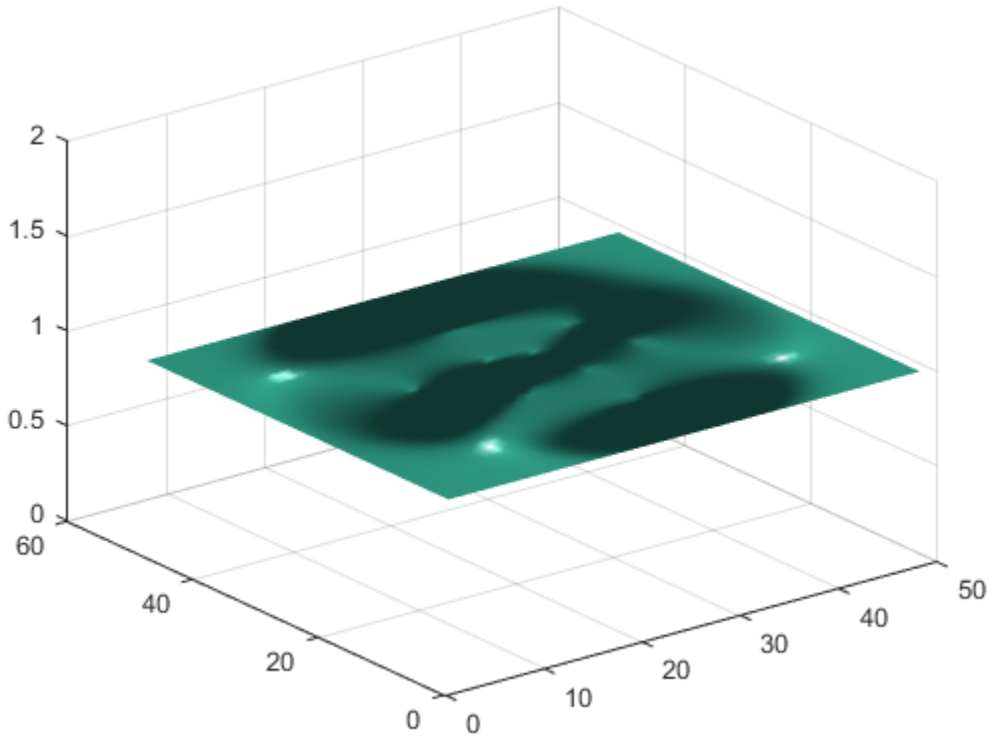
Compute the normal vectors of an expression representing a surface.

```
[nx, ny, nz] = surfnorm(peaks);
```

Assign these normals to the `VertexNormals` property which MATLAB® uses to calculate the surface lighting. Set the lighting algorithm to `gouraud` and add a light using `camlight`.

```
b = reshape([nx ny nz], 49,49,3);
figure
surf(ones(49),'VertexNormals',b,'EdgeColor','none');
lighting gouraud
```

```
camlight
```



More About

Surface Normal

An imaginary line perpendicular to a flat surface or perpendicular to the tangent plane at a point on a non-flat surface

Tips

- `surfnorm` does not accept complex inputs.

- Reverse the direction of the normals by calling `surfnorm` with transposed arguments:

  ```
  surfnorm(X',Y',Z')
  ```
- The surface normals represent conditions at vertices and are not normalized. Normals for surface elements that face away from the viewer do not display.
- `surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.

Algorithms

After performing a bicubic fit of the data in the *x*, *y*, and *z* directions, diagonal vectors are computed and crossed to form the normal at each vertex.

See Also

`surf` | `quiver3` | `surface` | `isonormals` | `surfl`

Introduced before R2006a

svd

Singular value decomposition

Syntax

```
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
[U,S,V] = svd(X,'econ')
```

Description

The `svd` command computes the matrix singular value decomposition.

`s = svd(X)` returns a vector of singular values.

`[U,S,V] = svd(X)` produces a diagonal matrix `S` of the same dimension as `X`, with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` so that `X = U*S*V'`.

`[U,S,V] = svd(X,0)` produces the "economy size" decomposition. If `X` is m-by-n with m > n, then `svd` computes only the first `n` columns of `U` and `S` is n-by-n.

`[U,S,V] = svd(X,'econ')` also produces the "economy size" decomposition. If `X` is m-by-n with m >= n, it is equivalent to `svd(X,0)`. For m < n, only the first m columns of `V` are computed and `S` is m-by-m.

Examples

For the matrix

```
X =
     1     2
     3     4
     5     6
     7     8
```

the statement

```
[U,S,V] = svd(X)
```

produces

```
U =
    -0.1525   -0.8226   -0.3945   -0.3800
    -0.3499   -0.4214    0.2428    0.8007
    -0.5474   -0.0201    0.6979   -0.4614
    -0.7448    0.3812   -0.5462    0.0407

S =
     14.2691         0
           0    0.6268
           0         0
           0         0

V =
    -0.6414    0.7672
    -0.7672   -0.6414
```

The economy size decomposition generated by

```
[U,S,V] = svd(X,0)
```

produces

```
U =
    -0.1525   -0.8226
    -0.3499   -0.4214
    -0.5474   -0.0201
    -0.7448    0.3812
S =
    14.2691         0
          0    0.6268
V =
    -0.6414    0.7672
    -0.7672   -0.6414
```

Diagnostics

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

```
Solution will not converge.
```

Introduced before R2006a

svds

Find singular values and vectors

Syntax

```
s = svds(A)
s = svds(A,k)
s = svds(A,k,sigma)
s = svds(A,k,'L')
s = svds(A,k,sigma,options)
[U,S,V] = svds(A,...)
[U,S,V,flag] = svds(A,...)
```

Description

`s = svds(A)` computes the six largest singular values and associated singular vectors of matrix `A`. If `A` is `m`-by-`n`, `svds(A)` manipulates eigenvalues and vectors returned by `eigs(B)`, where `B = [sparse(m,m) A; A' sparse(n,n)]`, to find a few singular values and vectors of `A`. The positive eigenvalues of the symmetric matrix `B` are the same as the singular values of `A`.

`s = svds(A,k)` computes the `k` largest singular values and associated singular vectors of matrix `A`.

`s = svds(A,k,sigma)` computes the `k` singular values closest to the scalar shift sigma. For example, `s = svds(A,k,0)` computes the `k` smallest singular values and associated singular vectors.

`s = svds(A,k,'L')` computes the `k` largest singular values (the default).

`s = svds(A,k,sigma,options)` sets some parameters (see `eigs`):

Option Structure Fields and Descriptions

Field name	Parameter	Default
`options.tol`	Convergence tolerance: `norm(AV-US,1)<=tol*norm(A,1)`	`1e-10`

Field name	Parameter	Default
`options.maxit`	Maximum number of iterations	300
`options.disp`	Number of values displayed each iteration	0

`svds` checks the accuracy of the computed singular vectors. If the vectors are not accurate enough, then `svds` returns fewer singular values than requested. To obtain the requested number of singular values, try decreasing the error tolerance in the options structure.

`[U,S,V] = svds(A,...)` returns three output arguments, and if `A` is `m`-by-`n`:

- `U` is `m`-by-`k` with orthonormal columns
- `S` is `k`-by-`k` diagonal
- `V` is `n`-by-`k` with orthonormal columns
- `U*S*V'` is the closest rank `k` approximation to `A`

`[U,S,V,flag] = svds(A,...)` returns a convergence flag. If `eigs` converged, then `norm(A*V-U*S,1) <= tol*norm(A,1)` and `flag` is `0`. If `eigs` did not converge, then `flag` is `1`.

Note `svds` is best used to find a few singular values of a large, sparse matrix. To find all the singular values of such a matrix, `svd(full(A))` will usually perform better than `svds(A,min(size(A)))`.

Examples

Singular Values of Sparse Matrix

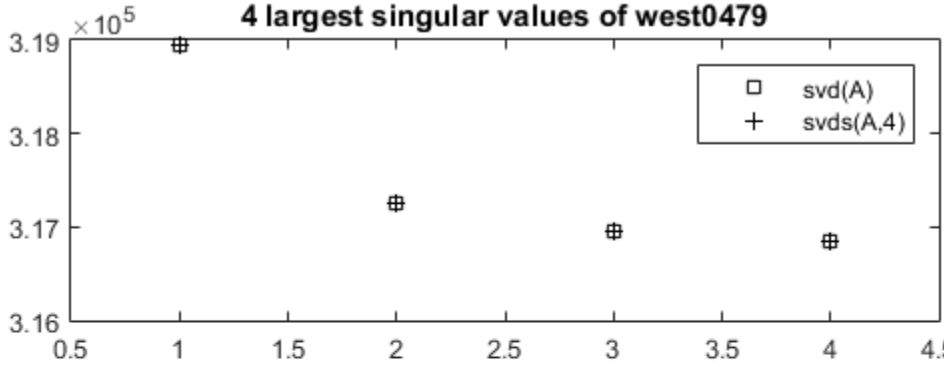
`west0479` is a real 479-by-479 sparse matrix. `svd` calculates all 479 singular values. `svds` picks out only the largest and smallest singular values.

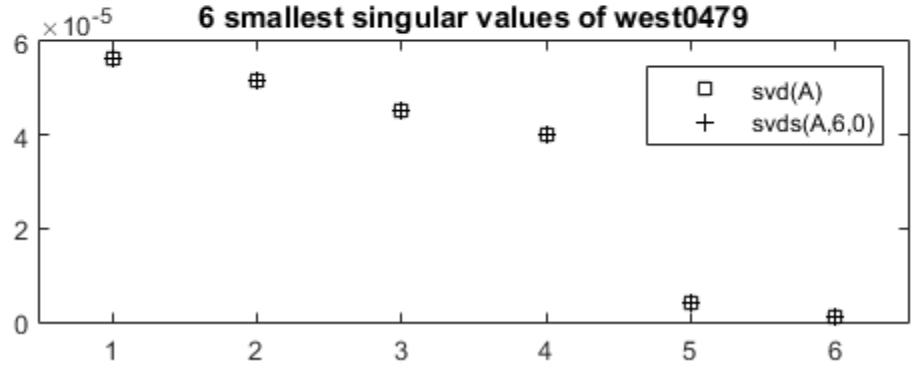
```
load west0479
s = svd(full(west0479));
sl = svds(west0479,4);
ss = svds(west0479,6,0);
```

```
Warning: NORMEST did not converge for 100 iterations with tolerance 1e-06
```

These plots show some of the singular values of west0479 as computed by `svd` and `svds`.

```
subplot(2,1,1)
plot(s(1:4),'ks')
hold on
plot(sl,'k+')
hold off
title('4 largest singular values of west0479')
legend('svd(A)','svds(A,4)')
xlim([0.5 4.5])

subplot(2,1,2)
plot(s(end-5:end),'ks')
hold on
plot(ss,'k+')
hold off
title('6 smallest singular values of west0479')
legend('svd(A)','svds(A,6,0)')
xlim([0.5 6.5])
```





The largest singular value of `west0479` can be computed a few different ways:

```
svds(west0479,1)
max(svd(full(west0479)))
norm(full(west0479))
```

```
ans =

   3.1895e+05


ans =
```

```
   3.1895e+05

ans =

   3.1895e+05
```

Or, to estimate the largest singular value:

```
normest(west0479)

Warning: NORMEST did not converge for 100 iterations with tolerance 1e-06

ans =

   3.1854e+05
```

More About

Algorithms

`svds(A,k)` uses `eigs` to find the `k` largest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`.

`svds(A,k,0)` uses `eigs` to find the `2k` smallest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`, and then selects the `k` positive eigenvalues and their eigenvectors.

See Also

`svd` | `eigs`

Introduced before R2006a

swapbytes

Swap byte ordering

Syntax

```
Y = swapbytes(X)
```

Description

`Y = swapbytes(X)` reverses the byte ordering of each element in array `X`, converting little-endian values to big-endian (and vice versa). The input array must contain all full, noncomplex, numeric elements.

Examples

Example 1

Reverse the byte order for a scalar 32-bit value, changing hexadecimal `12345678` to `78563412`:

```
A = uint32(hex2dec('12345678'));

B = dec2hex(swapbytes(A))
B =
    78563412
```

Example 2

Reverse the byte order for each element of a 1-by-4 matrix:

```
X = uint16([0 1 128 65535])
X =
      0      1    128  65535
```

```
Y = swapbytes(X);
Y =
      0    256  32768  65535
```

Examining the output in hexadecimal notation shows the byte swapping:

```
format hex

X, Y
X =
   0000   0001   0080   ffff
Y =
   0000   0100   8000   ffff
```

Example 3

Create a three-dimensional array A of 16-bit integers and then swap the bytes of each element:

```
format hex

A = uint16(magic(3) * 150);
A(:,:,2) = A * 40;

A
A(:,:,1) =
   04b0   0096   0384
   01c2   02ee   041a
   0258   0546   012c
A(:,:,2) =
   bb80   1770   8ca0
   4650   7530   a410
   5dc0   d2f0   2ee0

swapbytes(A)
ans(:,:,1) =
   b004   9600   8403
   c201   ee02   1a04
   5802   4605   2c01
ans(:,:,2) =
   80bb   7017   a08c
   5046   3075   10a4
   c05d   f0d2   e02e
```

See Also

`typecast`

Introduced before R2006a

switch, case, otherwise

Execute one of several groups of statements

Syntax

```
switch switch_expression
   case case_expression
      statements
   case case_expression
      statements
    ...
   otherwise
      statements
end
```

Description

`switch` *`switch_expression`*`, case` *`case_expression`*`, end` evaluates an expression and chooses to execute one of several groups of statements. Each choice is a case.

The `switch` block tests each case until one of the case expressions is true. A case is true when:

- For numbers, *`case_expression`* `==` *`switch_expression`*.
- For strings, `strcmp(`*`case_expression`*`,`*`switch_expression`*`) == 1`.
- For objects that support the `eq` function, *`case_expression`* `==` *`switch_expression`*.
- For a cell array *`case_expression`*, at least one of the elements of the cell array matches *`switch_expression`*, as defined above for numbers, strings, and objects.

When a case expression is true, MATLAB executes the corresponding statements and exits the `switch` block.

An evaluated *`switch_expression`* must be a scalar or string. An evaluated *`case_expression`* must be a scalar, a string, or a cell array of scalars or strings.

The `otherwise` block is optional. MATLAB executes the statements only when no case is true.

Examples

Compare Single Values

Display different text conditionally, depending on a value entered at the command prompt.

```
n = input('Enter a number: ');

switch n
    case -1
        disp('negative one')
    case 0
        disp('zero')
    case 1
        disp('positive one')
    otherwise
        disp('other value')
end
```

At the command prompt, enter the number 1.

```
positive one
```

Repeat the code and enter the number 3.

```
other value
```

Compare Against Multiple Values

Determine which type of plot to create based on the value of the string `plottype`. If `plottype` is either `'pie'` or `'pie3'`, create a 3-D pie chart. Use a cell array to contain both values.

```
x = [12 64 24];
plottype = 'pie3';

switch plottype
    case 'bar'
```

```
        bar(x)
        title('Bar Graph')
    case {'pie','pie3'}
        pie3(x)
        title('Pie Chart')
    otherwise
        warning('Unexpected plot type. No plot created.')
end
```

More About

Tips

- A *`case_expression`* cannot include relational operators such as < or > for comparison against the *`switch_expression`*. To test for inequality, use `if`, `elseif`, `else` statements.
- The MATLAB `switch` statement does not fall through like a C language `switch` statement. If the first `case` statement is `true`, MATLAB does not execute the other `case` statements. For example:

  ```
  result = 52;

  switch(result)
     case 52
        disp('result is 52')
     case {52, 78}
        disp('result is 52 or 78')
  end
  ```

  ```
  result is 52
  ```

- Define all variables necessary for code in a particular case within that case. Since MATLAB executes only one case of any `switch` statement, variables defined within one case are not available for other cases. For example, if your current workspace does not contain a variable `x`, only cases that define `x` can use it:

  ```
  switch choice
     case 1
        x = -pi:0.01:pi;
     case 2
        % does not know anything about x
  end
  ```

- Do not use a `break` statement within a `switch` block. `break` is not defined outside a `for` or `while` loop.

See Also

`end` | `for` | `if` | `while`

Introduced before R2006a

sylvester

Solve Sylvester equation $AX + XB = C$ for X

Syntax

```
X = sylvester(A,B,C)
```

Description

`X = sylvester(A,B,C)` returns the solution, `X`, to the Sylvester equation.

Input `A` is an m-by-m matrix, input `B` is an n-by-n matrix, and both `C` and `X` are m-by-n matrices.

Examples

Solve Sylvester Equation with 3-by-3 Output

Create the coefficient matrices `A` and `B`.

```
A = [1 -1 1; 1 1 -1; 1 1 1];
B = magic(3);
```

Define `C` as the 3-by-3 identity matrix.

```
C = eye(3);
```

Use the `sylvester` function to solve the Sylvester equation for these values of `A`, `B`, and `C`.

```
X = sylvester(A,B,C)
```

```
X =

    0.1223   -0.0725    0.0131
   -0.0806   -0.0161    0.1587
   -0.0164    0.1784   -0.1072
```

The result is a 3-by-3 matrix.

Solve Sylvester Equation with 4-by-2 Output

Create a 4-by-4 coefficient matrix, `A`, and 2-by-2 coefficient matrix, `B`.

```
A = [1 0 2 3; 4 1 0 2; 0 5 5 6; 1 7 9 0];
B = [0 -1; 1 0];
```

Define `C` as a 4-by-2 matrix to match the corresponding sizes of `A` and `B`.

```
C = [1 0; 2 0; 0 3; 1 1]
```

```
C =

     1     0
     2     0
     0     3
     1     1
```

Use the `sylvester` function to solve the Sylvester equation for these values of `A`, `B`, and `C`.

```
X = sylvester(A,B,C)
```

```
X =

    0.4732   -0.3664
   -0.4006    0.3531
    0.3305   -0.1142
    0.0774    0.3560
```

The result is a 4-by-2 matrix.

Input Arguments

`A,B,C` — Input matrices

matrices

Input matrices, specified as matrices. Input `A` is an m-by-m square matrix, input `B` is an n-by-n square matrix, and input `C` is an m-by-n rectangular matrix. The function returns an error if any input matrix is sparse.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`X` — Solution

matrix

Solution, returned as a matrix of the same size as `C`. The function returns an error if the eigenvalues of `A` and `-B` are not distinct (in this case, the solution, `X`, is singular or not unique).

More About

Sylvester Equation

The Sylvester equation is

$$AX + XB = C.$$

The equation has a unique solution when the eigenvalues of `A` and `-B` are distinct. In terms of the Kronecker tensor product, $\otimes$, the equation is

$$\left[I \otimes A + B^T \otimes I\right] X(:) = C(:),$$

where `I` is the identity matrix, and `X(:)` and `C(:)` denote the matrices `X` and `C` as single column vectors.

See Also

`ctranspose` | `eig` | `kron` | `mldivide` | `mtimes`

Introduced in R2014a

symamd

Symmetric approximate minimum degree permutation

Syntax

```
p = symamd(S)
p = symamd(S,knobs)
[p,stats] = symamd(...)
```

Description

`p = symamd(S)` for a symmetric positive definite matrix `S`, returns the permutation vector `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. To find the ordering for `S`, `symamd` constructs a matrix `M` such that `spones(M'*M) = spones (S)`, and then computes `p = colamd(M)`. The `symamd` function may also work well for symmetric indefinite matrices.

`S` must be square; only the strictly lower triangular part is referenced.

`p = symamd(S,knobs)` where `knobs` is a scalar. If `S` is `n`-by-`n`, rows and columns with more than `knobs*n` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs = spparms('wh_frac')`.

`[p,stats] = symamd(...)` produces the optional vector `stats` that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `symamd`
`stats(2)`	Number of dense or empty columns ignored by `symamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `symamd` (roughly of size `8.4*nnz(tril(S,-1)) + 9n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid

`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists
`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `symamd`. For this reason, `symamd` verifies that `S` is valid:

- If a row index appears two or more times in the same column, `symamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `symamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `symamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`).

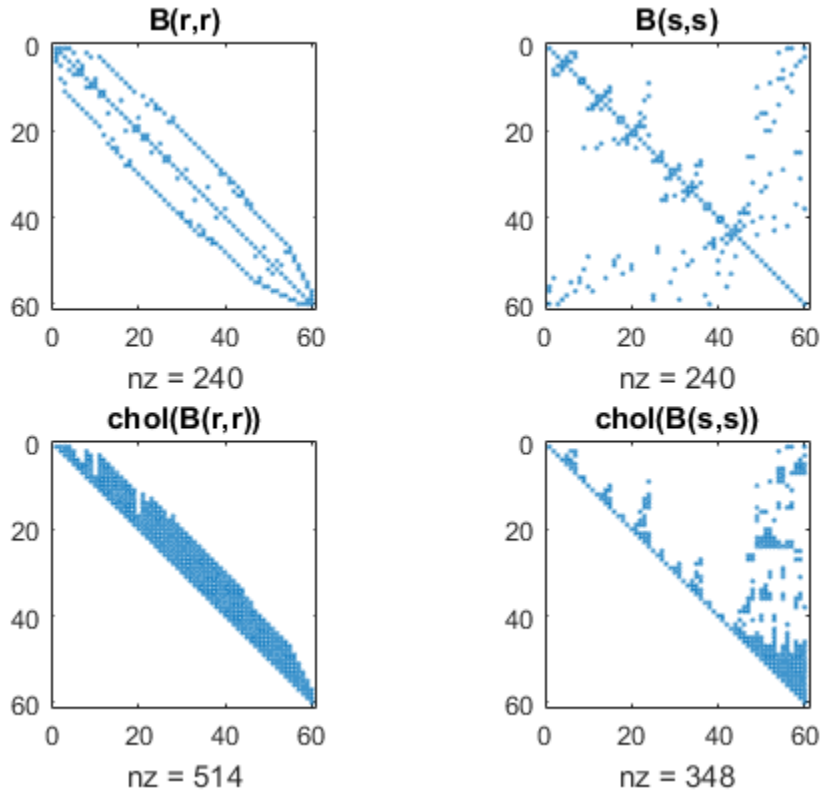
The ordering is followed by a symmetric elimination tree post-ordering.

Examples

Compare Reverse Cuthill-McKee and Minimum Degree

Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the `symrcm` reference page.

```
B = bucky+4*speye(60);
r = symrcm(B);
p = symamd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R,4), title('B(r,r)')
subplot(2,2,2), spy(S,4), title('B(s,s)')
subplot(2,2,3), spy(chol(R),4), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S),4), title('chol(B(s,s))')
```



Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

References

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (`davis@cise.ufl.edu`), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National

Laboratory. Sparse Matrix Algorithms Research at the University of Florida: http://www.cise.ufl.edu/research/sparse/

See Also

`colamd` | `colperm` | `symrcm` | `amd` | `spparms`

Introduced before R2006a

symbfact

Symbolic factorization analysis

Syntax

```
count = symbfact(A)
count = symbfact(A,'sym')
count = symbfact(A,'col')
count = symbfact(A,'row')
count = symbfact(A,'lo')
[count,h,parent,post,R] = symbfact(...)
[count,h,parent,post,L] = symbfact(A,type,'lower')
```

Description

`count = symbfact(A)` returns the vector of row counts of `R=chol(A)`. `symbfact` should be much faster than `chol(A)`.

`count = symbfact(A,'sym')` is the same as `count = symbfact(A)`.

`count = symbfact(A,'col')` returns row counts of `R=chol(A'*A)` (without forming it explicitly).

`count = symbfact(A,'row')` returns row counts of `R=chol(A*A')`.

`count = symbfact(A,'lo')` is the same as `count = symbfact(A)` and uses `tril(A)`.

`[count,h,parent,post,R] = symbfact(...)` has several optional return values.

The flop count for a subsequent Cholesky factorization is `sum(count.^2)`

Return Value	Description
`h`	Height of the elimination tree
`parent`	The elimination tree itself
`post`	Postordering of the elimination tree

Return Value	Description
`R`	0-1 matrix having the structure of `chol(A)` for the symmetric case, `chol(A'*A)` for the `'col'` case, or `chol(A*A')` for the `'row'` case.

`symbfact(A)` and `symbfact(A,'sym')` use the upper triangular part of `A (triu(A))` and assume the lower triangular part is the transpose of the upper triangular part. `symbfact(A,'lo')` uses `tril(A)` instead.

`[count,h,parent,post,L] = symbfact(A,type,'lower')` where `type` is one of `'sym'`,`'col'`, `'row'`, or`'lo'` returns a lower triangular symbolic factor `L=R'`. This form is quicker and requires less memory.

See Also

`chol` | `etree` | `treelayout`

Introduced before R2006a

symmlq

Symmetric LQ method

Syntax

```
x = symmlq(A,b)
symmlq(A,b,tol)
symmlq(A,b,tol,maxit)
symmlq(A,b,tol,maxit,M)
symmlq(A,b,tol,maxit,M1,M2)
symmlq(A,b,tol,maxit,M1,M2,x0)
[x,flag] = symmlq(A,b,...)
[x,flag,relres] = symmlq(A,b,...)
[x,flag,relres,iter] = symmlq(A,b,...)
[x,flag,relres,iter,resvec] = symmlq(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)
```

Description

`x = symmlq(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should also be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `symmlq` converges, a message to that effect is displayed. If `symmlq` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`symmlq(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `symmlq` uses the default, `1e-6`.

`symmlq(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `symmlq` uses the default, `min(n,20)`.

`symmlq(A,b,tol,maxit,M)` and `symmlq(A,b,tol,maxit,M1,M2)` use the symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = in(sqrt(M))*y`. If `M` is `[]` then `symmlq` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`symmlq(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `symmlq` uses the default, an all-zero vector.

`[x,flag] = symmlq(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`symmlq` converged to the desired tolerance `tol` within `maxit` iterations.
1	`symmlq` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`symmlq` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `symmlq` became too small or too large to continue computing.
5	Preconditioner `M` was not symmetric positive definite.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = symmlq(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = symmlq(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = symmlq(A,b,...)` also returns a vector of estimates of the `symmlq` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)` also returns a vector of estimates of the conjugate gradients residual norms at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50; M1 = spdiags(4*on,0,n,n);

x = symmlq(A,b,tol,maxit,M1);
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in the function `run_symmlq` that:

- Calls `symmlq` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_symmlq` are available to `afun`.

The following shows the code for `run_symmlq`:

```
function x1 = run_symmlq
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = symmlq(@afun,b,tol,maxit,M1);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
       end
```

```
end
```

When you enter

```
x1=run_symmlq;
```

MATLAB software displays the message

```
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 3

Use a symmetric indefinite matrix that fails with `pcg`.

```
A = diag([20:-1:1,-1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, `symmlq` can handle the indefinite matrix `A`.

```
x = symmlq(A,b,1e-6,40);
symmlq converged at iteration 39 to a solution with relative
residual 1.3e-007
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

See Also

`bicg` | `bicgstab` | `cgs` | `function_handle` | `gmres` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr`

Introduced before R2006a

symrcm

Sparse reverse Cuthill-McKee ordering

Syntax

```
r = symrcm(S)
```

Description

`r = symrcm(S)` returns the symmetric reverse Cuthill-McKee ordering of `S`. This is a permutation `r` such that `S(r,r)` tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric `S`.

For a real, symmetric sparse matrix, `S`, the eigenvalues of `S(r,r)` are the same as those of `S`, but `eig(S(r,r))` probably takes less time to compute than `eig(S)`.

Examples

Reverse Cuthill-McKee Ordering

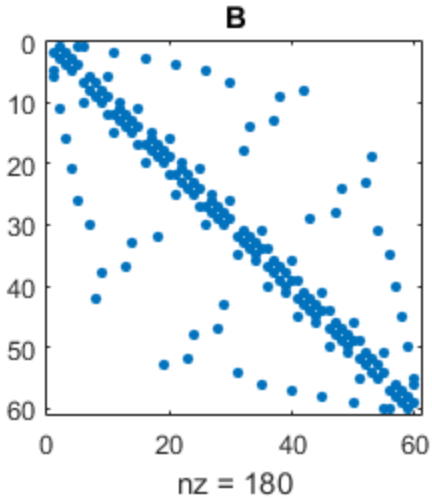
The statement

```
B = bucky;
```

uses a function in the `demos` toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name `bucky`), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together.

With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows:

```
figure();
subplot(1,2,1),spy(B),title('B')
```

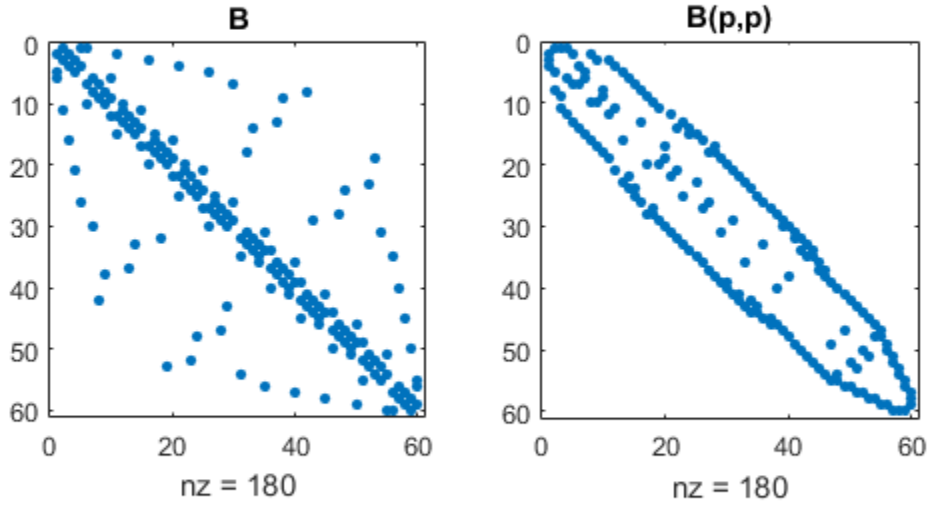


The reverse Cuthill-McKee ordering is obtained with:

```
p = symrcm(B);
R = B(p,p);
```

The `spy` plot shows a much narrower bandwidth.

```
subplot(1,2,2),spy(R),title('B(p,p)')
```



This example is continued in the reference page for `symamd`.

The bandwidth can also be computed with:

```
[i,j] = find(B);
bw = max(i-j) + 1;
```

The bandwidths of `B` and `R` are 35 and 12, respectively.

More About

Algorithms

The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

References

[1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

See Also

`colamd` | `colperm` | `symamd`

Introduced before R2006a

symvar

Determine symbolic variables in expression

Syntax

```
symvar 'expr'
s = symvar('expr')
```

Description

`symvar 'expr'` searches the expression, `expr`, for identifiers other than `i`, `j`, `pi`, `inf`, `nan`, `eps`, and common functions. `symvar` displays those variables that it finds or, if no such variable exists, displays an empty cell array, `{}`.

`s = symvar('expr')` returns the variables in a cell array of strings, `s`. If no such variable exists, `s` is an empty cell array.

Examples

`symvar` finds variables `beta1` and `x`, but skips `pi` and the `cos` function.

```
symvar 'cos(pi*x - beta1)'

ans =

    'beta1'
    'x'
```

See Also

`strfind`

Introduced before R2006a

syntax

Two ways to call MATLAB functions

Description

You can call MATLAB functions using either *command syntax* or *function syntax*, as described below.

Command Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname arg1 arg2 ... argn
```

Command syntax does not allow you to obtain any values that might be returned by the function. Attempting to assign output from the function to a variable using command syntax generates an error. Use function syntax instead.

Examples of command syntax:

```
save mydata.mat x y z
import java.awt.Button java.lang.String
```

Arguments are treated as string literals. See the examples below, under “Argument Passing” on page 1-7974.

Function Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by commas and enclosed in parentheses:

```
functionname(arg1, arg2, ..., argn)
```

You can assign the output of the function to one or more output values. When assigning to more than one output variable, separate the variables by commas or spaces and enclose them in square brackets (`[ ]`):

```
[out1,out2,...,outn] = functionname(arg1, arg2, ..., argn)
```

Examples of function syntax:

```
copyfile('srcfile', '..\mytests', 'writable')
[x1,x2,x3,x4] = deal(A{:})
```

Arguments are passed to the function by value. See the examples below, under "Argument Passing" on page 1-7974.

Argument Passing

When calling a function using command syntax, MATLAB passes the arguments as string literals. When using function syntax, arguments are passed by value.

In the following example, assign a value to `A` and then call `disp` on the variable to display the value passed. Calling `disp` with command syntax passes the variable name, `'A'`:

```
A = pi;
disp A
   A
```

while function syntax passes the value assigned to `A`:

```
A = pi;
disp(A)
   3.1416
```

The next example passes two strings to `strcmp` for comparison. Calling the function with command syntax compares the variable names, `'str1'` and `'str2'`:

```
str1 = 'one';    str2 = 'one';
strcmp str1 str2
ans =
     0        (unequal)
```

while function syntax compares the values assigned to the variables, `'one'` and `'one'`:

```
str1 = 'one';    str2 = 'one';
strcmp(str1, str2)
ans =
     1        (equal)
```

Passing Strings

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotes, (`'string'`). For example, to create a new folder called `myapptests`, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```
folder = 'myapptests';
mkdir(folder)
```

More About

- "Check Code for Errors and Warnings"

See Also

`mlint`

system

Execute operating system command and return output

Syntax

```
status = system(command)
[status,cmdout] = system(command)
[status,cmdout] = system(command,'-echo')
```

Description

`status = system(command)` calls the operating system to execute the specified command. The operation waits for the command to finish execution before returning the exit status of the command to the `status` variable.

`[status,cmdout] = system(command)` additionally returns the output of the command to `cmdout`. This syntax is most useful for commands that do not require user input, such as `dir`.

`[status,cmdout] = system(command,'-echo')` additionally displays (echoes) the command output in the MATLAB Command Window. This syntax is most useful for commands that require user input and that run correctly in the MATLAB Command Window.

Examples

Display Windows Operating System Command Status and Output

On a Windows system, display the current folder using the `cd` command.

```
command = 'cd';
[status,cmdout] = system(command)
```

```
status =
     0
cmdout =
C:\matlab\myfiles
```

A `status` of zero indicates that the command completed successfully. MATLAB returns a string containing the current folder in `cmdout`.

Save UNIX Command Exit Status and Output

List all users who are currently logged in, and save the command exit status and output. Then, view the status.

```
command = 'who';
[status,cmdout] = system(command);
status
```

```
status =

     0
```

A `status` of zero indicates that the command completed successfully. MATLAB returns a string containing the list of users in `cmdout`.

Input Arguments

`command` — Operating system command

string

Operating system command, specified as a string. The `command` executes in a system shell, which might not be the shell from which you launched MATLAB.

Example: `'dir'`

Example: `'ls'`

Output Arguments

`status` — Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (`&`), then `status` is the exit status upon command launch.

- If `command` does not include the ampersand character (`&`), then `status` is the exit status upon command completion.

`cmdout` — Output of operating system command
string

Output of the operating system command, returned as a string.

Limitations

- DOS does not support UNC path names. Therefore, if the current folder uses a UNC path name, then running `system` with a DOS `command` that relies on the current folder fails. To work around this limitation, change the folder to a mapped drive before calling `system`.

More About

Tips

- To execute the operating system command in the background, include the trailing character, `&`, in the `command` argument (for example, `'notepad &'` on a Windows platform, or `'emacs &'` on UNIX). The exit status is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

  Note: If `command` includes the trailing `&` character, `cmdout` is empty.

- On a UNIX system, the `system` function redirects `stdin` to the invoked command, `command`, by default. This redirection also forwards MATLAB script commands and the keyboard type-ahead buffer to the invoked command while the `system` function executes. This can lead to corrupted output when `system` does not complete execution immediately. To disable `stdin` and type-ahead redirection, include the formatted string `< /dev/null` in the call to the invoked command.

Algorithms

On UNIX, MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB

first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.

- "Run External Commands, Scripts, and Programs"

See Also

`!` (exclamation point) | `computer` | `dos` | `perl` | `unix`

Introduced before R2006a

table

Create table from workspace variables

Syntax

```
T = table(var1,...,varN)
T = table(var1,...,varN,Name,Value)
T = table
```

Description

`T = table(var1,...,varN)` creates a table from the input variables, var1,...,varN . Variables can be of different sizes and data types, but all variables must have the same number of rows.

For more information on creating and using the `table` data type, see “Tables”.

`T = table(var1,...,varN,Name,Value)` includes additional options specified by one or more Name,Value pair arguments.

For example, you can specify row names or variable names to include in the table.

`T = table` creates an empty 0-by-0 table.

Examples

Create Table from Workspace Variables

Define workspace variables with the same number of rows.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
```

Create a table, T, as a container for the workspace variables.

```
T = table(Age,Height,Weight,BloodPressure,...
    'RowNames',LastName)
```

```
T =

                Age    Height    Weight     BloodPressure
                ___    ______    ______    _______________

    Smith       38     71        176       124          93
    Johnson     43     69        163       109          77
    Williams    38     64        131       125          83
    Jones       40     67        133       117          75
    Brown       49     64        119       122          80
```

`table` names the variables with the workspace variable names.

Create Table and Specify Variable Names

Create a table from arrays with different data types.

```
T = table(['M';'F';'M'],[45;32;34],...
    {'NY';'CA';'MA'},logical([1;0;0]),...
    'VariableNames',{'Gender' 'Age' 'State' 'Vote'})
```

```
T =

    Gender    Age    State    Vote
    ______    ___    _____    _____

    M         45     'NY'     true
    F         32     'CA'     false
    M         34     'MA'     false
```

Each variable in T contains 3 rows.

`Gender` is a character array, `Age` is a double-precision array, `State` is a cell array of strings, and `Vote` is a logical array. You can use the function `summary` to print the data type and other information about the variables in the table.

- “Create a Table”
- “Modify Units, Descriptions and Table Variable Names”
- “Access Data in a Table”

Input Arguments

`var1,...,varN` — Input variables
arrays with the same number of rows

Input variables, specified as arrays with the same number of rows. The input variables can be of different sizes and different data types.

Common input variables are numeric arrays, logical arrays, character arrays, structure arrays, or cell arrays. Furthermore, input variables can be objects that are arrays that support indexing of the form `var(index1,...indexN)`, where `index1` is a numeric or logical vector that corresponds to rows of the variable `var`. In addition, the array must implement a `vertcat` method and a `size` method with a `dim` argument.

Example: `table([1:4]',ones(4,3,2),eye(4,2))` creates a table from variables with 4 rows, but different sizes.

Example: `table([1:3]',{'one';'two';'three'},['A';'B';'C'])` creates a table from variables with 3 rows, but different data types.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RowNames',{'row1','row2','row3'}` names the rows `row1`, `row2`, and `row3`.

`'RowNames'` — Row names
`{}` (default) | cell array of nonempty, distinct strings

Row names, specified as the comma-separated pair consisting of `'RowNames'` and a cell array of nonempty, distinct strings. The number of strings must equal the number of rows in the table.

`'VariableNames'` — Variable names
cell array of nonempty, distinct strings

Variable names, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of nonempty, distinct strings. The number of strings must equal the

number of variables. The strings must be valid MATLAB variable names. You can determine valid variable names using the function `isvarname`.

Output Arguments

`T` — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

Unless you specify the table variable names using the `VariableNames` name-value pair argument, the `table` function generates them automatically. If an input argument is a variable in the current workspace, `table` uses that variable's name as the table variable name. Otherwise, it generates a string from `Var` followed by an integer, such as `Var2`.

See Also

`array2table` | `cell2table` | `isvarname` | `readtable` | `struct2table` | `summary`

table2array

Convert table to homogeneous array

Syntax

```
A = table2array(T)
```

Description

`A = table2array(T)` converts the table, `T`, to a homogeneous array, `A`.

Examples

Convert Table of Numeric Data to Array

Create a table, `T`, consisting of numeric data.

```
T = table([1;2;3],[2 8; 4 10; 6 12],[3 12 21; 6 15 24; 9 18 27],...
    'VariableNames',{'One' 'Two' 'Three'})
```

```
T = 

    One        Two              Three        
    ___    ________    ________________

    1      2       8    3     12     21
    2      4      10    6     15     24
    3      6      12    9     18     27
```

Convert table, `T`, to an array.

```
A = table2array(T)
```

```
A =

     1     2     8     3    12    21
     2     4    10     6    15    24
```

```
     3     6    12     9    18    27
```

A contains two columns from variable Two and three columns from variable Three.

Convert Numeric Subset of Table to Array

Define the numeric subset of a table to convert to an array.

Create a table with nonnumeric data in the first variable.

```
T = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
    [71;69;64;67;64],[176;163;131;133;119],...
    'VariableNames',{'Gender' 'Age' 'Height' 'Weight'})
```

```
T =

    Gender    Age    Height    Weight
    ______    ___    ______    ______

    M         38     71        176
    M         43     69        163
    F         38     64        131
    F         40     67        133
    F         49     64        119
```

Convert T(:,2:4) to an array.

```
A = table2array(T(:,2:4))
```

```
A =

    38    71   176
    43    69   163
    38    64   131
    40    67   133
    49    64   119
```

A does not include data from the variable Gender.

Convert Table with Three-Dimensional Variables to Array

Create a table, T, with two rows and three variables where each variable has three dimensions.

```
T = table(ones(2,1,3),2*ones(2,2,3),3*ones(2,3,3),...
```

```
    'VariableNames',{'One' 'Two' 'Three'})
T =

        One               Two               Three
    ______________    ______________    ______________

    [1x1x3 double]    [1x2x3 double]    [1x3x3 double]
    [1x1x3 double]    [1x2x3 double]    [1x3x3 double]
```

The size of the table is 2-by-3.

Convert table `T` to an array.

```
A = table2array(T)
```

```
A(:,:,1) =

     1     2     2     3     3     3
     1     2     2     3     3     3


A(:,:,2) =

     1     2     2     3     3     3
     1     2     2     3     3     3


A(:,:,3) =

     1     2     2     3     3     3
     1     2     2     3     3     3
```

The size of `A` is 2-by-6-by-3.

Input Arguments

`T` — Input table

table

Input table, specified as a table. All variables in `T` must have sizes and data types that are compatible for horizontal concatenation. Specifically, the size of all variable dimensions greater than `2` must match.

- If `T` is an `m`-by`n` table with variables that each have one column, then each variable becomes one column in `A`, and `A` is an `m`-by-`n` array.
- If `T` contains variables that consist of more than one column, those variables become multiple columns in `A`, and the size of `A` is greater than the size of `T`.
- If `T` contains variables with more than two dimensions, the number of dimensions of `A` is the same as the number of variable dimensions.

More About

Tips

- `table2array` horizontally concatenates the variables in `T` to create `A`. If the variables in `T` are cell arrays, `table2array` does not concatenate their contents, and `A` is a cell array, equivalent to `table2cell(T)`. To create an array containing the contents of variables that are all cell arrays, use `cell2mat(table2cell(T))`.
- `table2array(T)` is equivalent to `T{:,:}`.

Algorithms

If `T` contains variables with different data types that are compatible for horizontal concatenation, `table2array` creates a homogeneous array, `A`, of the dominant data type. For example, if `T` contains `double` and `single` numeric data, `table2array(T)` returns an array with data type `single`.

- "Concatenating Objects of Different Classes"

See Also

`array2table` | `table` | `table2cell` | `table2struct`

table2cell

Convert table to cell array

Syntax

```
C = table2cell(T)
```

Description

`C = table2cell(T)` converts the table, `T`, to a cell array, `C`. Each variable in `T` becomes a column of cells in `C`.

Examples

Convert Table to Cell Array

Create a table, `T`, with five rows and three variables.

```
T = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'},...
    'RowNames',{'Smith' 'Johnson' 'Williams' 'Jones' 'Brown'})
```

```
T =

                Gender    Age    BloodPressure
                ______    ___    _____________

    Smith       M         38     124          93
    Johnson     M         43     109          77
    Williams    F         38     125          83
    Jones       F         40     117          75
    Brown       F         49     122          80
```

Convert `T` to a cell array.

```
C = table2cell(T)
```

```
C =

    'M'    [38]    [1x2 double]
    'M'    [43]    [1x2 double]
    'F'    [38]    [1x2 double]
    'F'    [40]    [1x2 double]
    'F'    [49]    [1x2 double]
```

`C` is a 5-by-3 cell array.

Vertically concatenate the table property, `T.Properties.VariableNames`, with `C` to include column headings for the cell array.

```
[T.Properties.VariableNames;C]
```

```
ans =

    'Gender'    'Age'    'BloodPressure'
    'M'         [ 38]       [1x2 double]
    'M'         [ 43]       [1x2 double]
    'F'         [ 38]       [1x2 double]
    'F'         [ 40]       [1x2 double]
    'F'         [ 49]       [1x2 double]
```

`T.Properties.VariableNames` is a cell array of strings.

Input Arguments

`T` — Input table

table

Input table, specified as a table.

If `T` is an `m`-by`n` table, then `C` is an `m`-by-`n` cell array.

See Also

`cell2table` | `table` | `table2array` | `table2struct`

table2struct

Convert table to structure array

Syntax

```
S = table2struct(T)
S = table2struct(T,'ToScalar',true)
```

Description

`S = table2struct(T)` converts the table, `T`, to a structure array, `S`. Each variable of `T` becomes a field in `S`. If `T` is an `m`-by-`n` table, then `S` is a `m`-by-1 structure array with `n` fields.

`S = table2struct(T,'ToScalar',true)` converts the table, `T`, to a scalar structure `S`. Each variable of `T` becomes a field in `S`. If `T` is a `m`-by-`n` table, then `S` has `n` fields, each of which has `m` rows.

Examples

Convert Table to Structure Array

Create a table, `T`, with five rows and three variables.

```
T = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'})
```

```
T =

    Gender    Age    BloodPressure
    ______    ___    _____________

    M         38     124          93
    M         43     109          77
    F         38     125          83
    F         40     117          75
```

```
    F         49     122          80
```

Convert T to a structure array.

```
S = table2struct(T)
```

```
S =

5x1 struct array with fields:

    Gender
    Age
    BloodPressure
```

The structure is 5-by-1, corresponding to the five rows of the table, T. The three fields of S correspond to the three variables from T.

Display the field data for the first element of S.

```
S(1)
```

```
ans =

           Gender: 'M'
              Age: 38
    BloodPressure: [124 93]
```

The information corresponds to the first row of the table.

Convert Table to Scalar Structure

Create a table, T, with five rows and three variables.

```
T = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'})
```

```
T =

    Gender    Age     BloodPressure
    ______    ___    ______________

    M         38     124          93
    M         43     109          77
    F         38     125          83
```

```
    F          40      117           75
    F          49      122           80
```

Convert `T` to a scalar structure.

```
S = table2struct(T,'ToScalar',true)
```

```
S =

           Gender: [5x1 char]
              Age: [5x1 double]
    BloodPressure: [5x2 double]
```

The data in the fields of the scalar structure are 5-by-1, corresponding to the five rows in the table `T`.

Display the data for the field `BloodPressure`.

```
S.BloodPressure
```

```
ans =

   124    93
   109    77
   125    83
   117    75
   122    80
```

The structure field `BloodPressure` contains all of the data that was in the variable of the same name from table `T`.

Convert Table with Row Names to Structure

Create a table, `T`, that includes row names.

```
T = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'},...
    'RowNames',{'Smith' 'Johnson' 'Williams' 'Jones' 'Brown'})
```

```
T =

                Gender    Age    BloodPressure
                ______    ___    _____________
```

```
    Smith       M          38      124          93
    Johnson     M          43      109          77
    Williams    F          38      125          83
    Jones       F          40      117          75
    Brown       F          49      122          80
```

Convert `T` to a scalar structure.

```
S = table2struct(T,'ToScalar',true)
```

```
S =

           Gender: [5x1 char]
              Age: [5x1 double]
    BloodPressure: [5x2 double]
```

Add a field for the row names from the table.

```
S.RowNames = T.Properties.RowNames
```

```
S =

           Gender: [5x1 char]
              Age: [5x1 double]
    BloodPressure: [5x2 double]
         RowNames: {5x1 cell}
```

If `S` is a nonscalar structure, use `[S.RowNames] = T.Properties.RowNames{:}` to include a field with the row names from the table.

Input Arguments

T — Input table

table

Input table, specified as a table.

See Also

`struct2table` | `table` | `table2array` | `table2cell`

Table Properties

Access and modify table metadata properties

Access and Modify Properties

A table, `T`, has properties that store metadata such as its variable names, row names, descriptions, and variable units. `T.Properties` returns a summary of all of the table properties.

You can access a property using *`T`*`.Properties.`*`PropName`*, where *`T`* is the name of the table and *`PropName`* is one of the table properties. For example, to access the `VariableDescriptions` property of a table named `Patients`, use `Patients.Properties.VariableDescriptions`.

You can modify a property value using *`T`*`.Properties.`*`PropName = P` where *`T`* is the name of the table, *`PropName`* is one of the table properties, and *`P`* is the desired property value. For example, to modify the `VariableUnits` property of a table named `Patients`, use `Patients.Properties.VariableUnits = P` where *`P`* is a cell array of strings containing the specified information.

In contrast, you can access and modify variables within a table using *`T.Variable`* or *`T.Variable = V`*, where *`T`* is the name of the table, *`Variable`* is the name of the variable you want to access or modify, and *`V`* is the variable value you want.

Properties

`VariableNames` — Variable names

cell array of nonempty, distinct strings

Variable names, specified as a cell array of nonempty, distinct strings. Variable names must be valid MATLAB variable names. The number of strings must equal the number of variables. MATLAB removes any leading or trailing white space from the strings.

If valid MATLAB identifiers are not available for use as variable names, MATLAB uses a cell array of `N` strings of the form `{'Var1' ... 'Var`*`N`*`'}` where *`N`* is the number of variables. You can determine valid MATLAB variable names using the function `isvarname`.

The variable names are visible when viewing the table and when using the `summary` function. Furthermore, you can use the variable names within parentheses, within curly braces, or with dot indexing to access table data.

Example

```
%% Create a table
T = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
    [71;69;64;67;64],[176;163;131;133;119])
```

```
T =

    Var1    Var2    Var3    Var4
    ____    ____    ____    ____

    M       38      71      176
    M       43      69      163
    F       38      64      131
    F       40      67      133
    F       49      64      119
```

```
%% Modify variable names
T.Properties.VariableNames = {'Gender' 'Age' 'Height' 'Weight'}
```

```
T =

    Gender    Age    Height    Weight
    ______    ___    ______    ______

    M         38     71        176
    M         43     69        163
    F         38     64        131
    F         40     67        133
    F         49     64        119
```

```
%% Create a subtable
%
% Use variable names within parentheses to create a subtable.
% Include al the rows, but only the variables Age and Gender.
T(:,{'Age','Gender'})
```

```
ans =

    Age    Gender
    ___    ______
```

```
    38     M
    43     M
    38     F
    40     F
    49     F

%% Extract data from the table
%
% Use variable names within curly braces to extract the
% numeric data from the variables Height and Weight.
T{:,{'Height','Weight'}}

ans =

    71   176
    69   163
    64   131
    67   133
    64   119

% Use variable names with dot indexing to extract
% all the data from the variable Gender.
T.Gender

ans =

M
M
F
F
F
```

`RowNames` — Row names

`{}` (default) | cell array of nonempty, distinct strings

Row names, specified as a cell array of nonempty, distinct strings. This property can be empty, but if not empty, the number of strings must equal the number of rows in the table. MATLAB removes any leading or trailing white space from the strings. The default property value is an empty cell array.

The row names are visible when you view the table. Furthermore, you can use the row names within parentheses or curly braces to access the table data.

Example

```
% Create a table
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);

% Add row names
T.Properties.RowNames = LastName;

% Remove row names
T.Properties.RowNames = {};
```

`DimensionNames` — Dimension names

`{'Row' 'Variable'}` (default) | two-element cell array of strings

Dimension names, specified as a two-element cell array of strings.

Example

```
% Create a table
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic,...
    'RowNames',LastName);

% Add dimension names
T.Properties.DimensionNames = {'Patient' 'Data'};
```

`Description` — Table description

`''` (default) | string

Table description, specified as a string. This string is visible when using the `summary` function.

Example

```
% Create a table
load patients
T = table(Gender,Age,Height,Weight);

% Add a table description
T.Properties.Description = 'Simulated patient data';

%View the summary
format compact
summary(T)
```

```
Description:  Simulated patient data
```

```
Variables:
    Gender: 100x1 char
    Age: 100x1 double
        Values:
            min       25
            median    39
            max       50
    Height: 100x1 double
        Values:
            min       60
            median    67
            max       72
    Weight: 100x1 double
        Values:
            min         111
            median    142.5
            max         202
```

`VariableDescriptions` — Variable descriptions

{} (default) | cell array of strings

Variable descriptions, specified as a cell array of strings. This property can be an empty cell array, which is the default. If the cell array is not empty, the number of strings must equal the number of variables. You can specify an individual empty string within the cell array for a variable that does not have a description.

The variable descriptions are visible when using the `summary` function.

Example

```
% Create a table
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);

% Add variable descriptions
T.Properties.VariableDescriptions = {'' '' '' '' ...
    'Has the patient ever been a smoker' ...
    'Systolic Pressure' 'Diastolic Pressure'};

% View the summary
format compact
summary(T)

Variables:
```

```
    Gender: 100x1 char
    Age: 100x1 double
        Values:
            min       25
            median    39
            max       50
    Height: 100x1 double
        Values:
            min       60
            median    67
            max       72
    Weight: 100x1 double
        Values:
            min         111
            median    142.5
            max         202
    Smoker: 100x1 logical
        Description:  Has the patient ever been a smoker
        Values:
            true     34
            false    66
    Systolic: 100x1 double
        Description:  Systolic Pressure
        Values:
            min       109
            median    122
            max       138
    Diastolic: 100x1 double
        Description:  Diastolic Pressure
        Values:
            min         68
            median    81.5
            max         99
```

```
% Remove all the variable descriptions
T.Properties.VariableDescriptions = {};
```

`VariableUnits` — Variable units

{} (default) | cell array of strings

Variable units, specified as a cell array of strings. This property can be an empty cell array, which is the default. If the cell array is not empty, the number of strings must equal the number of variables. You can specify an individual empty string within the cell array for a variable that does not have units.

The variable units are visible when using the `summary` function.

Example

```
% Create a table
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);

% Add variable units
T.Properties.VariableUnits = {'' 'Yrs' 'In' 'Lbs' '' 'mm Hg' 'mm Hg'};

% View the summary
format compact
summary(T)

Variables:
    Gender: 100x1 char
    Age: 100x1 double
        Units:  Yrs
        Values:
            min       25
            median    39
            max       50
    Height: 100x1 double
        Units:  In
        Values:
            min       60
            median    67
            max       72
    Weight: 100x1 double
        Units:  Lbs
        Values:
            min         111
            median    142.5
            max         202
    Smoker: 100x1 logical
        Values:
            true     34
            false    66
    Systolic: 100x1 double
        Units:  mm Hg
        Values:
            min       109
            median    122
            max       138
```

```
    Diastolic: 100x1 double
        Units:  mm Hg
        Values:
            min         68
            median    81.5
            max         99
```

```
% Remove all the variable units
T.Properties.VariableUnits = {};
```

`UserData` — Additional table information

{} (default) | variable containing information in any data type

Additional table information, specified as a variable containing information in any data type.

Example

```
% Create a table
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);

% Add an anonymous function to the table metadata
formula = @(x) x.^2;
T.Properties.UserData = formula;
```

More About

- "Access Data in a Table"

See Also

`isvarname` | `summary` | `table`

TabularTextDatastore Properties

Access and modify `TabularTextDatastore` properties

`TabularTextDatastore` properties describe the files associated with a `TabularTextDatastore` object. Specifically, the properties describe the format of the data in the files and control how the data should be read from the datastore. By changing property values, you can modify certain aspects of the datastore. Use dot notation to view or modify a particular property of a `TabularTextDatastore` object:

```
ds = datastore('airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.MissingValue = 0;
```

You also can specify the value of `TabularTextDatastore` properties using name-value pair arguments when you create a datastore using the `datastore` function:

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'MissingValue',0)
```

The first file specified by the `Files` property determines the variable names and format information. The `datastore` function reevaluates this information when you change any of the following properties of a `TabularTextDatastore`:

`Files`	`Delimiter`	`CommentStyle`
`ReadVariableNames`	`RowDelimiter`	`Whitespace`
`NumHeaderLines`	`TreatAsMissing`	`MultipleDelimitersAsOne`

File Properties

`Files` — Files included in datastore

cell array of strings

Files included in the datastore, specified as a cell array of strings, where each string is a full path to a file.

The first file specified by the `Files` property determines the variable names and format information for all files in the datastore.

Example: `{'C:\dir\data\mydata1.csv','C:\dir\data\mydata2.csv'}`

`ReadVariableNames` — Indicator for reading first row of first file as variable names

`true` (default) | `false` | `1` | `0`

Indicator for reading first row of the first file in the datastore as variable names, specified as either `true` (`1`) or `false` (`0`).

- If `true`, then the first nonheader row of the first file determines the variable names for the data.
- If `false`, then the first nonheader row of the first file contains the first row of data. The data is assigned default variable names, `Var1`, `Var2`, and so on.

Data Types: `logical`

`VariableNames` — Names of variables

cell array of strings

Names of variables in the datastore, specified as a cell array of strings. Specify the variable names in the order in which they appear in the files. If you do not specify the variable names, they are detected from the first nonheader line in the first file of the datastore. When modifying the `VariableNames` property, the number of new variable names must match the number of original variable names.

If `ReadVariableNames` is `false`, then `VariableNames` defaults to `{'Var1','Var2', ...}`.

Example: `{'Time','Name','Quantity'}`

Text Format Properties

`NumHeaderLines` — Number of lines to skip at beginning of file

`0` (default) | positive integer

Number of lines to skip at the beginning of the file, specified as a positive integer.

`datastore` ignores the specified number of header lines before reading the variable names or data.

Data Types: `double`

`Delimiter` — Field delimiter characters

`','` (default) | string | cell array of strings

Field delimiter characters, specified as a string or a cell array of strings. Specify multiple delimiters in a cell array of strings.

Example: `'|'`

Example: `{';','*'}`

Repeated delimiter characters in a file are interpreted as separate delimiters with empty fields between them.

When you specify one of the following escape sequences as a delimiter, it is converted to the corresponding control character:

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (\)

`RowDelimiter` — Row delimiter character

`\r\n` (default) | string

Row delimiter character, specified as a string. The string must be a single character or one of the strings, `'\r'`, `'\n'`, or `'\r\n'`.

Example: `':'`

`TreatAsMissing` — Numeric values to treat as missing values

`''` (default) | string | cell array of strings

Numeric values to treat as missing values, specified as a single string or cell array of strings. This option only applies to numeric fields. This property is equivalent to the `TreatAsEmpty` name-value pair argument for the `textscan` function.

Example: `'NA'`

Example: `{'-',''}`

Data Types: `char` | `cell`

`MissingValue` — Value for missing numeric fields

`NaN` (default) | scalar

Value for missing numeric fields in delimited text files, specified as a scalar. This property is equivalent to the `EmptyValue` name-value pair argument for the `textscan` function.

Data Types: `double`

Advanced Text Format Properties

`TextscanFormats` — Format of the data fields

cell array of strings

Format of the data fields, specified as a cell array of strings, where each string contains one conversion specifier.

When you specify or modify the `TextscanFormats` property, you can use the same conversion specifiers that the `textscan` function accepts, including specifiers that skip fields using an asterisk (*) character and specifiers that skip literal text. The number of conversion specifiers must match the number of variables in the `VariableNames` property.

If the value of `TextscanFormats` includes conversion specifiers that skip fields using asterisk characters (*), then the value of the `SelectedVariableNames` property automatically updates. MATLAB uses the `%*q` conversion specifier to skip fields omitted by the `SelectedVariableNames` property and treats the field contents as literal strings. For fixed width files, indicate a skipped field using the appropriate conversion specifier along with the field width. For example, `%*52c` skips a field that contains 52 characters.

If you do not specify a value for `TextscanFormats`, then `datastore` determines the format of the data fields by scanning text from the first non-header line in the first file of the datastore.

Example: `{'%s','%s','%f'}`

`ExponentCharacters` — Exponent characters

`'eEdD'` (default) | string

Exponent characters, specified as a string. The default exponent characters are `e`, `E`, `d`, and `D`.

`CommentStyle` — Style of comments

`''` (default) | string | cell array of strings

Style of comments in the file, specified as a string or cell array of strings.

For example, specify a string such as `'%'` to ignore characters following the string on the same line. Specify a cell array of two strings, such as `{'/*', '*/'}`, to ignore characters between the strings.

When reading from a `TabularTextDatastore`, the `read` function checks for comments only at the start of each field, not within a field.

Example: `'CommentStyle',{'/*', '*/'}`

Data Types: `char` | `cell`

`Whitespace` — White-space characters

`' \b\t'` (default) | string

White-space characters, specified as a string of one or more characters.

When you specify one of the following escape sequences as any white-space character, `datastore` converts that sequence to the corresponding control character:

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (\)

Example: `' \b\t'`

Data Types: `char`

`MultipleDelimitersAsOne` — Multiple delimiter handling

`0 (false)` (default) | `1 (true)`

Multiple delimiter handling, specified as either `true` or `false`. If `true`, then `datastore` treats consecutive delimiters as a single delimiter. Repeated delimiters separated by white-space are also treated as a single delimiter.

Properties that Control the Table Returned by `preview`, `read`, `readall` Functions

`SelectedVariableNames` — Variables to read

cell array of strings

Variables to read from the file, specified as a cell array of strings, where each string contains the name of one variable. You can specify the variable names in any order.

Example: `{'Var3','Var7','Var4'}`

`SelectedFormats` — Formats of selected variables

cell array of strings

Formats of the selected variables to read, specified as a cell array of strings, where each string contains one conversion specifier. The variables to read are indicated by the `SelectedVariableNames` property. The number of strings in `SelectedFormats` must match the number of variables to read.

You can use the same conversion specifiers that the `textscan` function accepts, including specifiers that skip literal text. However, you cannot use a conversion specifier that skips a field. That is, the conversion specifier cannot include an asterisk character (*).

Example: `{'%d','%d'}`

`ReadSize` — Amount of data to read

20000 (default) | positive scalar | `'file'`

Amount of data to read in a call to the `read` function, specified as a positive scalar or the string, `'file'`.

- If `ReadSize` is a positive integer, then each call to `read` reads up to the specified number of rows from the datastore.
- If `ReadSize` is `'file'`, then each call to `read` reads all of the data in one file.

When you change `ReadSize` from a numeric scalar to `'file'` or vice versa, MATLAB resets the datastore to the state where no data has been read from it.

`RowsPerRead` — Upper limit on number of rows to read

20000 (default) | positive scalar

Note: `RowsPerRead` will be removed in a future release. Use `ReadSize` instead.

Upper limit on number of rows to read from a file in a call to the `read` function, specified as a positive scalar.

See Also

`datastore` | `reset` | `textscan`

More About

- Using TabularTextDatastore Objects

tan

Tangent of argument in radians

Syntax

```
Y = tan(X)
```

Description

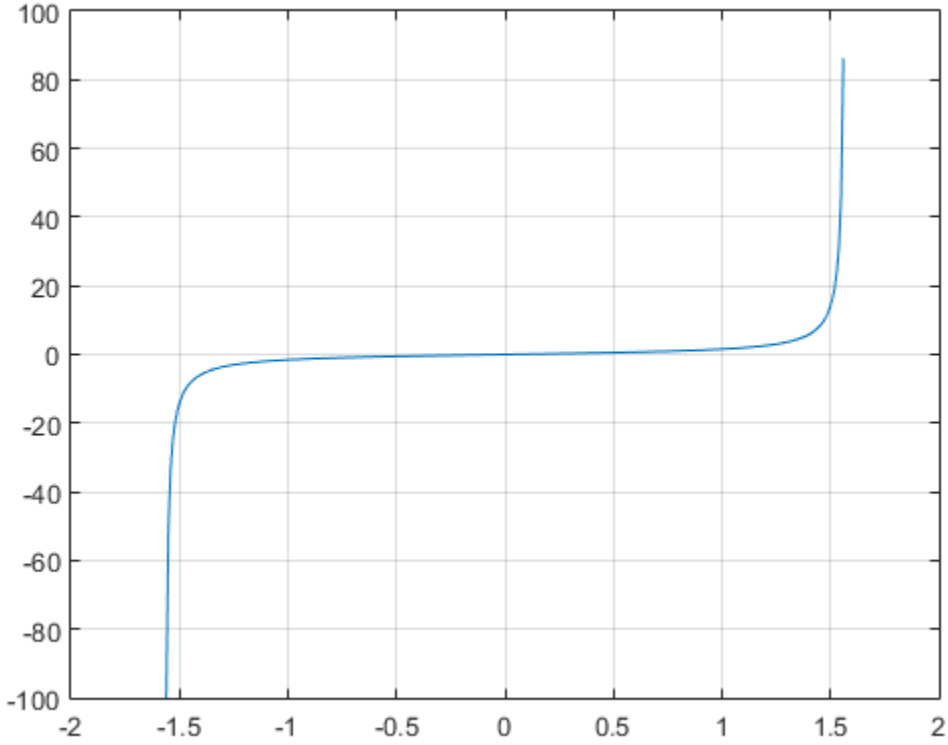
`Y = tan(X)` returns the tangent of each element of `X`. The `tan` function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval [-Inf, Inf], `tan` returns real values in the interval [-Inf ,Inf]. For complex values of X, `tan` returns complex values. All angles are in radians.

Examples

Plot Tangent Function

Plot the tangent function over the domain $-\pi/2 \le x \le \pi/2$.

```
x = (-pi/2)+0.01:0.01:(pi/2)-0.01;
plot(x,tan(x)), grid on
```



Tangent of Vector of Complex Angles

Calculate the tangent of the complex angles in vector `x`.

```
x = [-i pi+i*pi/2 -1+i*4];
y = tan(x)
```

```
y =

   0.0000 - 0.7616i  -0.0000 + 0.9172i  -0.0006 + 1.0003i
```

Input Arguments

X — Input angle in radians

number | vector | matrix | multidimensional array

Input angle in radians, specified as a number, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Tangent of input angle

scalar value | vector | matrix | N-D array

Tangent of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or N-D array.

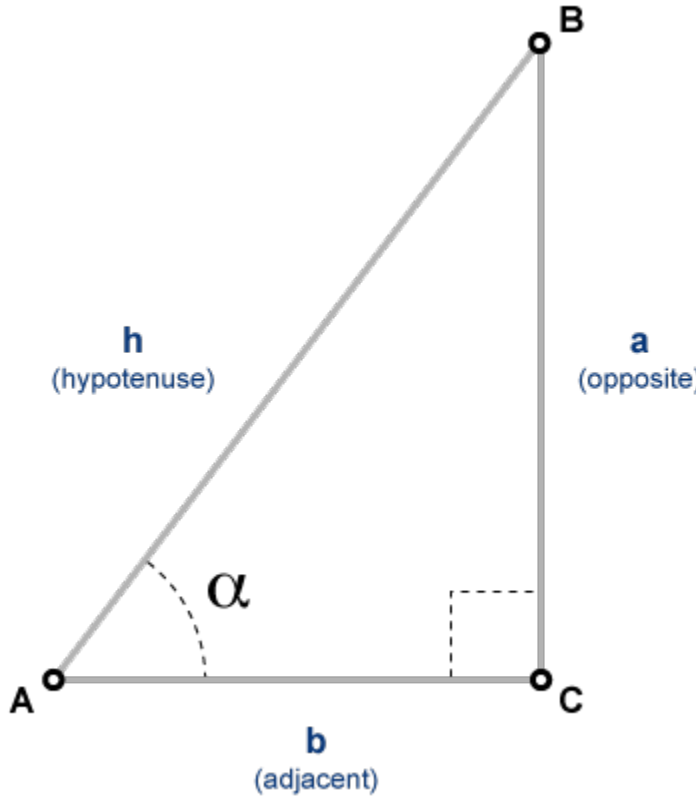
More About

Tangent Function

The tangent of an angle, α, defined with reference to a right angled triangle is

$$\tan(\alpha) = \frac{\text{opposite side}}{\text{adjacent side}} = \frac{a}{b}.$$

.



The tangent of a complex angle, α, is

$$\tan(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{i\left(e^{i\alpha} + e^{-i\alpha}\right)}.$$

.

See Also

`atan` | `atan2` | `atan2d` | `atand` | `tand` | `tanh`

Introduced before R2006a

tand

Tangent of argument in degrees

Syntax

```
Y = tand(X)
```

Description

`Y = tand(X)` returns the tangent of the elements of `X`, which are expressed in degrees.

Examples

Tangent of 90 degrees compared to tangent of π/2 radians

```
tand(90)
```

```
ans =

   Inf
```

```
tan(pi/2)
```

```
ans =

   1.6331e+16
```

`tand(90)` is infinite, whereas `tan(pi/2)` is large but finite.

Tangent of vector of complex angles, specified in degrees

```
z = [180+i 15+2i 10+3i];
y = tand(z)
```

```
y =
```

```
   0 + 0.0175i   0.2676 + 0.0374i   0.1758 + 0.0539i
```

Input Arguments

X — Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `tand` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Tangent of angle

scalar value | vector | matrix | N-D array

Tangent of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

See Also

`atan` | `atand` | `tan`

Introduced before R2006a

tanh

Hyperbolic tangent

Syntax

```
Y = tanh(X)
```

Description

The `tanh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

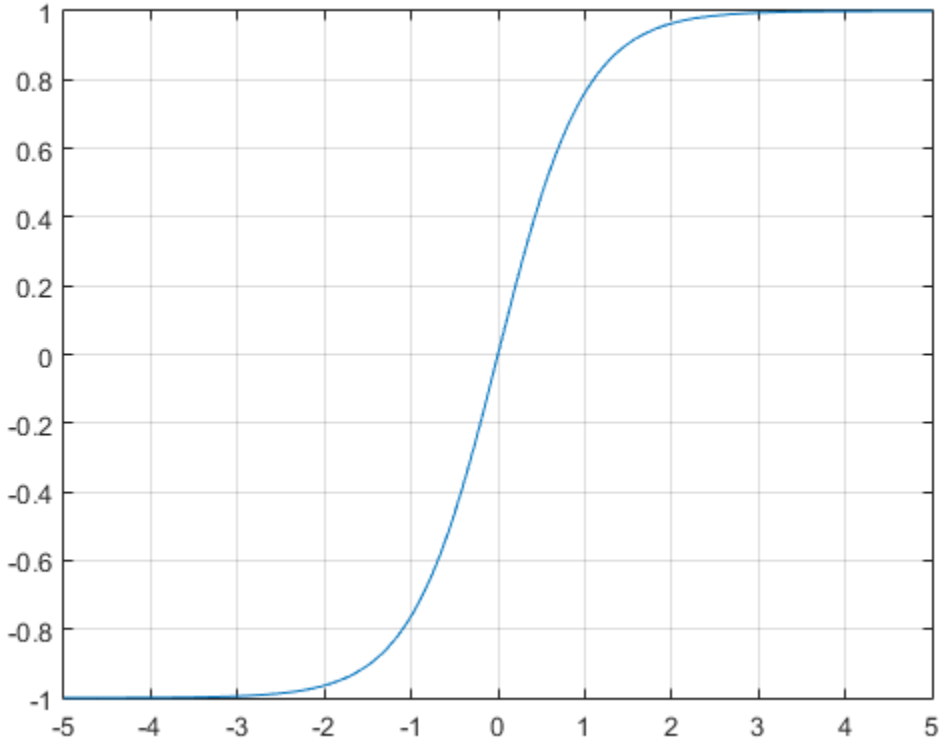
`Y = tanh(X)` returns the hyperbolic tangent of each element of `X`.

Examples

Graph Hyperbolic Tangent Function

Graph the hyperbolic tangent function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,tanh(x)), grid on
```



More About

Hyperbolic Tangent

The hyperbolic tangent of z is

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}.$$

See Also

`atan` | `atan2` | `tan` | `atanh` | `sinh` | `cosh`

Introduced before R2006a

tar

Compress files into tar file

Syntax

```
tar(tarfilename,files)
tar(tarfilename,files,rootfolder)
entrynames = tar(...)
```

Description

`tar(tarfilename,files)` creates a tar file named *`tarfilename`* from the list of files and folders specified in *`files`*. Folders recursively include all of their content. If *`files`* includes relative paths, the tar file also contains relative paths. The tar file does not include absolute paths.

`tar(tarfilename,files,rootfolder)` specifies the path for *`files`* relative to *`rootfolder`* rather than the current folder. Relative paths in the tar file reflect the relative paths in *`files`*, and do not include path information from *`rootfolder`*.

`entrynames` `= tar(...)` returns a string cell array of the names of the files contained in *`tarfilename`*. If *`files`* includes relative paths, *`entrynames`* also contains relative paths.

Input Arguments

`tarfilename`

String specifying the name of the tar file. If *`tarfilename`* has no extension, MATLAB appends the `.tar` extension. The *`tarfilename`* extension can end in `.tgz` or `.gz`. In this case, *`tarfilename`* is gzipped.

`files`

String or cell array of strings containing the list of files or folders included in *`tarfilename`*.

Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current folder or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~username/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`rootfolder`

String specifying the path for *`files`*.

Examples

Tar all files in the current folder to the file `backup.tgz`.

```
tar('backup.tgz','.');
```

More About

Tips

`tar` cannot compress folders larger than 2 GB.

See Also

`gzip` | `gunzip` | `untar` | `unzip` | `zip`

Introduced before R2006a

targetupdater

Set up support package that is already installed

Syntax

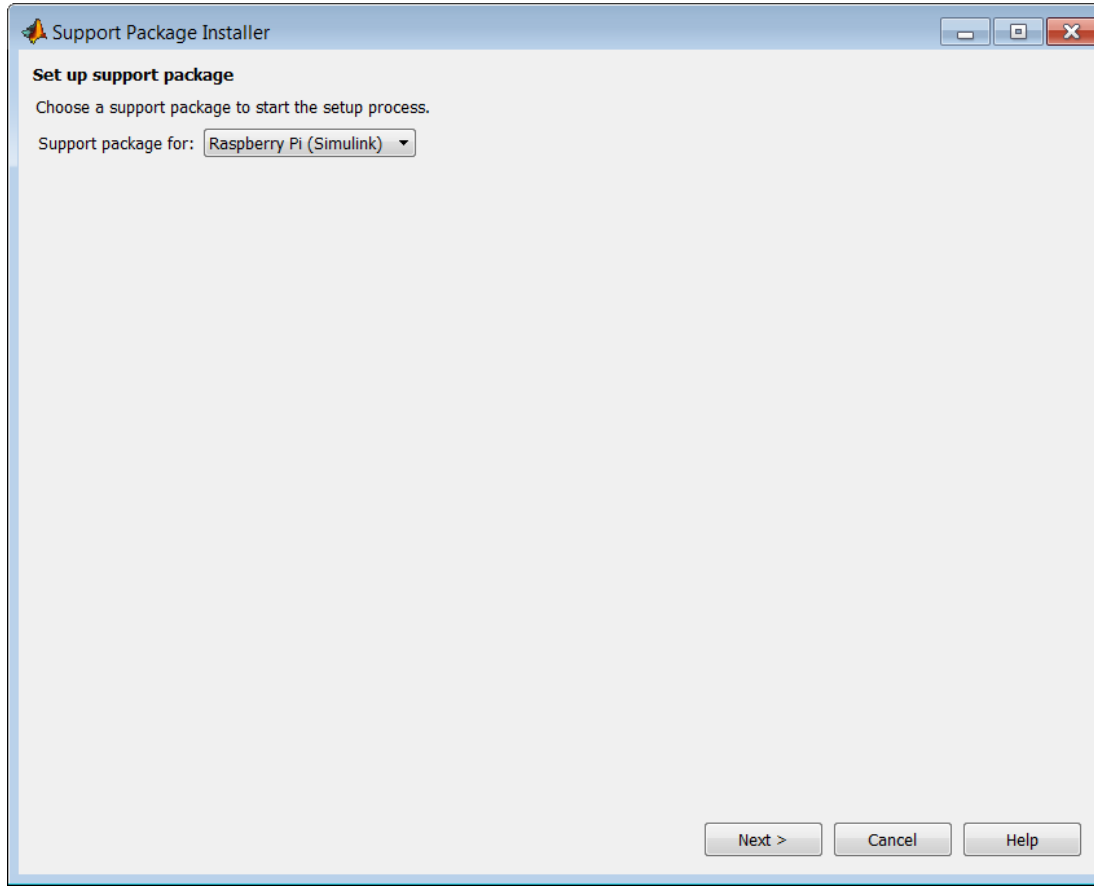
Description

To set up a support package that is already installed, use `targetupdater`.

Setting up a support package can include:

- Updating firmware on the target hardware.
- Validating the installation folder of third-party software.
- Validating licenses.

The `targetupdater` function starts Support Package Installer at the **Set up support package** screen.



Tip Some support packages do not require set up. If the **Set up support package** screen does not display an installed support package, the support package does not require set up.

To check whether a support package is installed, use the `matlabshared.supportpkg.getInstalled` function.

See Also

`matlabshared.supportpkg.checkForUpdate` | `matlabshared.supportpkg.getInstalled` | `supportPackageInstaller`

tcpclient

Create TCP/IP client object to communicate over TCP/IP

Syntax

```
t = tcpclient('Address',Port)
t = tcpclient('Address',Port,'Timeout',<timeout_value>)
```

Description

`t = tcpclient('Address',Port)` constructs a TCP/IP object, `t`, associated with remote host, `Address`, and remote port value, `Port`. The address can be either a remote host name or a remote IP address. The port must be a positive integer between `1` and `65535`.

If an invalid address or port is specified, or the connection to the server cannot be established, the object will not be created.

`t = tcpclient('Address',Port,'Timeout',<timeout_value>)` additionally sets a timeout value. The `Timeout` property specifies the waiting time to complete read and write operations in seconds, and the default is `10`.

Examples

Create Object Using Host Name

Create a TCP/IP object called `t`, using the host address shown, and Port of `4012`.

```
t = tcpclient('www.mathworks.com', 4012)
```

```
t =

   tcpclient with properties:

          Address: 'www.mathworks.com'
             Port: 4012
          Timeout: 10
```

```
   BytesAvailable: 0
```

Create Object Using IP Address

Create a TCP/IP object called `t`, using the IP address shown, and Port of `4012`.

```
t = tcpclient('172.28.154.231', 4012)

t =

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 10
   BytesAvailable: 0
```

Create Object Setting the Timeout Property

Create a TCP/IP object called `t`, and increase the `Timeout` to `20` seconds.

```
t = tcpclient('172.28.154.231', 4012, 'Timeout', 20)

t =

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 20
   BytesAvailable: 0
```

The output reflects the `Timeout` property change.

Input Arguments

`Address` — Remote host name or IP address for connection

character string

Remote host name or IP address for connection, specified as a character string. Specify address as the first argument when you create the `tcpclient` object.

Example: `t = tcpclient('www.mathworks.com', 4012)`

Data Types: `char`

`Port` — Remote host port for connection

numeric scalar

Remote host port for connection, specified as a numeric scalar. Specify port number as the second argument when you create the `tcpclient` object. The `Port` must be a positive integer between `1` and `65535`.

Example: `t = tcpclient('www.mathworks.com', 4012)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `t = tcpclient('172.28.154.231', 120, 'Timeout', 15)`

`'Timeout'` — Timeout for read/write operation

10 (default) | numeric scalar

Timeout for read/write operation specified as the comma-separated pair consisting of 'Timeout" and a positive value of type `double`. You can change the value either during object creation or after you create the object.

For information on how to change the timeout value after object creation, see Configure Properties for TCP/IP Communication.

Example: `t = tcpclient('172.28.154.231', 4012, 'Timeout', 20)`

Data Types: `double`

tempdir

Name of system's temporary folder

Syntax

```
tmp_folder = tempdir
```

Description

`tmp_folder = tempdir` returns the name of the system's temporary folder, if one exists. This function does not create a new folder.

More About

- "Create Temporary Files"

See Also

`delete` | `recycle` | `tempname`

Introduced before R2006a

tempname

Unique name for temporary file

Syntax

```
tmpName = tempname
```

Description

`tmpName = tempname` returns a string, `tmpName`, suitable for use as a temporary file path in your system's temporary folder.

Examples

Create Temporary File Name with Extension

Create a temporary file name that has the extension, `.dat`, by concatenating two strings.

```
tmpName = [tempname,'.dat'];
```

Write Data to Temporary File

Create a temporary file name.

```
filename = tempname;
```

Create a new file with the temporary file name, and write data to the file.

```
fileID = fopen(filename,'w');
fwrite(fileID,magic(5));
fclose(fileID);
```

Limitations

- In most cases, `tempname` generates a universally unique identifier (UUID). However, if you run MATLAB without JVM software, then `tempname` generates a random

string using the CPU counter and time, and this string is not guaranteed to be unique. For more information about the MATLAB startup option that does not load JVM software, see “Commonly Used Startup Options”.

More About

- “Create Temporary Files”

See Also

`tempdir`

Introduced before R2006a

tetramesh

Tetrahedron mesh plot

Syntax

```
tetramesh(T,X,c)
tetramesh(T,X)
tetramesh(TR)
h = tetramesh(...)
tetramesh(...,'param','value','param','value'...)
```

Description

`tetramesh(T,X,c)` displays the tetrahedrons defined in the `m`-by-4 matrix `T` as mesh. `T` is usually the output of a Delaunay triangulation of a 3-D set of points. A row of `T` contains indices into `X` of the vertices of a tetrahedron. `X` is an `n`-by-3 matrix, representing `n` points in 3 dimension. The tetrahedron colors are defined by the vector `C`, which is used as indices into the current colormap.

`tetramesh(T,X)` uses `C = 1:m` as the color for the `m` tetrahedra. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

`tetramesh(TR)` displays the tetrahedra in a triangulation representation.

`h = tetramesh(...)` returns a vector of tetrahedron handles. Each element of `h` is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch `'Visible'` property `'on'` or `'off'`.

`tetramesh(...,'param','value','param','value'...)` allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to `0.9`. You can overwrite this value by using the property name/property value pair `('FaceAlpha',value)` where `value` is a number between `0` and `1`. See Patch Properties for information about the available properties.

Examples

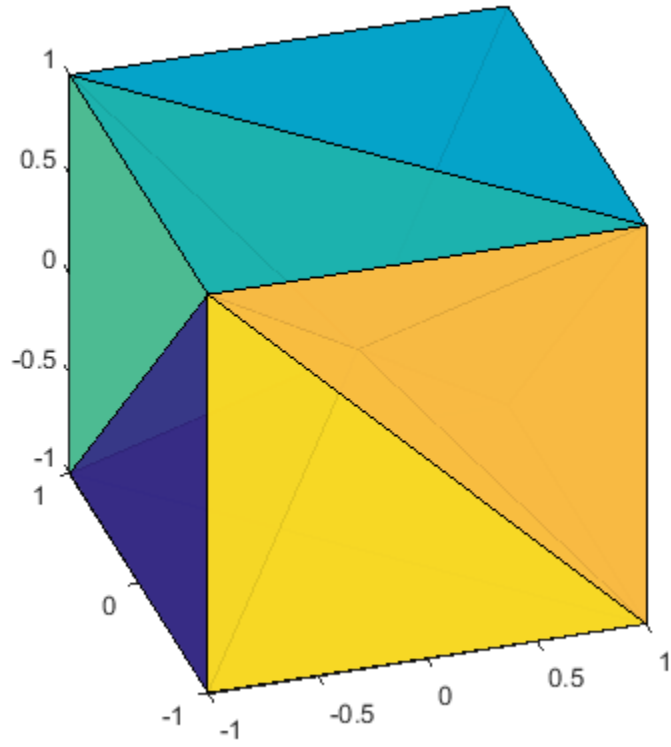
Plot Tetrahedrons of 3-D Delaunay Triangulation

Generate a 3-D Delaunay triangulation, then use `tetramesh` to visualize the tetrahedrons.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d); % a cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
```

`[x,y,z]` are corners of a cube plus the center.

```
DT = delaunayTriangulation(x,y,z);
tetramesh(DT);
camorbit(20,0)
```



See Also

`trimesh` | `patch` | `triangulation` | `delaunayTriangulation` | `freeBoundary(triangulation)` | `trisurf` | `delaunayn`

Introduced before R2006a

matlab.unittest.Test class

Package: matlab.unittest
Superclasses: matlab.unittest.TestSuite

Specification of a single test method

Description

The `matlab.unittest.Test` class holds the information needed for the `TestRunner` object to be able to run a single `Test` method of a `TestCase` class. A scalar `Test` instance is the fundamental element contained in `TestSuite` arrays. A simple array of `Test` instances is a commonly used form of a `TestSuite` array.

Properties

`Name`

Name of the `Test` element.

`Parameterization`

Row vector of parameters required for the `Test`. The `Parameterization` property contains all the parameterized data needed by the `TestRunner`.

`SharedTestFixtures`

Row vector of fixtures required for the `Test`. The `SharedTestFixtures` property contains all the fixtures specified by the `SharedTestFixtures` class-level attribute of the `TestCase` class.

Examples

Show Class of a TestSuite Array

Create a suite of `Test` objects of all test methods in the `BankAccountTest` class.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?BankAccountTest);

whos suite
```

```
  Name       Size            Bytes  Class                    Attributes

  suite      1x5              1636  matlab.unittest.Test
```

Each test is a `matlab.unittest.Test` object.

Display test method names.

```
{suite.Name}'
```

```
ans =

    'BankAccountTest/testConstructor'
    'BankAccountTest/testConstructorNotEnoughInputs'
    'BankAccountTest/testDesposit'
    'BankAccountTest/testWithdraw'
    'BankAccountTest/testNotifyInsufficientFunds'
```

See Also

`matlab.unittest.TestSuite` | `matlab.unittest.TestRunner` | `matlab.unittest.TestCase` | `matlab.unittest.fixtures`

matlab.unittest.TestCase class

Package: matlab.unittest

Superclass of all `matlab.unittest` test classes

Description

The `TestCase` class is the means by which a test is written in the `matlab.unittest` framework. It provides the means to write and identify test content, as well as test fixture setup and teardown routines. Creating such a test requires deriving from `TestCase` to produce a `TestCase` subclass. Then, subclasses can leverage the metadata attributes to specify tests and test fixtures.

Construction

Note: The ability to directly construct `TestCase` instances by calling the `matlab.unittest.TestCase` constructor will be removed in a future release. For interactive, command line use, instantiate a `TestCase` using the `forInteractiveUse` static method.

Use the `forInteractiveUse` static method to create a `TestCase` for interactive, command line use. When tests are run in the framework, `TestCase` instances are constructed by the `matlab.unittest.TestRunner`.

Methods

addTeardown	Dynamically add teardown routine
applyFixture	Use fixture with `TestCase`
forInteractiveUse	Create TestCase for interactive use

getSharedTestFixtures	Provide access to shared test fixtures
log	Record diagnostic information
run	Run `TestCase` test

Inherited Methods

The `TestCase` class inherits methods from the following classes:

matlab.unittest.qualifications.Assertable	Qualification to validate preconditions of a test
matlab.unittest.qualifications.Assumable	Qualification to filter test content
matlab.unittest.qualifications.FatalAssertable	Qualification to abort test execution
matlab.unittest.qualifications.Verifiable	Qualification to produce soft-failure conditions

Attributes

Class Attributes

`TestCase` objects support the following class level attributes. Specify class-level attributes in the `classdef` block before the class name.

`SharedTestFixtures`	Class block to contain shared test fixtures. You must define `SharedTestFixtures` as a cell array of `matlab.unittest.fixtures.Fixture` instances.

`TestTags`	Class block to contain tests tagged with a specified value. You must define `TestTags` as a cell array of non-empty strings, where each string is a tag for the test.

Method Attributes

Classes that derive from `TestCase` can define `methods` blocks which contain `matlab.unittest` framework-specific attributes to specify test content.

`Test`	Method block to contain test methods.
`TestMethodSetup`	Method block to contain setup code.
`TestMethodTeardown`	Method block to contain teardown code.
`TestClassSetup`	Method block to contain class level setup code.
`TestClassTeardown`	Method block to contain class level teardown code.
`ParameterCombination`	Method block to contain parameterized testing code. This attribute accepts the following values: • `'exhaustive'` (default): Test methods are invoked for all combinations of parameters. • `'sequential'`: Test methods are invoked with corresponding values from each parameter. Each parameter must contain the same number of values. • `'pairwise'`: Test methods are invoked for every pair of parameter values at least once.
`TestTags`	Method block to contain tests tagged with a specified value. You must define `TestTags` as a cell array of non-empty strings, where each string is a tag for the test.

Property Attributes

Classes that derive from `TestCase` can define `properties` blocks which contain `matlab.unittest` framework-specific attributes to specify test content.

`ClassSetupParameter`	Property block to define parameterized testing properties for methods in the `TestClassSetup` block
`MethodSetupParameter`	Property block to define parameterized testing properties for methods in the`MethodSetup` block
`TestParameter`	Property block to define parameterized testing properties for methods in the `Test` block

Events

`VerificationFailed`	Triggered upon failing verification. A `QualificationEventData` object is passed to listener callback functions.
`VerificationPassed`	Triggered upon passing verification. A `QualificationEventData` object is passed to listener callback functions.
`AssertionFailed`	Triggered upon failing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssertionPassed`	Triggered upon passing assertion. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionFailed`	Triggered upon failing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionPassed`	Triggered upon passing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.

`AssumptionFailed`	Triggered upon failing assumption. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionPassed`	Triggered upon passing assumption. A `QualificationEventData` object is passed to listener callback functions.
`ExceptionThrown`	Triggered by the `TestRunner` when an exception is thrown. An `ExceptionEventData` object is passed to listener callback functions.
`DiagnosticLogged`	Triggered by the `TestRunner` upon a call to the `log` method. A `LoggedDiagnosticEventData` object is passed to the listener callback functions.

Examples

Create Test Case Class

Create a test case class, `FigurePropertiesTest`, with `TestMethodSetup` and `TestMethodTeardown` methods.

```
classdef FigurePropertiesTest < matlab.unittest.TestCase

    properties
        TestFigure
    end

    methods(TestMethodSetup)
        function createFigure(testCase)
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    end
```

```
    methods(Test)

        function defaultCurrentPoint(testCase)

            cp = testCase.TestFigure.CurrentPoint;
            testCase.verifyEqual(cp, [0 0], ...
                'Default current point is incorrect')
        end

        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty

            co = testCase.TestFigure.CurrentObject;
            testCase.verifyThat(co, IsEmpty, ...
                'Default current object should be empty')
        end

    end

end
```

- “Create Basic Parameterized Test”
- “Create Advanced Parameterized Test”
- “Tag Unit Tests”

See Also

`matlab.unittest.qualifications.QualificationEventData` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.diagnostics.LoggedDiagnosticEventData` | `addlistener` | `matlab.unittest.constraints` | `matlab.unittest.qualifications` | `TestRunner`

More About

- “Method Attributes”
- “Class Attributes”

Introduced in R2013a

addTeardown

Class: matlab.unittest.TestCase
Package: matlab.unittest

Dynamically add teardown routine

Syntax

```
addTeardown(testCase,tearDownFcn)
addTeardown(testCase,tearDownFcn,arg1,...,argN)
```

Description

`addTeardown(testCase,tearDownFcn)` adds the tearDownFcn function handle that defines fixture teardown code to the testCase instance. The teardown code is executed in the reverse order to which it is added. This is known as LIFO (or Last-In-First-Out).

`addTeardown(testCase,tearDownFcn,arg1,...,argN)` provides input arguments to the `tearDownFcn`.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

Default:

`tearDownFcn`

Function, specified as a function handle, that defines the fixture teardown code

Default:

`arg1,...,argN`

Input arguments, 1 through N (if any), required by tearDownFcn, specified by any type. The argument type is specified by the function argument list.

Default:

Examples

Call addTeardown in a TestMethodSetup Method

```
classdef SomeTest < matlab.unittest.TestCase

    methods(TestMethodSetup)
        function createFixture(testCase)
            p = path;
            testCase.addTeardown(@path, p);
            addpath(fullfile(pwd,'testHelpers'));
        end
    end
end
```

applyFixture

Class: matlab.unittest.TestCase
Package: matlab.unittest

Use fixture with `TestCase`

Syntax

```
applyFixture(testCase,fixture)
```

Description

`applyFixture(testCase,fixture)` prepares the specified fixture for use with the `TestCase`. This method enables the use of a fixture within the scope of a single `Test` method or `TestCase` class. The life cycle of the fixture is tied to the `TestCase`. When the `TestCase` goes out of scope, the test framework tears down the fixture.

Call `applyFixture` within a `Test` method or `TestMethodSetup` method to use a fixture for the current test method alone. Use `applyFixture` within a `TestClassSetup` method to set up a fixture for the entire class.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

`fixture`

`matlab.unittest.fixtures.Fixture` instance

Examples

Apply Fixtures to `TestCase` Class

Create a temporary folder and make it the current working folder.

```
classdef applyFixtureTest < matlab.unittest.TestCase
    methods(TestMethodSetup)
        function addHelpers(testCase)
            import matlab.unittest.fixtures.TemporaryFolderFixture;
            import matlab.unittest.fixtures.CurrentFolderFixture;

            % Create a temporary folder and make it the current working
            % folder.
            tempFolder = testCase.applyFixture(TemporaryFolderFixture);
            testCase.applyFixture(CurrentFolderFixture(tempFolder.Folder));
        end
    end
end
```

Each test method can write files to the current working folder, which is the temporary folder. After each test method runs, the test framework restores the working folder to its previous state and deletes the temporary folder.

See Also

`matlab.unittest.fixtures`

matlab.unittest.TestCase.forInteractiveUse

Class: matlab.unittest.TestCase
Package: matlab.unittest

Create TestCase for interactive use

Syntax

```
tc = matlab.unittest.TestCase.forInteractiveUse
```

Description

`tc = matlab.unittest.TestCase.forInteractiveUse` creates a `TestCase` instance for interactive use. The `TestCase` is configured for experimentation at the command prompt. It reacts to qualification tests by printing messages to the screen for both passing and failing conditions.

Examples

Verify Values Using Interactive TestCase

Create a `TestCase` for interactive use.

```
import matlab.unittest.TestCase;
testCase = TestCase.forInteractiveUse;
```

Produce a passing verification.

```
testCase.verifyTrue(true, 'true should be true');
```

```
Interactive verification passed.
```

Produce a failing verification.

```
testCase.verifyTrue(false);
```

```
Interactive verification failed.
```

```
---------------------
Framework Diagnostic:
---------------------
verifyTrue failed.
--> The value must evaluate to "true".

Actual Value:
         0
```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.qualifications`

Introduced in R2014a

getSharedTestFixtures

Class: matlab.unittest.TestCase
Package: matlab.unittest

Provide access to shared test fixtures

Syntax

```
fixtures = getSharedTestFixtures(testCase)
fixtures = getSharedTestFixtures(testCase,fixtureClassName)
```

Description

`fixtures = getSharedTestFixtures(testCase)` provides access to the array of all shared test fixtures for `testCase`. `getSharedTestFixtures` returns an array of fixture objects, `fixtures`. Specify shared fixtures are using the `SharedTestFixtures` attribute for the `testCase` class.

`fixtures = getSharedTestFixtures(testCase,fixtureClassName)` returns only the shared fixtures that have the class name `fixtureClassName`.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

Default:

`fixtureClassName`

Name of test fixture class, specified as a string

Examples

Obtain Array of All Shared Fixtures

Create the following class, `myTest`, on your MATLAB path. Two shared fixtures are used within the test method. This example assumes that the subfolder `helperFiles` exists in your working folder. Create the subfolder `helperFiles` in your working folder if it does not exist.

```
classdef (SharedTestFixtures={...
matlab.unittest.fixtures.PathFixture('helperFiles'),...
        matlab.unittest.fixtures.TemporaryFolderFixture}) ...
        myTest < matlab.unittest.TestCase
    methods(Test)
        function accessFixtures(testCase)
            myFixtures = testCase.getSharedTestFixtures
        end
    end
end
```

At the command prompt, run the test.

```
run(myTest);
```

```
Setting up PathFixture.
Description: Adds 'H:\Documents\doc_examples\helperFiles' to the path.
__________

Setting up TemporaryFolderFixture.
Description: Creates a temporary folder.
__________

Running myTest

myFixtures =

  1x2 heterogeneous Fixture (PathFixture, TemporaryFolderFixture) array with no propert

.
Done myTest
__________

Tearing down TemporaryFolderFixture.
```

```
Description: Deletes the temporary folder and all its contents.
__________

Tearing down PathFixture.
Description: Restores the path to its previous state.
__________
```

Access Shared Fixtures of Particular Class

Create the class, `mySecondTest`, on your MATLAB path.

```
classdef (SharedTestFixtures={...
        matlab.unittest.fixtures.TemporaryFolderFixture})...
        mySecondTest < matlab.unittest.TestCase
    methods(Test)
        function accessTemporaryFolderFixture(testCase)
            tempFolderFixture = testCase.getSharedTestFixtures...
                ('matlab.unittest.fixtures.TemporaryFolderFixture');
            temporaryFolder = tempFolderFixture.Folder
        end
    end
end
```

At the command prompt, run the test. The name of the temporary folder varies.

```
run(mySecondTest);
```

```
Setting up TemporaryFolderFixture.
Description: Creates a temporary folder.
__________

Running mySecondTest

temporaryFolder =

C:\Temp\tpb92c9c67_02fa_4714_bfb0_b2127df0f31d

.
Done mySecondTest
__________

Tearing down TemporaryFolderFixture.
Description: Deletes the temporary folder and all its contents.
```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.fixtures`

More About

- Class Attributes

log

Class: matlab.unittest.TestCase
Package: matlab.unittest

Record diagnostic information

Syntax

```
log(testCase,diagnostic)
log(testCase,v,diagnostic)
```

Description

`log(testCase,diagnostic)` logs the supplied diagnostic. The log method provides a means for tests to log information during their execution. The testing framework displays logged messages only if you configure it to do so by adding an appropriate plugin, such as the `matlab.unittest.plugins.LoggingPlugin`.

`log(testCase,v,diagnostic)` logs the diagnostic at the specified verbosity level, `v`.

Input Arguments

`testCase` — Instance of test case

`matlab.unittest.TestCase` instance

Instance of test case, specified as a `matlab.unittest.TestCase`.

`diagnostic` — Diagnostic information to display upon a failure

string | function handle | `matlab.unittest.diagnostics.Diagnostic` instance

Diagnostic information to display upon a failure, specified as a string, function handle, or `matlab.unittest.diagnostics.Diagnostic` instance.

`v` — Verbosity level

2 (default) | 1 | 3 | 4 | `matlab.unittest.Verbosity` enumeration

Verbosity level, specified as an integer value between 1 and 4 or a `matlab.unittest.Verbosity` enumeration object. The default verbosity level for diagnostic messages is `Concise`. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Corresponding Enumeration Member	Verbosity Description
1	`matlab.unittest.Verbosity.Terse`	minimal amount of information
2	`matlab.unittest.Verbosity.Concis`	typical amount of information
3	`matlab.unittest.Verbosity.Detail`	supplemental amount of information
4	`matlab.unittest.Verbosity.Verbos`	surplus of information

Examples

Log Diagnostic Information

Create a function-based test in a file, `sampleLogTest.m`, in your working folder.

```
function tests = sampleLogTest
tests = functiontests(localfunctions);

function svdTest(testCase)
import matlab.unittest.Verbosity

log(testCase,'Generating matrix.');
m = rand(1000);

log(testCase,1,'About to call SVD.');
[U,S,V] = svd(m);

log(testCase,Verbosity.Terse,'SVD finished.');

verifyEqual(testCase,U*S*V',m,'AbsTol',1e-6)
```

At the command prompt, run the test.

```
results = run(sampleLogTest);
```

```
Running sampleLogTest
   [Terse] Diagnostic logged (2014-04-14T14:20:59): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:20:59): SVD finished.
.
Done sampleLogTest
__________
```

The default runner reports the diagnostics at level 1 (`Terse`).

Create a test runner to report the diagnostics at levels 1 and 2, and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(2);
runner.addPlugin(p);

results = runner.run(sampleLogTest);
```

```
 [Concise] Diagnostic logged (2014-04-14T14:28:14): Generating matrix.
   [Terse] Diagnostic logged (2014-04-14T14:28:14): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:28:15): SVD finished.
```

See Also

`matlab.unittest.plugins.LoggingPlugin` | `matlab.unittest.Verbosity`

Introduced in R2014b

run

Class: matlab.unittest.TestCase
Package: matlab.unittest

Run `TestCase` test

Syntax

```
result = run(testCase)
result = run(testCase,testMethod)
```

Description

`result = run(testCase)` uses testCase as a prototype to run a `TestSuite` array created from all test methods in the class defining testCase. This suite is run using a `TestRunner` object configured for text output.

`result = run(testCase,testMethod)` uses testCase as a prototype to run a `TestSuite` array created from testMethod. This test is run using a `TestRunner` object configured for text output.

This is a convenience method to allow interactive experimentation of `TestCase` classes in MATLAB, yet running the tests contained in them using a supported `TestRunner` object.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

Default:

`testMethod`

Name of desired test method, specified as one of the following:

- string
- `meta.method` instance

The method must correspond to a valid `Test` method of the testCase instance.

Default:

Output Arguments

`result`

A `matlab.unittest.TestResult` object containing the result of the test run.

Examples

Run Test Directly from Test Case

Add the `FigurePropertiesTest.m` test case file to a folder on your MATLAB path.

```
classdef FigurePropertiesTest < matlab.unittest.TestCase

    properties
        TestFigure
    end

    methods(TestMethodSetup)
        function createFigure(testCase)
            % comment
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    end

    methods(Test)
```

```
        function defaultCurrentPoint(testCase)

            cp = testCase.TestFigure.CurrentPoint;
            testCase.verifyEqual(cp, [0 0], ...
                'Default current point is incorrect')
        end

        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty

            co = testCase.TestFigure.CurrentObject;
            testCase.verifyThat(co, IsEmpty, ...
                'Default current object should be empty')
        end

    end

end
```

Create a testcase object.

```
tc = FigurePropertiesTest;
```

Run the tests.

```
tc.run;
```

```
Running FigurePropertiesTest
..
Done FigurePropertiesTest
__________
```

All tests passed.

See Also

`matlab.unittest.TestSuite.run` | `matlab.unittest.TestRunner.run`

matlab.unittest.TestResult class

Package: matlab.unittest

Result of running test suite

Description

The `matlab.unittest.TestResult` class holds the information describing the result of running a test suite using the `matlab.unittest` framework. The results include information describing whether the test passed, failed, or ran to completion, as well as the duration of each test.

Construction

`TestResult` arrays are created and returned by the test runner, and are of the same size as the suite which was run.

Properties

`Duration`

Time elapsed running test.

The `Duration` property indicates the amount of time taken to run a particular test, including the time taken setting up and tearing down any test fixtures.

Fixture setup time is accounted for in the duration of the first test suite array element that uses the fixture. Fixture teardown time is accounted for in the duration of the last test suite array element that uses the fixture.

The total run time for a suite of tests exceeds the sum of the durations for all the elements of the suite because the `Duration` property does not include all the overhead of the `TestRunner` object, nor any of the time consumed by test runner plugins.

`Failed`

Logical value showing if test failed.

A TRUE `Failed` property indicates some form of test failure. When `Failed` is FALSE, then no failing conditions were encountered. A failing result can occur with a failure condition either in a test or in setting up and tearing down test fixtures. Failures can occur due to the following:

- Verification failures
- Assertion failures
- Uncaught MExceptions

Fatal assertions are also failing conditions, but in the event of a fatal assertion failure, the entire framework aborts and a `TestResult` object is never produced.

`Incomplete`

Logical value showing if test did not run to completion.

A TRUE `Incomplete` property indicates a test did not run to completion. When it is FALSE, then no conditions were encountered that prevented the test from completing. In other words, when FALSE there were no stack disruptions out of the running test content. An incomplete result can occur with a stack disruption in either a test or when setting up and tearing down test fixtures. Incomplete tests can occur due to the following:

- Assertion failures
- Tests filtered through assumption
- Uncaught MExceptions

Fatal assertions are also conditions that prevent the completion of tests, but in the event of a fatal assertion failure the entire framework aborts and a `TestResult` object is never produced.

`Name`

The name of the `TestSuite` object for the result.

The `Name` property is a string that holds the name of the test corresponding to this result.

`Passed`

Logical value showing if the test passed.

When the `Passed` property is TRUE, then the test completed as expected without any failure. When it is FALSE, then the test did not run to completion and/or encountered a failure condition.

Tips

- Create a table from the `TestResult` object for access to `table` functionality such as sorting rows, displaying a summary, and writing the table to a file. For example,

  ```
  rt = table(results);
  ```

Examples

Use TestResult Object to Identify and Rerun Failed Tests

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;
```

Result display method provides a summary of the results

```
suite = TestSuite.fromClass(?SomeTestClass)
results = run(suite)

results =

  matlab.unittest.TestResult

  Test Suite Summary:
    12 Passed, 4 Failed, 0 Not Run To Completion.
    5.5091 seconds testing time.
```

Rerun only the failed tests.

```
failedTests = suite([results.Failed]);
failedResults = run(failedTests)

failedResults =

  matlab.unittest.TestResult

  Test Suite Summary:
    0 Passed, 4 Failed, 0 Not Run To Completion.
    1.2894 seconds testing time.
```

Make the fix and rerun results.

```
newResults = run(failedTests)

newResults =

  matlab.unittest.TestResult

  Test Suite Summary:
    4 Passed, 0 Failed, 0 Not Run To Completion.
    1.1607 seconds testing time.
```

See Also

`table` | `TestRunner` | `TestSuite`

More About

- Property Attributes

matlab.unittest.TestRunner class

Package: matlab.unittest

Class for running tests in `matlab.unittest` framework

Description

The `matlab.unittest.TestRunner` class is the fundamental API used to run a suite of tests in the `matlab.unittest` framework. It runs and operates on `TestSuite` arrays. Use this class to customize running tests.

The `TestRunner` class is a sealed class; you cannot derive classes from the `TestRunner` class.

Construction

To create a simple, silent `TestRunner` object, call the static `withNoPlugins` method.

```
runner = matlab.unittest.TestRunner.withNoPlugins
```

To create a `TestRunner` object to run tests from the MATLAB Command Window, call the static `withTextOutput` method.

```
runner = matlab.unittest.TestRunner.withTextOutput
```

To create a customized `TestRunner` object, call the `addPlugin` method.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(SomePlugin())
```

Properties

`PrebuiltFixtures` — Fixtures set up outside the test runner

scalar `Fixture` instance | row vector of `Fixture` instances

Fixtures that are set up outside the test runner, specified as a scalar or row vector of `matlab.unittest.fixture.Fixture` instances. Use this property to specify that the

environmental configuration is performed manually instead of automatically during fixture setup and teardown.

The test runner considers these fixtures as already set up and never attempts to set up or tear down any fixtures specified by the `PrebuiltFixtures` property. If a test suite requires a shared test fixture and that test fixture is specified as a prebuilt fixture, the test runner does not perform set up or tear down actions.

Note: The test runner uses a prebuilt fixture only if it is specified by the `PrebuiltFixtures` property and is listed as a `SharedTestFixture` in the test class definition. The test runner does not use a prebuilt fixture if the fixture is registered using the `TestCase.applyFixture` method.

Methods

addPlugin	Add plugin to `TestRunner` object
run	Run all tests in `TestSuite` array
runInParallel	Run all tests in `TestSuite` array in parallel
withNoPlugins	Create simplest runner possible
withTextOutput	Create `TestRunner` object for command window output

Examples

Create TestRunner Object Configured for Text Output

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
```

```
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create the `TestRunner` object and run the suite.

```
runner = TestRunner.withTextOutput;
result = run(runner,suite);
```

Include Prebuilt Fixture

This example uses a shared test fixture and then specifies the fixture as prebuilt. The test runner does not set up and tear down the prebuilt fixture. Since the test assumes that the fixture exists, you must manually perform the setup work that the fixture ordinarily performs.

Create a test class in a file in your working folder. The test class uses a `PathFixture` as a shared test fixture. This example assumes that the subfolder, `helperFiles`, exists in your working folder.

```
classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.PathFixture('helperFiles')}) ...
        SampleTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            f = testCase.getSharedTestFixtures;

            import matlab.unittest.constraints.ContainsSubstring
            testCase.assertThat(path,ContainsSubstring(f.Folder))
        end
    end
end
```

Create a test suite and test runner at the command prompt.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?SampleTest);
runner = TestRunner.withTextOutput;
```

Run the tests using the shared test fixture. In this case, the fixture is not prebuilt.

```
runner.run(suite);
```

```
Setting up PathFixture
Done setting up PathFixture: Added 'C:\Work\helperFiles' to the path.
__________

Running SampleTest
.
Done SampleTest
__________

Tearing down PathFixture
Done tearing down PathFixture: Restored the path to its original state.
__________
```

The test runner sets up and tears down the shared test fixture.

Create an instance of the fixture and add it to the test runner.

```
f = matlab.unittest.fixtures.PathFixture('helperFiles');
runner.PrebuiltFixtures = f;
```

Manually add the `'helperFiles'` folder to your path. The `PathFixture` adds the specified folder to your path, and the tests rely on this setup action. However, since the fixture is defined as prebuilt, the test runner does not perform set up or tear down actions, and you must perform them manually. In this case, if you do not manually add it to your path, the test fails.

```
p = fullfile(pwd,'helperFiles');
oldPath = addpath(p);
```

Run the tests.

```
runner.run(suite);
```

```
Running SampleTest
.
Done SampleTest
__________
```

The test runner assumes that the fixture is prebuilt and does not set it up or tear it down.

Manually reset your path.

```
path(oldPath)
```

- “Write Tests Using Shared Fixtures”

See Also

`TestResult` | `TestSuite`

Introduced in R2013a

addPlugin

Class: matlab.unittest.TestRunner
Package: matlab.unittest

Add plugin to `TestRunner` object

Syntax

```
addPlugin(runner,plugin)
```

Description

`addPlugin(runner,plugin)` adds plugin to runner.

Input Arguments

`runner`

`matlab.unittest.TestRunner` object.

Default:

`plugin`

Mechanism provided to customize the manner in which a `TestSuite` array is run, specified as a `TestRunnerPlugin` object.

Default:

Examples

Run Test with Custom Plugin

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a `TestRunner` object.

```
runner = TestRunner.withNoPlugins;
```

Add a custom plugin.

```
import matlab.unittest.plugins.DiagnosticsValidationPlugin;
runner.addPlugin(DiagnosticsValidationPlugin);
```

Run the test.

```
result = run(runner,suite);
```

run

Class: matlab.unittest.TestRunner
Package: matlab.unittest

Run all tests in `TestSuite` array

Syntax

```
result = run(runner,suite)
```

Description

`result = run(runner,suite)` runs the `TestSuite` array defined by suite using the `TestRunner` object provided in runner, and returns the result in result.

This method runs all of the appropriate methods of the `TestCase` class to set up fixtures and run test content. It handles errors and qualification failures and records the information in result.

Input Arguments

`runner`

`matlab.unittest.TestRunner` object.

Default:

`suite`

Set of tests, specified as a `matlab.unittest.TestSuite` array.

Output Arguments

`result`

A `matlab.unittest.TestResult` object containing the result of the test run. result is the same size as suite and each element is the result of the corresponding element in suite.

Examples

Run All Tests in a Package

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a test suite, and a test runner that displays text.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
runner = TestRunner.withTextOutput;
```

Run the test suite.

```
result = runner.run(suite)
```

See Also

`matlab.unittest.TestSuite.run` | `matlab.unittest.TestCase.run`

runInParallel

Class: matlab.unittest.TestRunner
Package: matlab.unittest

Run all tests in `TestSuite` array in parallel

Syntax

```
result = runInParallel(runner,suite)
```

Description

`result = runInParallel(runner,suite)` runs all tests in the `TestSuite` array in parallel and returns the results in a `TestResult` object. The `runInParallel` method divides `suite` into separate groups and uses `runner` to run each group on the current parallel pool.

Note: The `runInParallel` method requires the Parallel Computing Toolbox. The testing framework might vary the order and number of groups or which tests it includes in each group.

When you select a test suite to run in parallel, consider possible resource contention. For example, if your test fixtures access global resources, such as a shared file on the same network, the parallel sessions could conflict with each other. In such cases, consider using a prebuilt shared test fixture.

Input Arguments

`runner` — Test runner for parallel test groups

`matlab.unittest.TestRunner` instance

Test runner for parallel test groups, specified as a `matlab.unittest.TestRunner` instance.

Consider your test runner configuration before running tests in parallel. Since the `runInParallel` method runs separate groups of tests on different workers, some plugins, such as `StopOnFailuresPlugin` or `CodeCoveragePlugin`, are not well suited for parallelization. The framework does not support running parallelized tests using a test runner with a custom plugin.

`suite` — Set of tests to run in parallel
`matlab.unittest.Test` array

Set of tests to run in parallel, specified as a `matlab.unittest.Test` array.

Examples

Run Tests in Parallel

Create the following parameterized test in a file in your current working folder.

```
classdef TestRand < matlab.unittest.TestCase
    properties (TestParameter)
        dim1 = createDimensionSizes;
        dim2 = createDimensionSizes;
        dim3 = createDimensionSizes;
        type = {'single','double'};
    end

    methods (Test)
        function testRepeatable(testCase,dim1,dim2,dim3)
            state = rng;
            firstRun = rand(dim1,dim2,dim3);
            rng(state)
            secondRun = rand(dim1,dim2,dim3);
            testCase.verifyEqual(firstRun,secondRun);
        end
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end

function sizes = createDimensionSizes
% Create logarithmicly spaced sizes up to 100
sizes = num2cell(round(logspace(0,2,10)));
end
```

At the command prompt, create a suite from `TestRand.m` and a test runner that displays text in the Command Window.

```
suite = matlab.unittest.TestSuite.fromClass(?TestRand);
runner = matlab.unittest.TestRunner.withTextOutput();
```

The suite contains 1200 test elements.

Run the test suite in parallel.

```
result = runInParallel(runner,suite)
```

```
Split tests into 12 groups and running them on 4 workers.
----------------
Finished Group 2
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
.........
Done TestRand
__________


----------------
Finished Group 4
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
```

```
..........
.....
Done TestRand
__________


---------------
Finished Group 3
---------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
.......
Done TestRand
__________


---------------
Finished Group 1
---------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
..
Done TestRand
__________
```

```
----------------
Finished Group 7
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
.........
Done TestRand
__________


----------------
Finished Group 5
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
...
Done TestRand
__________


----------------
Finished Group 6
----------------
Running TestRand
..........
..........
```

```
..........
..........
..........
..........
..........
..........
..........
..........
.
Done TestRand
__________


----------------
Finished Group 8
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
.......
Done TestRand
__________


-----------------
Finished Group 11
-----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
```

```
.
Done TestRand
__________


-----------------
Finished Group 12
-----------------
Running TestRand
.........
.........
.........
.........
.........
.........
.........
.........
.........
.......
Done TestRand
__________


-----------------
Finished Group 10
-----------------
Running TestRand
.........
.........
.........
.........
.........
.........
.........
.........
.........
...
Done TestRand
__________


---------------
Finished Group 9
---------------
Running TestRand
```

```
..........
..........
..........
..........
..........
..........
..........
..........
..........
.....
Done TestRand
__________


result =

  1200x1 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration

Totals:
   1200 Passed, 0 Failed, 0 Incomplete.
   11.4023 seconds testing time.
```

- “Write Tests Using Shared Fixtures”

See Also

`matlab.unittest.TestRunner.run` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.TestResult` | `matlab.unittest.plugins.TestRunnerPlugin`

More About

- “Key Problems Addressed by Parallel Computing”

Introduced in R2015a

matlab.unittest.TestRunner.withNoPlugins

Class: matlab.unittest.TestRunner
Package: matlab.unittest

Create simplest runner possible

Syntax

```
runner = matlab.unittest.TestRunner.withNoPlugins
```

Description

`runner = matlab.unittest.TestRunner.withNoPlugins` creates a TestRunner that is guaranteed to have no plugins installed and returns it in runner. It is the method one can use to create the simplest runner possible without violating the guarantees a test writer has when writing TestCase classes. This runner is a silent runner, meaning that regardless of passing or failing tests, this runner produces no command window output, although the results returned after running a test suite are accurate.

This method can also be used when it is desirable to have complete control over which plugins are installed and in what order. It is the only method guaranteed to produce the minimal TestRunner with no plugins, so one can create it and add additional plugins as desired.

Output Arguments

`runner`

`matlab.unittest.TestRunner` object.

Default:

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Create a Silent TestRunner Object with no Plugins

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a `TestRunner` object.

```
runner = TestRunner.withNoPlugins;

% Run the suite silently
result = run(runner,suite)
```

Control Plugins

Using the `TestRunner` object created in the previous example, control which plugins are installed and in what order they are installed.

Add `matlab.unittest` class to the current import list.

```
import matlab.unittest.plugins;
```

Add specific plugins.

```
runner.addPlugin(DiagnosticsValidationPlugin);
runner.addPlugin(TestRunProgressPlugin.withVerbosity(2));
```

Rerun the tests.

```
result = run(runner,suite)
```

matlab.unittest.TestRunner.withTextOutput

Class: matlab.unittest.TestRunner
Package: matlab.unittest

Create `TestRunner` object for command window output

Syntax

```
runner = matlab.unittest.TestRunner.withTextOutput
runner = matlab.unittest.TestRunner.withTextOutput('Verbosity',v)
```

Description

`runner = matlab.unittest.TestRunner.withTextOutput` creates a `TestRunner` object that is configured for running tests from the MATLAB Command Window and returns it in runner. The output produced includes test progress as well as diagnostics in the event of test failures.

`runner = matlab.unittest.TestRunner.withTextOutput('Verbosity',v)` reacts to messages logged at or below the specified verbosity level and controls the level of detail about the test run that is displayed in the Command Window.

Input Arguments

v — Verbosity levels supported by test runner

`1 | 2 | 3 | 4 | matlab.unittest.Verbosity` enumeration

Verbosity levels supported by the test runner, specified as an integer value between 1 and 4 or a `matlab.unittest.Verbosity` enumeration object. The runner reacts to diagnostics that are logged at this level and lower. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Corresponding Enumeration Object	Verbosity Description
1	`matlab.unittest.Verbosity.Terse`	Minimal amount of information

Numeric Representation	Corresponding Enumeration Object	Verbosity Description
2	`matlab.unittest.Verbosity.Concise`	Typical amount of information
3	`matlab.unittest.Verbosity.Detailed`	Supplemental amount of information
4	`matlab.unittest.Verbosity.Verbose`	Surplus of information

Output Arguments

`runner`

`matlab.unittest.TestRunner` object.

Default:

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Display Test Results in Command Window

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a `TestRunner` object that produced output to the Command Window.

```
runner = TestRunner.withTextOutput;

% Run the suite
result = run(runner,suite)
```

Specify Output Verbosity

Create the follow class In a file in your current working folder, `ExampleLogTest.m`.

```
classdef ExampleLogTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            log(testCase,matlab.unittest.Verbosity.Detailed,'Starting Test')
            log(testCase,'Testing 5==5')
            testCase.verifyEqual(5,5)
            log(testCase,matlab.unittest.Verbosity.Verbose,'Test Complete')
        end
    end
end
```

At the command prompt, run the test.

```
result = run(ExampleLogTest);
```

```
Running ExampleLogTest
.
Done ExampleLogTest
__________
```

Create a test runner to display logged messages at verbosity level 4 and lower, and then run the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
suite = TestSuite.fromClass(?ExampleLogTest);
runner = TestRunner.withTextOutput('Verbosity',4);

results = runner.run(suite);
```

```
 Running ExampleLogTest
  Setting up ExampleLogTest
  Done setting up ExampleLogTest in 0 seconds
   Running ExampleLogTest/testOne
    Evaluating Test: testOne
[Detailed] Diagnostic logged (2014-04-18T14:28:19): Starting Test
```

```
 [Concise] Diagnostic logged (2014-04-18T14:28:19): Testing 5==5
 [Verbose] Diagnostic logged (2014-04-18T14:28:20): Test Complete
   Done ExampleLogTest/testOne in 0.21106 seconds
  Tearing down ExampleLogTest
  Done tearing down ExampleLogTest in 0 seconds
 Done ExampleLogTest in 0.21106 seconds
```

See Also

matlab.unittest.Verbosity | matlab.unittest.fixtures.Fixture.log | matlab.unittest.TestCase.log | run

matlab.unittest.TestSuite class

Package: matlab.unittest

Class for grouping tests to run

Description

The `matlab.unittest.TestSuite` class is the fundamental interface used to group and run a set of tests in the unit test framework. The `matlab.unittest.TestRunner` object can only run arrays of `TestSuite` objects.

Construction

`TestSuite` arrays are created using static methods of the `TestSuite` class. These methods may return subclasses of the `TestSuite` class depending on the method call and context.

Methods

fromClass	Create suite from `TestCase` class
fromFile	Create `TestSuite` array from test file
fromFolder	Create `TestSuite` array from all tests in folder
fromMethod	Create `TestSuite` array from single test method
fromName	Create `Test` object from name of test element

fromPackage	Create `TestSuite` array from all tests in package
run	Run `TestSuite` array using `TestRunner` object configured for text output
selectIf	Select test suite elements that satisfy conditions

Examples

Create Test Suite of Every Type of Test Set

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;
```

Create test suites using each method.

```
fileSuite    = TestSuite.fromFile('SomeTestFile.m');
folderSuite  = TestSuite.fromFolder(pwd);
packageSuite = TestSuite.fromPackage('mypackage.subpackage');
classSuite   = TestSuite.fromClass(?mypackage.MyTestClass);
methodSuite  = TestSuite.fromMethod(?SomeTestClass,'testMethod');
```

Concatenate the suites.

```
largeSuite = [fileSuite, folderSuite, packageSuite, classSuite, methodSuite];
```

Run the full suite.

```
result = run(largeSuite)
```

See Also

`Test` | `TestResult` | `TestRunner`

matlab.unittest.TestSuite.fromClass

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Create suite from `TestCase` class

Syntax

```
suite = matlab.unittest.TestSuite.fromClass(testClass)
suite = matlab.unittest.TestSuite.fromClass(testClass,s)
suite = matlab.unittest.TestSuite.fromClass( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromClass(testClass)` creates a `TestSuite` array from all of the `Test` methods contained in testClass and returns that array in suite.

`suite = matlab.unittest.TestSuite.fromClass(testClass,s)` creates a `TestSuite` array from all of the `Test` methods contained in testClass that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromClass( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more Name,Value pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Tips

- testClass must be on the MATLAB path when using this method to create suite, as well as when suite is run.

Input Arguments

`testClass`

Class containing test methods, specified as a `meta.class` instance. Use the `?` operator to create a `meta.class` instance. testClass must derive from `matlab.unittest.TestCase`.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`'Name'`

String indicating the name of the suite element. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterProperty'`

String indicating the name of a property that defines a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterName'`

String indicating the name of a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'BaseFolder'`

String indicating the name of the folder that contains the file defining the test class or function. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one

character. For test classes defined in packages, the base folder is the parent of the top-level package folder.

`'Tag'`

String indicating the name of the tag applied to the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run Tests in a Package Class

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;
```

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
result = run(suite)
```

Run Tests in a Class Without a Package

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;
```

```
suite = TestSuite.fromClass(?MyTestClass);
result = run(suite)
```

Create Suite of Test Elements Using Selector

In your working folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from the test elements that tests the `'double'` data type.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;

suite = TestSuite.fromClass(?testZeros, ...
```

```
    HasParameter('Property','type','Name','double'));
{suite.Name}'

ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `testZeros.m` class from the previous example.

At the command prompt, create a test suite from the test elements that tests the `'double'` data type.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?testZeros, ...
    'ParameterProperty','type', 'ParameterName','double');
{suite.Name}'

ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

See Also

`fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `TestRunner`

Introduced in R2013a

matlab.unittest.TestSuite.fromFile

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Create `TestSuite` array from test file

Syntax

```
suite = matlab.unittest.TestSuite.fromFile(file)
suite = matlab.unittest.TestSuite.fromFile(file,s)
suite = matlab.unittest.TestSuite.fromFile( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromFile(file)` creates a `TestSuite` array from all of the tests in file. When the test suite is run, MATLAB changes the current folder to the folder that defines the test content, and adds it to the path for the duration of the test run.

`suite = matlab.unittest.TestSuite.fromFile(file,s)` creates a `TestSuite` array from all of the tests in file that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromFile( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more Name,Value pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`file`

Test file, specified as a string.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Name'`

String indicating the name of the suite element. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterProperty'`

String indicating the name of a property that defines a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterName'`

String indicating the name of a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'BaseFolder'`

String indicating the name of the folder that contains the file defining the tests. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character. For tests defined in packages, the base folder is the parent of the top-level package folder.

`'Tag'`

String indicating the name of the tag applied to the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run Tests in Class File

Function for unit testing:

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

`TestCase` class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double')
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
    end
```

```
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m')
```

```
result = run(suite);
```

```
Running Add5Test
..
Done Add5Test
__________
```

Create Suite of Test Elements Using Selector

In your working folder, create `testZeros.m`. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized test methods in `testZeros.m` using the `HasParameter` selector.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;

suite = TestSuite.fromFile('testZeros.m', HasParameter)
```

```
suite =

  1x8 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   5 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Create a test suite from only the test elements from the `testSize` method using the `HasName` selector with the `StartsWithSubstring` constraint.

```
import matlab.unittest.selectors.HasName;
import matlab.unittest.constraints.StartsWithSubstring;

suite = TestSuite.fromFile('testZeros.m',...
    HasName(StartsWithSubstring('testZeros/testSize')));
{suite.Name}'
```

```
ans =

    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
```

The test suite contains the two parameterized tests from the `testSize` method.

Create Suite of Test Elements Using Name-Value Arguments

Create the `testZeros.m` class from the previous example.

At the command prompt, create a test suite from all test methods in `testZeros.m` that have a name starting with `'testZeros/testSize'`. This test suite contains parameterized tests from the `testSize` method.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFile('testZeros.m', 'Name', 'testZeros/testSize*');
{suite.Name}'

ans =

    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
```

To ensure that a test suite is comprised of test elements associated with one particular test method, use the `fromMethod` method of `TestSuite`.

At the command prompt, create a test suite from all test methods in `testZeros.m` that have a name ending in `'Size'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFile('testZeros.m', 'Name', '*Size');
{suite.Name}'

ans =

    'testZeros/testDefaultSize'
```

Note that elements from the `testSize` method are not included in the test suite. The name of these elements contains information about the parameterization, and therefore it does not end with `'Size'`.

Create a test suite of all tests that use the parameter name `'double'`.

```
suite = TestSuite.fromFile('testZeros.m', 'ParameterName', 'double');
{suite.Name}'

ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

To construct the same test suite using selectors, use `suite = TestSuite.fromFile('testZeros.m', HasParameter('Name','double'))`.

See Also

`fromFolder` | `matlab.unittest.selectors` | `TestRunner`

Introduced in R2013a

matlab.unittest.TestSuite.fromFolder

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Create `TestSuite` array from all tests in folder

Syntax

```
suite = matlab.unittest.TestSuite.fromFolder(folder)
suite = matlab.unittest.TestSuite.fromFolder(folder,s)
suite = matlab.unittest.TestSuite.fromFolder( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromFolder(folder)` creates a `TestSuite` array from all of the `Test` methods of all concrete `TestCase` classes contained in folder and returns that array in suite. If tests are function-based or script-based, a `Test` file is included in the `TestSuite` array if it follows the naming convention of starting or ending in the word 'test', which is case-insensitive. Class-based tests do not need to follow this naming convention. The method is not recursive, returning only those tests directly in the specified folder.

When the test suite is run, MATLAB changes the current folder to the folder that defines the test content, and adds it to the path for the duration of the test run.

`suite = matlab.unittest.TestSuite.fromFolder(folder,s)` creates a `TestSuite` array from all of the `Test` methods contained in folder that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromFolder( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more Name,Value pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`folder`

Folder containing tests, specified as a string. folder can be either an absolute or relative path to the desired folder.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`'IncludingSubfolders'`

Indicator for whether to include tests from any folder subfolders, excluding package, class, and private folders, specified as `false` or `true` (`logical 0` or `1`). This property is `false` by default. You can specify it as `true` during construction.

`'Name'`

String indicating the name of the suite element. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterProperty'`

String indicating the name of a property that defines a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterName'`

String indicating the name of a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'BaseFolder'`

String indicating the name of the folder that contains the file defining the tests. For a test element to be included in the suite, the test element must be contained in the

specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character. For tests defined in packages, the base folder is the parent of the top-level package folder.

`'Tag'`

String indicating the name of the tag applied to the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run Tests in Current Folder

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFolder(pwd);
result = run(suite);
```

Run Tests in Subfolders

```
suite = TestSuite.fromFolder(pwd, 'IncludingSubfolders', true);
```

```
result = run(suite);
```

Create Suite of Test Elements Using Selector

In your working folder, create a new folder, `myTests`. In that folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized tests that use the parameter name `'double'`.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;

suite = TestSuite.fromFolder('myTests', HasParameter('Name','double'));
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `myTests` folder and `testZeros.m` class from the previous example.

Create a test suite of all tests that use the parameter name `'double'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFolder('myTests', 'ParameterName', 'double');
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

See Also

`fromFile` | `matlab.unittest.selectors` | `TestRunner`

Introduced in R2013a

matlab.unittest.TestSuite.fromMethod

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Create `TestSuite` array from single test method

Syntax

```
suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod)
suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod,s)
suite = matlab.unittest.TestSuite.fromMethod( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod)` creates a `TestSuite` array from the test class described by testClass and the test method described by testMethod and returns it in suite.

`suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod,s)` creates a `TestSuite` array from all of the `Test` methods contained in testMethod that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromMethod( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more Name,Value pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Tips

- testClass must be on the MATLAB path when using this method to create suite, as well as when suite is run.

Input Arguments

`testClass`

Class describing the test methods, specified as a `meta.class` instance which must derive from `matlab.unittest.TestCase`.

`testMethod`

Test method, specified by either the `meta.method` instance or the name as a string. The method must be defined with a `Test` method attribute.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`'Name'`

String indicating the name of the suite element. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterProperty'`

String indicating the name of a property that defines a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterName'`

String indicating the name of a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'BaseFolder'`

String indicating the name of the folder that contains the file defining the test class or function. For a test element to be included in the suite, the test element must be

contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character. For test classes defined in packages, the base folder is the parent of the top-level package folder.

`'Tag'`

String indicating the name of the tag applied to the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run a Single Test Method

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;

cls = ?mypackage.MyTestClass;

% Create the suite using the method name
suite = TestSuite.fromMethod(cls, 'testMethod');
result = run(suite)
```

```
% Create the suite using the meta.method instance
metaMethod = findobj(cls.MethodList, 'Name', 'testMethod');
suite = TestSuite.fromMethod(cls, metaMethod);
result = run(suite)
```

Create Suite of Test Elements Using Selector

In your working folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized tests from the `testClass` method that use the parameter name `'single'`.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;
```

```
suite = TestSuite.fromMethod(?testZeros,'testClass', ...
    HasParameter('Name','single'));
{suite.Name}'

ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `testZeros.m` class from the previous example.

At the command prompt, create a test suite from all parameterized tests from the `testClass` method that use the parameter name `'single'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromMethod(?testZeros,'testClass', ...
    'ParameterName','single');
{suite.Name}'

ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
```

See Also

`fromClass` | `fromPackage` | `matlab.unittest.selectors` | `TestRunner`

Introduced in R2013a

matlab.unittest.TestSuite.fromName

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Create `Test` object from name of test element

Syntax

```
testObj = matlab.unittest.TestSuite.fromName(name)
```

Description

`testObj = matlab.unittest.TestSuite.fromName(name)` creates a scalar `Test` object, `testObj`, from the name of the test element, name.

Input Arguments

`name`

Name of the `matlab.unittest.Test` element, specified as a string. For class-based tests, `name` contains the name of the `TestCase` class and the test method, as well as information about parameterization. For function-based tests, `name` contains the name of the main function and the local test function. For script-based tests, `name` contains the name of the script and the title of the test section or cell. If the section does not have a title, MATLAB assigns one. The `name` argument corresponds to the `Name` property of the `Test` object.

The test class, function or script described by `name` must be on the MATLAB path when you are creating and running the `TestSuite`.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Create and Run Test from Test Suite Element

Create a function to test, `add5`, in a file on your MATLAB path.

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

Create a file, `Add5Test.m`, on your MATLAB path that contains the following `TestCase` class.

```
classdef Add5Test < matlab.unittest.TestCase
    properties (TestParameter)
        Type = {'double','single','int8','int32'};
    end

    methods (Test)
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
        function testResultType(testCase, Type)
            actOutput = add5(cast(1,Type));
            testCase.verifyClass(actOutput, Type)
        end

    end
end
```

At the command prompt, create a test object for the `testNonNumericInput` method in the `Add5Test` class.

```
import matlab.unittest.TestSuite
testObj = TestSuite.fromName('Add5Test/testNonNumericInput');
```

Run the test

```
result = run(testObj);
```

```
Running Add5Test
.
Done Add5Test
__________
```

Create a parameterized test for the `testResultType` method in the `Add5Test` class, and run the test.

```
testObj = TestSuite.fromName('Add5Test/testResultType(Type=single)');
result = run(testObj);
```

```
Running Add5Test
.
Done Add5Test
__________
```

See Also

`TestRunner`

matlab.unittest.TestSuite.fromPackage

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Create `TestSuite` array from all tests in package

Syntax

```
suite = matlab.unittest.TestSuite.fromPackage(package)
suite = matlab.unittest.TestSuite.fromPackage(package,s)
suite = matlab.unittest.TestSuite.fromPackage( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromPackage(package)` creates a `TestSuite` array from all of the `Test` methods of all the tests contained in package and returns that array in suite. The method is not recursive, returning only those tests directly in the package specified.

`suite = matlab.unittest.TestSuite.fromPackage(package,s)` creates a `TestSuite` array from all the tests contained in package that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromPackage( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more Name,Value pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Tips

- The root folder(s) where package is defined must be on the MATLAB path when creating suite using this method as well as when suite is run.

Input Arguments

`package`

The name of the desired package to find tests, specified as a string.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`'IncludingSubpackages'`

Indicator for whether to include subpackages in the `TestSuite` array, specified as `false` or `true` (`logical 0` or `1`). This property is `false` by default. You can specify it as `true` during construction.

`'Name'`

String indicating the name of the suite element. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterProperty'`

String indicating the name of a property that defines a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterName'`

String indicating the name of a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'BaseFolder'`

String indicating the name of the folder that contains the tests. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use

the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character. For tests defined in packages, the base folder is the parent of the top-level package folder.

`'Tag'`

String indicating the name of the tag applied to the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run All Tests in a Package

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;
```

```
suite = TestSuite.fromPackage('mypackage.subpackage');
result = run(suite)
```

Run tests in `mypackage` including all subpackages.

```
suite = TestSuite.fromPackage('mypackage','IncludingSubpackages',true);
```

```
result = run(suite)
```

Create Suite of Test Elements Using Selector

In your working folder, create a new package by creating a new folder, `+myPackage`. In that folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized tests that use the parameter property `'outSize'`.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;
```

```
suite = TestSuite.fromPackage('myPackage', ...
    HasParameter('Property','outSize'));
{suite.Name}'

ans =

    'myPackage.testZeros/testClass(type=single,outSize=s2d)'
    'myPackage.testZeros/testClass(type=single,outSize=s3d)'
    'myPackage.testZeros/testClass(type=double,outSize=s2d)'
    'myPackage.testZeros/testClass(type=double,outSize=s3d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s2d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s3d)'
    'myPackage.testZeros/testSize(outSize=s2d)'
    'myPackage.testZeros/testSize(outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `+myPackage` folder and `testZeros.m` class from the previous example.

At the command prompt, create a test suite from all parameterized tests that use the parameter property `'outSize'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromPackage('myPackage', ...
    'ParameterProperty', 'outSize');
{suite.Name}'

ans =

    'myPackage.testZeros/testClass(type=single,outSize=s2d)'
    'myPackage.testZeros/testClass(type=single,outSize=s3d)'
    'myPackage.testZeros/testClass(type=double,outSize=s2d)'
    'myPackage.testZeros/testClass(type=double,outSize=s3d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s2d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s3d)'
    'myPackage.testZeros/testSize(outSize=s2d)'
    'myPackage.testZeros/testSize(outSize=s3d)'
```

See Also

`fromClass` | `fromMethod` | `matlab.unittest.selectors` | `TestRunner`

Introduced in R2013a

run

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Run `TestSuite` array using `TestRunner` object configured for text output

Syntax

```
result = run(suite)
```

Description

`result = run(suite)` runs the `TestSuite` object defined by suite using a `TestRunner` object configured for text output.

Tips

- This is a convenience method which is equivalent to using a `TestRunner` object created from the `TestRunner.withTextOutput` method to run suite.

Input Arguments

`suite`

Set of tests, specified as a `matlab.unittest.TestSuite` array.

Output Arguments

`result`

A `matlab.unittest.TestResult` object containing the result of the test run. result is the same size as suite and each element is the result of the corresponding element in suite.

Examples

Compare TestSuite.run with TestRunner.run

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a test suite and a test runner.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
runner = TestRunner.withTextOutput;
```

The following test results are equivalent.

```
result = runner.run(suite)
result = run(suite)
```

See Also

`matlab.unittest.TestRunner.run` | `matlab.unittest.TestCase.run`

selectIf

Class: matlab.unittest.TestSuite
Package: matlab.unittest

Select test suite elements that satisfy conditions

Syntax

```
newsuite = selectIf(suite,s)
newsuite = selectIf(suite,Name,Value)
```

Description

`newsuite = selectIf(suite,s)` selects from `suite` the test elements that satisfy the conditions specified by the selector, `s`, and returns them in the `TestSuite` array, `newsuite`.

`newsuite = selectIf(suite,Name,Value)` creates a `TestSuite` array with additional options specified by one or more Name,Value pair arguments.

Input Arguments

`suite`

Set of tests, specified as a `matlab.unittest.TestSuite` array.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Name'`

String indicating the name of the suite element. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterProperty'`

String indicating the name of a property that defines a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'ParameterName'`

String indicating the name of a parameter used by the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`'BaseFolder'`

String indicating the name of the folder that contains the tests. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character. For tests defined in packages, the base folder is the parent of the top-level package folder.

`'Tag'`

String indicating the name of the tag applied to the test suite element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Examples

Select Test Elements Using Selector

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef (SharedTestFixtures={...
        matlab.unittest.fixtures.PathFixture(fullfile(...
        matlabroot, 'help', 'techdoc', 'matlab_oop', 'examples'))})...
        ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
        function testOne(testCase)
            % test code
        end
         function testTwo(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasSharedTestFixture;
import matlab.unittest.selectors.HasName;
import matlab.unittest.fixtures.PathFixture;
import matlab.unittest.constraints.EndsWithSubstring;
import matlab.unittest.constraints.ContainsSubstring;

suite = TestSuite.fromClass(?ExampleTest)
```

```
suite =

  1x3 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 1 Unique Shared Test Fixture Class, 0 Tags.
```

The test suite contains three test elements.

Create a filtered test suite of tests comprising tests with names that contain the case-insensitive substring `'pAtH'`.

```
newSuite = selectIf(suite,HasName(ContainsSubstring('pAtH','IgnoringCase',true)))

newSuite =

  Test with properties:

                 Name: 'ExampleTest/testPathAdd'
      Parameterization: []
    SharedTestFixtures: [1x1 matlab.unittest.fixtures.PathFixture]
                 Tags: {0x1 cell}

Tests Include:
   0 Parameterizations, 1 Unique Shared Test Fixture Class, 0 Tags.
```

Only the `testPathAdd` test is part of the suite.

Alternatively, create the same suite using a name-value pair.

```
newSuite = selectIf(suite,'Name','*Path*');
```

However, unlike the `ContainsSubstring` constraint, the name-value pair does not have an option to ignore case.

Create a filtered suite of tests comprising tests that use a shared path fixture and do not have names ending with `'One'`.

```
newSuite = suite.selectIf(~HasName(EndsWithSubstring('One')) ...
    & HasSharedTestFixture(PathFixture(fullfile(matlabroot, 'help',...
    'techdoc', 'matlab_oop', 'examples'))));
{newSuite.Name}

ans =

    'ExampleTest/testPathAdd'    'ExampleTest/testTwo'
```

The test suite contains two tests. All of the tests use the specified path fixture, but the test named `'testOne'` is excluded from the suite.

Select Test Elements Using Parameterization

In your working folder, create `testZeros.m`. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
```

```
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from the file.

```
s = matlab.unittest.TestSuite.fromFile('testZeros.m');
{s.Name}'
```

```
ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s3d)'
    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
    'testZeros/testDefaultClass'
    'testZeros/testDefaultSize'
```

```
    'testZeros/testDefaultValue'
```

The suite contains 11 test elements. Six from the parameterized `testClass` method, two from the parameterized `testSize` method, and one from each of the `testDefaultClass`, `testDefaultSize`, and `testDefaultValue` methods.

Select all of the test elements from parameterized test methods.

```
import matlab.unittest.selectors.HasParameter;

s1 = s.selectIf(HasParameter);
{s1.Name}'

ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s3d)'
    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
```

The suite contains the eight test elements from the two parameterized test methods.

Select all of the test elements from non-parameterized test methods.

```
s2 = s.selectIf(~HasParameter);
{s2.Name}'

ans =

    'testZeros/testDefaultClass'
    'testZeros/testDefaultSize'
    'testZeros/testDefaultValue'
```

Select all test elements that are parameterized and have a property named `'type'` with a parameter name `'double'`.

```
s3 = s.selectIf('ParameterProperty','type', 'ParameterName','double');
{s3.Name}'

ans =
```

```
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

The resulting suite contains two elements. The `testClass` method is the only method in `testZeros` that uses the `'type'` property, and selecting only `'double'` from the parameters results in two test elements — one for each value of `'outSize'`.

Select all test elements that are parameterized and have a parameters defined by a property starting with `'t'`.

```
s4 = s.selectIf('ParameterProperty','t*');
{s4.Name}'
```

```
ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s3d)'
```

The resulting suite contains the six parameterized test elements from the `testClass` method. The `testSize` method is parameterized, but the elements from the method are not included in the suite because the method does not use a property that starts with `'t'`.

Select all test elements that are parameterized and test the `zeros` function with a `2-D` array. A parameter value representing a `2-D` array will have a length of `1` (e.g. `zeros(3)`) or `2` (e.g. `zeros(2,3)`).

```
import matlab.unittest.constraints.HasLength;

s5 = s.selectIf(HasParameter('Property','outSize',...
    'Value', HasLength(1)|HasLength(2)));
{s5.Name}'
```

```
ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=uint16,outSize=s2d)'
    'testZeros/testSize(outSize=s2d)'
```

Select only the test element that tests that the output is a `double` data type and the has the correct size for a `2-D` array.

```
s6 = s.selectIf(HasParameter('Property','type','Name','double')...
    & HasParameter('Property','outSize','Name','s2d'))
```

```
s6 =

  Test with properties:

                  Name: 'testZeros/testClass(type=double,outSize=s2d)'
      Parameterization: [1x2 matlab.unittest.parameters.TestParameter]
    SharedTestFixtures: []
                  Tags: {0x1 cell}

Tests Include:
   2 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Select Test Elements Using Tags

Create the following test class in a file, `ExampleTest.m`, in your current working folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit'})
        function testB (testCase)
            % test code
        end
        function testC (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit','FeatureA'})
        function testD (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'System','FeatureA'})
        function testE (testCase)
            % test code
        end
```

```
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class and examine the contents.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasTag

suite = TestSuite.fromClass(?ExampleTest)
```

```
suite =

  1x5 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 3 Unique Tags.
```

Click the hyperlink for `3 Unique Tags` to display all the tags in the suite.

```
       Tag
    __________

    'FeatureA'
    'System'
    'Unit'
```

Select all the test suite elements that have the tag `'Unit'`.

```
s1 = suite.selectIf(HasTag('Unit'))
```

```
s1 =

  1x3 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
    Tags
```

```
Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 2 Unique Tags.
```

Select all the test suite elements that do not contain the tag `'FeatureA'`.

```
s2 =  suite.selectIf(~HasTag('FeatureA'));
{s2.Name}

ans =

    'ExampleTest/testB'    'ExampleTest/testC'    'ExampleTest/testA'
```

Select all the test suite elements that have no tags.

```
s3 =  suite.selectIf(~HasTag)

s3 =

  Test with properties:

                 Name: 'ExampleTest/testA'
      Parameterization: []
    SharedTestFixtures: []
                 Tags: {0x1 cell}

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

See Also

`matlab.unittest.constraints` | `matlab.unittest.selectors`

Introduced in R2014a

texlabel

Format text into TeX string

Syntax

```
TeXString = texlabel(f)
TeXString = texlabel(f,'literal')
```

Description

`TeXString = texlabel(f)` converts the MATLAB expression `f` into the TeX equivalent for use in text strings. `texlabel` converts Greek variable names (for example, lambda, delta, and so on) into a string that is displayed as Greek letters. The `TeXString` output is useful as an argument to annotation functions such as `title`, `xlabel`, and `text`.

If `TeXString` is too long to fit into a figure window, then the center of the expression is replaced with a tilde ellipsis (~~~).

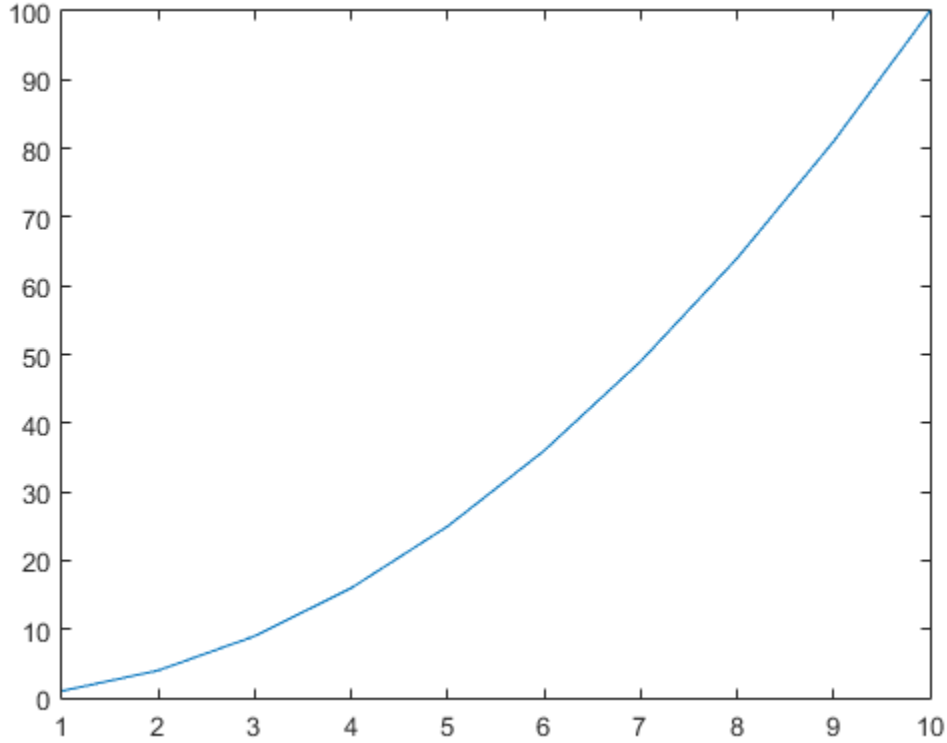
`TeXString = texlabel(f,'literal')` interprets Greek variable names literally.

Examples

Insert TeX String in Figure

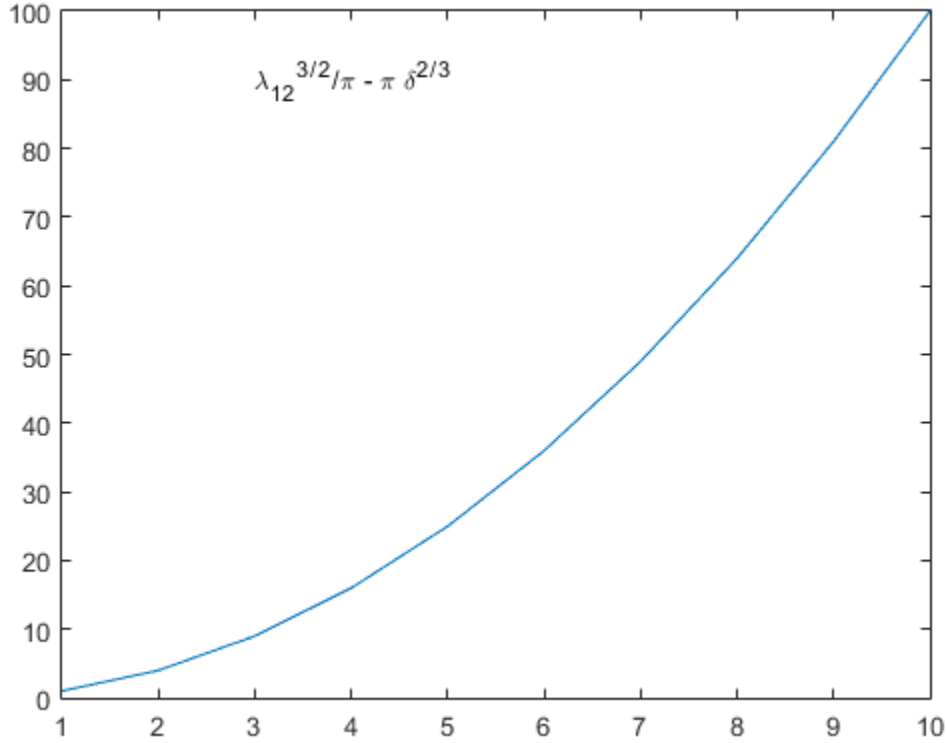
Create a figure and a TeX string for use in a text graphics object.

```
figure
plot((1:10).^2)
TeXString = texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)');
```



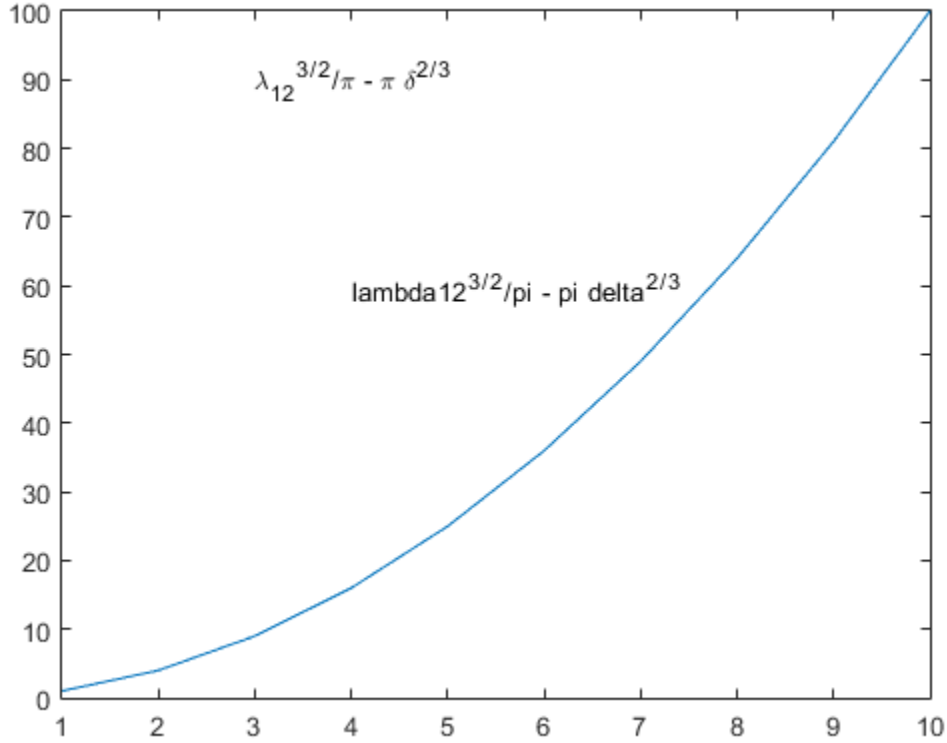
Add a text object containing the TeX string to the figure.

```
text(3,90,TeXString)
```



If you include the `'literal'` argument, `texlabel` interprets Greek variable names literally. Add a text object containing the literal string.

```
text(4,60,texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)','literal'))
```



Input Arguments

`f` — Input MATLAB expression

string

Input MATLAB expression, specified as a string.

Example: `'theta (degrees)'`

Data Types: `char`

See Also

`text` | `title` | `xlabel` | `ylabel` | `zlabel`

Introduced before R2006a

text

Create text object in current axes

Syntax

```
text(x,y,'string')
text(x,y,z,'string')
text(x,y,z,'string','PropertyName',PropertyValue....)
text('PropertyName',PropertyValue....)
h = text(...)
```

Properties

For a list of properties, see Text Properties.

Description

`text` is the low-level function for creating text graphics objects. Use `text` to place character strings at specified locations.

`text(x,y,'string')` adds the string in quotes to the location specified by the point `(x,y)` `x` and `y` must be numbers of class `double`.

`text(x,y,z,'string')` adds the string in 3-D coordinates. `x`, `y` and `z` must be numbers of class `double`.

`text(x,y,z,'string','PropertyName',PropertyValue....)` adds the string in quotes to the location defined by the coordinates and uses the values for the specified text properties. For a description of the properties, see Text Properties.

`text('PropertyName',PropertyValue....)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

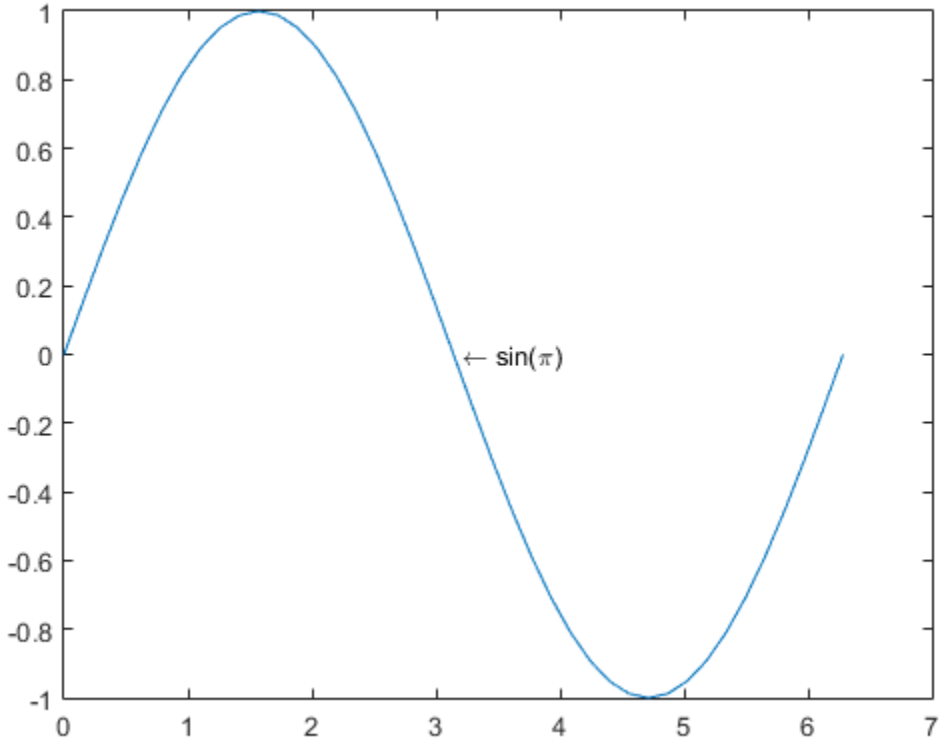
`h = text(...)` returns a column vector of handles to text objects, one handle per object. All forms of the `text` function optionally return this output argument.

Examples

Label Point on Graph

Annotate the point $(\pi, 0)$ with the string $\sin(\pi)$.

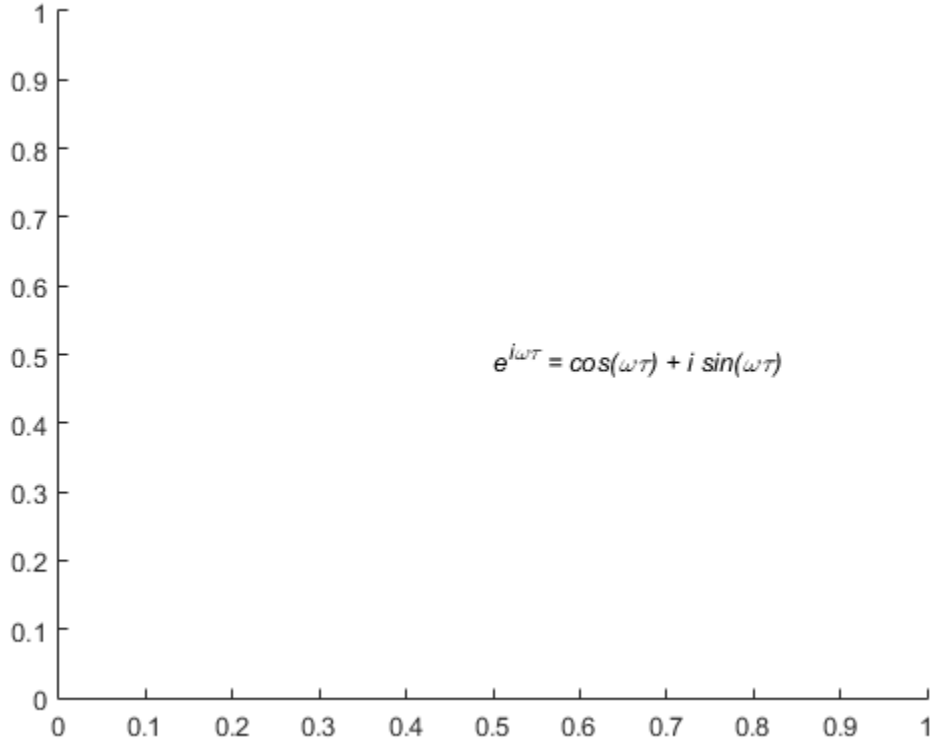
```
x = 0:pi/20:2*pi;
y = sin(x);

figure % new figure window
plot(x,y)
text(pi,0,' \leftarrow sin(\pi)')
```



Display Equation with Greek Symbols

Use embedded TeX sequences to display an equation in an empty axes.

```
axes
text(0.5,0.5,'\ite^{i\omega\tau} = cos(\omega\tau) + i sin(\omega\tau)')
```



Setting Default Properties

You can set default text properties on the axes, figure, and root levels:

```
set(0,'DefaulttextProperty',PropertyValue...)
set(gcf,'DefaulttextProperty',PropertyValue...)
set(gca,'DefaulttextProperty',PropertyValue...)
```

Where *`Property`* is the name of the text property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access text properties.

More About

Tips

Position Text Within the Axes

The default text units are the units used to plot data in the graph. Specify the text location coordinates (the `x`, `y`, and `z` arguments) in the data units of the current graph. You can use other units to position the text by setting the text `Units` property to `normalized` or one of the nonrelative units (`pixels`, `inches`, `centimeters`, `points`, or `characters`).

Note that the Axes Units property controls the positioning of the Axes within the figure and is not related to the axes data units used for graphing.

The `Extent`, `VerticalAlignment`, and `HorizontalAlignment` properties control the positioning of the character string with regard to the text location point.

If the coordinates are vectors, `text` writes the string at all locations defined by the list of points. If the character string is an array the same length as `x`, `y`, and `z`, `text` writes the corresponding row of the string array at each point specified.

Multiline Text

When specifying strings for multiple text objects, the string can be:

- A cell array of strings
- A padded string matrix

Each element of the specified string array creates a different text object.

When specifying the string for a single text object, cell arrays of strings and padded string matrices result in a text object with a multiline string, while vertical slash characters are not interpreted as separators and result in a single line string containing vertical slashes.

Behavior of the Text Function

`text` is a low-level function that accepts property name/property value pairs as input arguments. However, the convenience form:

```
text(x,y,z,'string')
```

is equivalent to:

```
text('Position',[x,y,z],'String','string')
```

You can specify other properties only as property name/property value pairs. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

`text` does not respect the setting of the figure or axes `NextPlot` property. This allows you to add text objects to an existing axes without setting `hold` to `on`.

See Also

Functions

`annotation` | `int2str` | `num2str` | `title` | `xlabel` | `ylabel` | `zlabel`

Properties

Text Properties

Introduced before R2006a

Text Properties

Control text appearance and behavior

Text properties control the appearance and behavior of a text object. By changing property values, you can modify certain aspects of the text.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = text(0.5,0.5,'text here');
s = h.FontSize;
h.FontSize = 12;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Text

`String` — Text to display

`''` (default) | character array | cell array | numeric value

Text to display, specified as a character array, cell array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: `123`

To include numeric variables with text, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the Interpreter property.

To create multiline text:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.

- Use `sprintf` to create a string with a new line character, such as `sprintf('first line \n second line')`. This property converts strings with new line characters to cell arrays.

Text that contains only a numeric value is converted to a string using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note: The words `default`, `factory`, and `remove` are reserved words that will not appear in text when quoted as a normal string. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`

Modifier	Description	Example of String
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ε	`\Delta`	Δ	`\spadesuit`	♠

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\zeta	ζ	\Theta	Θ	\leftrighta	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	°
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

Font Style

`Color` — Text color
`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. The default color is black with an RGB triplet value of `[0 0 0]`. If you set the color to `'none'`, then the text is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`FontName` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`FontSize` — Font size

`10` (default) | scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. One point equals `1/72` inch. To change the font units, use the `FontUnits` property.

Example: `12`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontUnits` — Font size units

`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

`Units`	Description
`'points'`	1 point equals `1/72` inch.
`'inches'`	Inches
`'centimeters'`	Centimeters
`'normalized'`	Font size is interpreted as a fraction of the axes height. If you resize the axes, MATLAB modifies the font size accordingly.

Units	Description
'pixels'	Pixel size depends on the screen resolution.

This property affects the FontSize property. If you set both the FontSize and the FontUnits property in the same function call, then you must set the FontUnits property first for MATLAB to interpret the specified FontSize correctly.

FontAngle — Character slant

'normal' (default) | 'italic'

Character slant, specified as 'normal' or 'italic'. Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Note: The 'oblique' value has been removed. Use 'italic' instead.

FontWeight — Thickness of text characters

'normal' (default) | 'bold'

Thickness of the text characters, specified as one of these values:

- 'normal' — Default weight as defined by the particular font
- 'bold' — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The 'light' and 'demi' font weight values have been removed. Use 'normal' instead.

FontSmoothing — Smooth font character appearance

'on' (default) | 'off'

Smooth font character appearance, specified as one of these values:

- 'on' — Apply font smoothing. Reduce the appearance of jaggedness in the text characters to make the text easier to read.
- 'off' — Do not apply font smoothing.

Text Box

`EdgeColor` — Color of box outline

`'none'` (default) | RGB triplet | color string

Color of box outline, specified as `'none'`, a three-element RGB triplet, or a color string. The default edge color of `'none'` makes the box outline invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`BackgroundColor` — Color of text box background

`'none'` (default) | RGB triplet | color string

Color of text box background, specified as `'none'`, a three-element RGB triplet, or a color string. The default background color of `'none'` makes the text box background transparent.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: 'blue'

Example: [0 0 1]

Margin — Space around text within the text box

3 (default) | scalar numeric value

The space around the text within the text box, specified as scalar numeric value in point units.

MATLAB uses the Extent property value plus the Margin property value to determine the size of the text box.

Example: 8

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

LineStyle — Line style of box outline

'-' (default) | '--' | ':' | '-.' | 'none'

Line style of box outline, specified as one of the strings in this table.

String	Line Style	Resulting Line
'-'	Solid line	————
'--'	Dashed line	- — — — —
':'	Dotted line	.....

String	Line Style	Resulting Line
`'-.'`	Dash-dotted line	
`'none'`	Line is invisible	

`LineWidth` — Width of box outline

`0.5` (default) | scalar numeric value

Width of box outline, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Location and Size

`Position` — Location of text

`[0 0 0]` (default) | two-element vector of form `[x y]` | three-element vector of form `[x y z]`

Location of the text, specified as a two-element vector of the form `[x y]` or a three-element vector of the form `[x y z]`. If you omit the third element, `z`, then MATLAB sets it to `0`.

By default, the position value is defined in data units. To change the units, use the Units property.

Example: `[0.5 0.5 0]`

`Extent` — Size and location of rectangle that encloses text string

four-element vector

Size and location of the rectangle that encloses the text string, not including the margin, returned as a four-element vector of the form `[left bottom width height]`. The first two elements, `left` and `bottom`, define the position of the lower left corner of the rectangle. The last two elements, `width` and `height`, define the dimensions of the rectangle.

By default, the extent value is defined in data units. To change the units, use the Units property.

Example: `[0.5 0.5 0.4 0.2]`

`Units` — Position and extent units

`'data'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

`Units`	Description
`'data'` (default)	Data coordinates.
`'normalized'`	Normalized with respect to the axes. The lower left corner of the axes maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the size of characters in the default system font. The width of one character unit is the width of the letter `x`. The height of one character unit is the distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Pixel size depends on the screen resolution.

All units, except for `'data'`, are measured from the lower left corner of the axes. This property affects the Position and Extent properties.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the text object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

`Rotation` — Text orientation

0 (default) | scalar value in degrees

Text orientation, specified as a scalar value in degrees. The default rotation of 0 degrees makes the text horizontal. For vertical text, set this property to `90` or `-90`. Positive values rotate the text counterclockwise. Negative values rotate the text clockwise.

Example: `90`

Example: `-90`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`HorizontalAlignment` — Horizontal alignment of text with respect to position point

`'left'` (default) | `'center'` | `'right'`

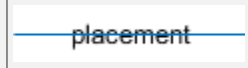
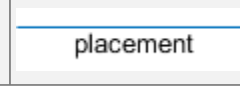
Horizontal alignment of the text with respect to the `x` value in the `Position` property, specified as one of the values in this table. The vertical line indicates where the `x` value lies in relation to the text.

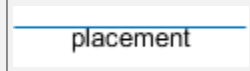
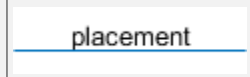
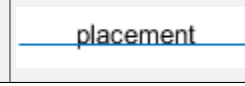
Value	Result
`'left'` (default)	
`'center'`	
`'right'`	

`VerticalAlignment` — Vertical alignment of text with respect to position point

`'middle'` (default) | `'top'` | `'bottom'` | `'baseline'` | `'cap'`

Vertical alignment of the text with respect to the `y` value in the `Position` property, specified as one of the values in this table. The horizontal line indicates where the `y` value lies in relation to the text.

Value	Result
`'middle'`	
`'top'`	

Value	Result
`'cap'`	
`'bottom'`	
`'baseline'`	

Visibility

`Visible` — Visibility of text

`'on'` (default) | `'off'`

Visibility of text, specified as one of these values:

- `'on'` — Display the text.
- `'off'` — Hide the text without deleting it. You still can access the properties of an invisible text object.

`Clipping` — Clipping of text to axes plot box

`'off'` (default) | `'on'`

Clipping of the text to the axes plot box, which is the box defined by the axis limits, specified as one of these values:

- `'off'` — Do not clip the text. Portions of it might appear outside the axes plot box.
- `'on'` — Clips the text to the axes plot box.
  - If the axes `ClippingStyle` property is set to `'3dbox'`, which is the default, then MATLAB either displays the entire text or none of the text, depending on the text position. If the point defined by the text `Position` property lies inside the axes, then MATLAB displays the entire text. If the point lies outside the axes, then MATLAB displays none of it.
  - If the axes `ClippingStyle` property is set to `'rectangle'`, then MATLAB displays portions of the text lying inside the axes plot box and does not display portions of the text lying outside the axes plot box.

Note: If the `Clipping` property of the associated axes is set to `'on'`, which is the default, then each individual object controls its own clipping behavior. If the `Clipping` property of the axes is set to `'off'`, then MATLAB does not clip any objects in the axes, regardless of the `Clipping` property of the individual object.

`EraseMode` — (removed) Technique to draw and erase objects

`'normal'` (default) | `'none'` | `'xor'` | `'background'`

Note: `EraseMode` has been removed. You can delete code that accesses the `EraseMode` property with minimal impact. If you were using `EraseMode` to create line animations, use the `animatedline` function instead.

Technique to draw and erase objects, specified as one of these values:

- `'normal'` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and, therefore, are less accurate.
- `'none'` — Do not erase the object when it is moved or destroyed. After you erase the object with `EraseMode,'none'`, it is still visible on the screen. However, you cannot print the object because MATLAB does not store any information on its former location.
- `'xor'` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object color depends on the color of whatever is beneath it on the display.
- `'background'` — Erase the object by redrawing it in the axes background color, or the figure background color if the axes `Color` property is `'none'`. This damages objects that are behind the erased object, but properly colors the erased object.

MATLAB always prints figures as if the `EraseMode` property of all objects is set to `'normal'`. This means graphics objects created with `EraseMode` set to `'none'`, `'xor'`, or `'background'` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors and ignores three-dimensional sorting to obtain greater rendering speed. However, MATLAB does not apply these techniques to the printed output. Use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

Identifiers

`Type` — Type of graphics object
`'text'`

Type of graphics object, returned as `'text'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Tag to associate with text
`''` (default) | string

Tag to associate with the text, specified as a string. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

`UserData` — Data to associate with text
`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the text object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

Parent/Child

`Parent` — Parent of text
axes object | group object | transform object

Parent of text, specified as an axes, group, or transform object.

`Children` — Children of text
empty `GraphicsPlaceholder` array

The text has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of text object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The text object handle is always visible.
- `'off'` — The text object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The text object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the text at the command-line, but allows callback functions to access it.

If the text object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to `'on'` to list all object handles regardless of their HandleVisibility property setting.

Interactive Control

`Editing` — Interactive edit mode

`'off'` (default) | `'on'`

Interactive edit mode, specified as one of these values:

- `'off'` — Do no allow interactive string editing. To change the text string you must set the String property. This is the default value.
- `'on'` — Allow interactive string editing. MATLAB places an insert cursor within the text and typing changes the text. To apply the new text string, do any of the following:
  - Press the **Esc** key.
  - Click anywhere away from the text string.

- Reset the `Editing` property to `'off'`.

MATLAB updates the String property to contain the new text and resets the `Editing` property to `'off'`.

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | string

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the text. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The text object — You can access properties of the text object from within the callback function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the text. Create the context menu using the `uicontextmenu` function.

Note: If the PickableParts property is set to `'none'` or if the HitTest property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the text when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the SelectionHighlight property also is set to `'on'`, then MATLAB displays selection handles around the text.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles when selected

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the Selected property is set to `'on'`.
- `'off'` — Never display selection handles, even when the Selected property is set to `'on'`.

Callback Execution Control

`PickableParts` — Ability to capture mouse clicks

`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The Visible property must be set to `'on'`. The HitTest property determines if the text responds to the click or if an ancestor does.
- `'all'` — Can capture mouse clicks regardless of visibility. The Visible property can be set to `'on'` or `'off'`. The HitTest property determines if the text responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the text passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The HitTest property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the ButtonDownFcn callback of the text. If you have defined the UIContextMenu property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the text that has a HitTest property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note: The PickableParts property determines if the text object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.

- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the text is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see "Interrupt Callback Execution".
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note: There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the text tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

Creation and Deletion Control

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | string

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the text. Setting the `CreateFcn` property on an existing text has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during text creation. MATLAB executes the callback after creating the text and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The text object — You can access properties of the text object from within the callback function. You also can access the text object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | string

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- String that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the text. MATLAB executes the callback before destroying the text so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The text object — You can access properties of the text object from within the callback function. You also can access the text object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`BeingDeleted` — Deletion status of text

`'off'` (default) | `'on'`

Deletion status of text, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the text begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the text no longer exists.

Check the value of the BeingDeleted property to verify that the text is not about to be deleted before querying or modifying it.

See Also

`text`

More About

- "Access Property Values"
- "Graphics Object Properties"

textread

Read data from text file; write to multiple outputs

Note: `textread` is not recommended. Use `textscan` instead.

Syntax

```
[A,B,C,...] = textread(filename,format)
[A,B,C,...] = textread(filename,format,N)
[...] = textread(...,param,value,...)
```

Description

`[A,B,C,...] = textread(filename,format)` reads data from the file `filename` into the variables `A`, `B`, `C`, and so on, using the specified `format`, until the entire file is read. The `filename` and `format` inputs are strings, each enclosed in single quotes. `textread` is useful for reading text files with a known format. `textread` handles both fixed and free format files.

Note When reading large text files, reading from a specific point in a file, or reading file data into a cell array rather than multiple outputs, you might prefer to use the `textscan` function.

`textread` matches and converts groups of characters from the input. Each input field is defined as a string of non-white-space characters that extends to the next white-space or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated white-space characters are treated as one.

The `format` string determines the number and types of return arguments. The number of return arguments is the number of items in the `format` string. The `format` string supports a subset of the conversion specifiers and conventions of the C language

`fscanf` routine. Values for the `format` string are listed in the table below. White-space characters in the `format` string are ignored.

format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a file that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space or delimiter-separated string.	Cell array of strings
`%q`	Read a double quoted string, ignoring the quotes.	Cell array of strings
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings
`%[^...]`	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
`%*...` `instead of %`	Ignore the matching characters specified by `*`.	No output
`%w...` `instead of %`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

`[A,B,C,...] = textread(filename,format,N)` reads the data, reusing the `format` string `N` times, where `N` is an integer greater than zero. If `N` is smaller than zero, `textread` reads the entire file.

`[...] = textread(...,param,value,...)` customizes `textread` using `param`/`value` pairs, as listed in the table below.

param	value	Action
`bufsize`	Positive integer	Specifies the maximum string length, in bytes. Default is `4095`.

param	value		Action
`commentstyle`	`matlab`		Ignores characters after `%`.
`commentstyle`	`shell`		Ignores characters after `#`.
`commentstyle`	`c`		Ignores characters between `/*` and `*/`.
`commentstyle`	`c++`		Ignores characters after `//`.
`delimiter`	One or more characters		Act as delimiters between elements. Default is none.
`emptyvalue`	Scalar double		Value given to empty cells when reading delimited files. Default is 0.
`endofline`	Single character or `'\r\n'`		Character that denotes the end of a line. Default is determined from file
`expchars`	Exponent characters		Default is `eEdD`.
`headerlines`	Positive integer		Ignores the specified number of lines at the beginning of the file.
`whitespace`	Any from the list below:		Treats vector of characters as white space. Default is `' \b\t'`.
	`' '`	Space	
	`\b`	Backspace	
	`\n`	Newline	
	`\r`	Carriage return	
	`\t`	Horizontal tab	

Note When `textread` reads a consecutive series of `whitespace` values, it treats them as one white space. When it reads a consecutive series of `delimiter` values, it treats each as a separate delimiter.

Examples

Example 1 — Read All Fields in Free Format File Using %

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a free format file using the `%` format.

```
[names, types, x, y, answer] = textread('mydata.dat', ...
'%s %s %f %d %s', 1)
```

returns

```
names =
    'Sally'
types =
    'Level1'
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

Example 2 — Read as Fixed Format File, Ignoring the Floating Point Value

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a fixed format file, ignoring the floating-point value.

```
[names, types, y, answer] = textread('mydata.dat', ...
'%9c %6s %*f %2d %3s', 1)
```

returns

```
names =
Sally
types =
    'Level1'
y =
    45
answer =
    'Yes'
```

`%*f` in the `format` string causes `textread` to ignore the floating point value, in this case, `12.34`.

Example 3 — Read Using Literal to Ignore Matching Characters

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file, ignoring the characters `Type` in the second field.

```
[names, typenum, x, y, answer] = textread('mydata.dat', ...
'%s Type%d %f %d %s', 1)
```

returns

```
names =
    'Sally'
typenum =
    1
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

`Type%d` in the `format` string causes the characters `Type` in the second field to be ignored, while the rest of the second field is read as a signed integer, in this case, `1`.

Example 4 — Specify Value to Fill Empty Cells

For files with empty cells, use the `emptyvalue` parameter. Suppose the file `data.csv` contains:

```
1,2,3,4,,6
7,8,9,,11,12
```

Read the file using `NaN` to fill any empty cells:

```
data = textread('data.csv', '', 'delimiter', ',', ...
                'emptyvalue', NaN);
```

Example 5 — Read File into a Cell Array of Strings

Read the file `fft.m` into cell array of strings.

```
file = textread('fft.m', '%s', 'delimiter', '\n', ...
                'whitespace', '');
```

More About

Tips

If you want to preserve leading and trailing spaces in a string, use the `whitespace` parameter as shown here:

```
textread('myfile.txt', '%s', 'whitespace', '')
ans =
    '   An  example      of preserving    spaces      '
```

See Also

`textscan` | `fscanf` | `dlmread`

Introduced before R2006a

textscan

Read formatted data from text file or string

Syntax

```
C = textscan(fileID,formatSpec)
C = textscan(fileID,formatSpec,N)

C = textscan(str,formatSpec)
C = textscan(str,formatSpec,N)

C = textscan( ___ ,Name,Value)

[C,position] = textscan( ___ )
```

Description

`C = textscan(fileID,formatSpec)` reads data from an open text file into a cell array, `C`. The text file is indicated by the file identifier, `fileID`. Use `fopen` to open the file and obtain the `fileID` value. When you finish reading from a file, close the file by calling `fclose(fileID)`.

`textscan` attempts to match the data in the file to `formatSpec`, which is a string of conversion specifiers. `textscan` reapplies `formatSpec` throughout the entire file and stops when it cannot match `formatSpec` to the data.

`C = textscan(fileID,formatSpec,N)` reads file data using the `formatSpec` `N` times, where `N` is a positive integer. To read additional data from the file after `N` cycles, call `textscan` again using the original `fileID`. If you resume a text scan of a file by calling `textscan` with the same file identifier (`fileID`), then `textscan` automatically resumes reading at the point where it terminated the last read.

`C = textscan(str,formatSpec)` reads data from a string, `str`, into cell array `C`. For strings, repeated calls to `textscan` restart the scan from the beginning each time. To restart a scan from the last position, request a `position` output.

`textscan` attempts to match the data in the string, `str`, to `formatSpec`, which is a string of conversion specifiers.

`C = textscan(str,formatSpec,N)` reads string data using the `formatSpec` `N` times, where `N` is a positive integer.

`C = textscan( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments, in addition to any of the input arguments in the previous syntaxes.

`[C,position] = textscan( ___ )` returns the file or string position at the end of the scan as the second output argument, using any of the input arguments in the previous syntaxes. For a file, this is the value that `ftell(fileID)` would return after calling `textscan`. For a string, `position` indicates how many characters `textscan` read.

Examples

Read a String

Read a string of floating-point numbers.

```
str = '0.41 8.24 3.57 6.24 9.27';
C = textscan(str,'%f');
```

The `formatSpec` string `'%f'` tells `textscan` to match each field in `str` to a double-precision floating-point number.

Display the contents of cell array `C`.

```
celldisp(C)
```

```
C{1} =

    0.4100
    8.2400
    3.5700
    6.2400
    9.2700
```

Read the same string, truncating each value to one decimal digit.

```
C = textscan(str,'%3.1f %*1d');
```

The specifier `%3.1f` indicates a field width of 3 digits and a precision of 1. `textscan` reads a total of 3 digits, including the decimal point and the 1 digit after the decimal point. The specifier, `%*1d`, tells `textscan` to skip the remaining digit.

Display the contents of cell array `C`.

```
celldisp(C)
```

```
C{1} =

    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
```

Read Different Types of Data

Using a text editor, create a file `scan1.dat` that contains data in the following form:

```
09/12/2005 Level1 12.34 45 1.23e10 inf Nan Yes 5.1+3i
10/12/2005 Level2 23.54 60 9e19 -inf  0.001 No 2.2-.5i
11/12/2005 Level3 34.90 12 2e5   10  100   No 3.1+.1i
```

Open the file, and read each column with the appropriate conversion specifier.

```
fileID = fopen('scan1.dat');
C = textscan(fileID,'%s %s %f32 %d8 %u %f %f %s %f');
fclose(fileID);
celldisp(C)
```

```
C{1}{1} =

09/12/2005


C{1}{2} =

10/12/2005


C{1}{3} =
```

```
11/12/2005


C{2}{1} =

Level1


C{2}{2} =

Level2


C{2}{3} =

Level3


C{3} =

   12.3400
   23.5400
   34.9000



C{4} =

   45
   60
   12



C{5} =

  4294967295
  4294967295
      200000



C{6} =
```

```
   Inf
  -Inf
    10


C{7} =

       NaN
    0.0010
  100.0000


C{8}{1} =

Yes


C{8}{2} =

No


C{8}{3} =

No


C{9} =

   5.1000 + 3.0000i
   2.2000 - 0.5000i
   3.1000 + 0.1000i
```

`textscan` returns a 1-by-9 cell array `C`.

View the MATLAB data type of each of the cells in `C`.

```
C
```

```
C =

  Columns 1 through 5
```

```
    {3x1 cell}    {3x1 cell}    [3x1 single]    [3x1 int8]    [3x1 uint32]

  Columns 6 through 9

    [3x1 double]    [3x1 double]    {3x1 cell}    [3x1 double]
```

For example, `C{1}` and `C{2}` are cell arrays. `C{5}` is of data type `uint32`, so the first two elements of `C{5}` are the maximum values for a 32-bit unsigned integer, or `intmax('uint32')`.

Remove a Literal String

Remove the text `'Level'` from each field in the second column of the data from the previous example.

Match the literal string in the `formatSpec` input.

```
fileID = fopen('scan1.dat');
C = textscan(fileID,'%s Level%d %f32 %d8 %u %f %f %s %f');
fclose(fileID);
C{2}
```

```
ans =

           1
           2
           3
```

View the MATLAB data type of the second cell in `C`.

```
class(C{2})
```

```
ans =

int32
```

The second cell of the 1-by-9 cell array, `C`, is now of data type `int32`.

Skip the Remainder of a Line

Read the first column of the file in the previous example into a cell array, skipping the rest of the line.

```
fileID = fopen('scan1.dat');
dates = textscan(fileID,'%s %*[^\n]');
```

```
fclose(fileID);
dates{1}

ans =

    '09/12/2005'
    '10/12/2005'
    '11/12/2005'
```

`textscan` returns a 1-by-1 cell array `dates`.

Specify Delimiter and Empty Value Conversion

Using a text editor, create a comma-delimited file, `data.csv`, that contains

```
1,  2,  3,  4,   ,  6
7,  8,  9,   , 11, 12
```

Read the file, converting empty cells to `-Inf`.

```
fileID = fopen('data.csv');
C = textscan(fileID,'%f %f %f %f %u8 %f',...
'Delimiter',',','EmptyValue',-Inf);
fclose(fileID);
column4 = C{4}, column5 = C{5}

column4 =

     4
  -Inf


column5 =

    0
   11
```

`textscan` returns a 1-by-6 cell array, `C`. The `textscan` function converts the empty value in `C{4}` to `-Inf`, where `C{4}` is associated with a floating-point format. Because MATLAB represents unsigned integer `-Inf` as `0`, `textscan` converts the empty value in `C{5}` to `0`, and not `-Inf`.

Read Custom Empty Value Strings and Comments

Using a text editor, create a comma-delimited file, `data2.csv`, that contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```

Designate the input that `textscan` should treat as comments or empty values.

```
fileID = fopen('data2.csv');
C = textscan(fileID,'%s %n %n %n %n','Delimiter',',',...
'TreatAsEmpty',{'NA','na'},'CommentStyle','//');
fclose(fileID);
celldisp(C)

C{1}{1} =
abc

C{1}{2} =
def

C{2} =
     2
   NaN

C{3} =
   NaN
     5

C{4} =
     3
     6

C{5} =
     4
     7
```

Treat Repeated Delimiters as One

Using a text editor, create a file, `data3.csv`, that contains

```
1,2,3,,4
5,6,7,,8
```

To treat the repeated commas as a single delimiter, use the `MultipleDelimsAsOne` parameter, and set the value to `1 (true)`.

```
fileID = fopen('data3.csv');
```

```
C = textscan(fileID,'%f %f %f %f','Delimiter',',',...
'MultipleDelimsAsOne',1);
fclose(fileID);
celldisp(C)

C{1} =
     1
     5

C{2} =
     2
     6

C{3} =
     3
     7

C{4} =
     4
     8
```

Specify Repeated Conversion Specifiers and Collect Numeric Data

Using a text editor, create a file, `grades.txt`, that contains:

```
Student_ID  | Test1  | Test2  | Test3
   1          91.5     89.2     77.3
   2          88.0     67.8     91.0
   3          76.3     78.1     92.5
   4          96.4     81.2     84.6
```

Read the column headers using the format `'%s'` four times.

```
fileID = fopen('grades.txt');

formatSpec = '%s';
N = 4;
C_text = textscan(fileID,formatSpec,N,'Delimiter','|');
```

Read the numeric data in the file.

```
C_data0 = textscan(fileID,'%d %f %f %f')

C_data0 =
  [4x1 int32]    [4x1 double]    [4x1 double]    [4x1 double]
```

The default value for `CollectOutput` is 0 `(false)`, so `textscan` returns each column of the numeric data in a separate array.

Set the file position indicator to the beginning of the file.

```
frewind(fileID);
```

Reread the file and set `CollectOutput` to `1 (true)` to collect the consecutive columns of the same class into a single array. You can use the `repmat` function to indicate that the `%f` conversion specifier should appear three times. This technique is useful when a format repeats many times.

```
C_text = textscan(fileID,'%s',N,'Delimiter','|');
```

```
C_data1 = textscan(fileID,['%d',repmat('%f',[1,3])],'CollectOutput',1)
```

```
C_data1 =
    [4x1 int32]    [4x3 double]
```

The test scores, which are all `double`, are collected into a single 4-by-3 array.

Close the file, `grades.txt`.

```
fclose(fileID);
```

Read or Skip Quoted Strings and Numeric Fields

Read the first and last columns of data from a text file. Skip a column of strings and a column of integer data.

In a text editor, create a comma-delimited text file called `names.txt` that contains:

```
"Smith, J.","M",38,71.1
"Bates, G.","F",43,69.3
"Curie, M.","F",38,64.1
"Murray, G.","F",40,133.0
"Brown, K.","M",49,64.9
```

Read the first and last columns of data in the file. Use the conversion specifier, `%q` to read a string enclosed by double quotation marks ("). `%*q` skips a quoted string, `%*d` skips an integer field, and `%f` reads a floating-point number. Specify the comma delimiter using the `'Delimiter'` name-value pair argument.

```
fileID = fopen('names.txt','r');
C = textscan(fileID,'%q %*q %*d %f','Delimiter',',');
```

```
fclose(fileID);
celldisp(C)

C{1}{1} =

Smith, J.


C{1}{2} =

Bates, G.


C{1}{3} =

Curie, M.


C{1}{4} =

Murray, G.


C{1}{5} =

Brown, K.


C{2} =

   71.1000
   69.3000
   64.1000
  133.0000
   64.9000
```

`textscan` returns a 1-by-2 cell array, `C`. Double quotation marks enclosing the strings are removed.

Read Foreign-Language Dates

Create a sample file named `myfile.txt` that contains comma-separated values. The first column of values contains dates in German and the second and third columns are numeric values.

```
fileID = fopen('myfile.txt','w','n','ISO-8859-15');
fprintf(fileID,'1 Januar 2014, 20.2, 100.5 \n');
fprintf(fileID,'1 Februar 2014, 21.6, 102.7 \n');
fprintf(fileID,'1 März 2014, 20.7, 99.8 \n');
fclose(fileID);
```

The sample file looks like this:

```
1 Januar 2014, 20.2, 100.5
1 Februar 2014, 21.6, 102.7
1 März 2014, 20.7, 99.8
```

Open the file. Specify the character encoding scheme associated with the file as the last input to `fopen`.

```
fileID = fopen('myfile.txt','r','n','ISO-8859-15');
```

Read the file. Specify the format of the dates in the file using the `%{dd % MMMM yyyy}D` specifier. Specify the locale of the dates using the `DateLocale` name-value pair argument.

```
C = textscan(fileID,'%{dd MMMM yyyy}D %f %f',...
    'DateLocale','de_DE','Delimiter',',');
fclose(fileID);
```

View the contents of the first cell in `C`.

```
C{1}
```

```
ans =

   01 January 2014
   01 February 2014
   01 March 2014
```

The dates display in the language MATLAB uses depending on your system locale.

Read Nondefault Control Characters

Use `sprintf` to convert nondefault escape sequences in your data.

Create a string that includes a form feed character, `\f`. Then, to read the string using `textscan`, call `sprintf` to explicitly convert the form feed.

```
lyric = sprintf('Blackbird\fsinging\fin\fthe\fdead\fof\fnight');
C = textscan(lyric,'%s','delimiter',sprintf('\f'));
C{1}


ans =

    'Blackbird'
    'singing'
    'in'
    'the'
    'dead'
    'of'
    'night'
```

`textscan` returns a 1-by-1 cell array, `C`.

Resume a Text Scan of a String

Resume a scan of a string from a position other than the beginning.

If you resume a text scan of a string, `textscan` reads from the beginning of the string each time. To resume a scan from any other position in the string, use the two-output argument syntax in your initial call to `textscan`.

For example, create a string called `lyric`. Read the first word of the string, and then resume the scan.

```
lyric = 'Blackbird singing in the dead of night';
[firstword,pos] = textscan(lyric,'%9c',1);
lastpart = textscan(lyric(pos+1:end),'%s');
```

- “Import Data from a Nonrectangular Text File”
- “Import Large Text File Data in Blocks”
- “Access Data in a Cell Array”

Input Arguments

`fileID` — File identifier

numeric scalar

File identifier of an open text file, specified as a number. Before reading a file with `textscan`, you must use `fopen` to open the file and obtain the `fileID`.

Data Types: `double`

`formatSpec` — Format of the data fields

string

Format of the data fields, specified as a string of one or more conversion specifiers. When `textscan` reads a file or string, it attempts to match the data to the `formatSpec` string. If `textscan` fails to match a data field, it stops reading and returns all fields read before the failure.

The number of conversion specifiers determines the number of cells in output array, `C`.

Numeric Fields

This table lists available conversion specifiers for numeric inputs.

Numeric Input Type	Conversion Specifier	Output Class
Integer, signed	`%d`	`int32`
	`%d8`	`int8`
	`%d16`	`int16`
	`%d32`	`int32`
	`%d64`	`int64`
Integer, unsigned	`%u`	`uint32`
	`%u8`	`uint8`
	`%u16`	`uint16`
	`%u32`	`uint32`
	`%u64`	`uint64`
Floating-point number	`%f`	`double`
	`%f32`	`single`
	`%f64`	`double`
	`%n`	`double`

Nonnumeric Fields

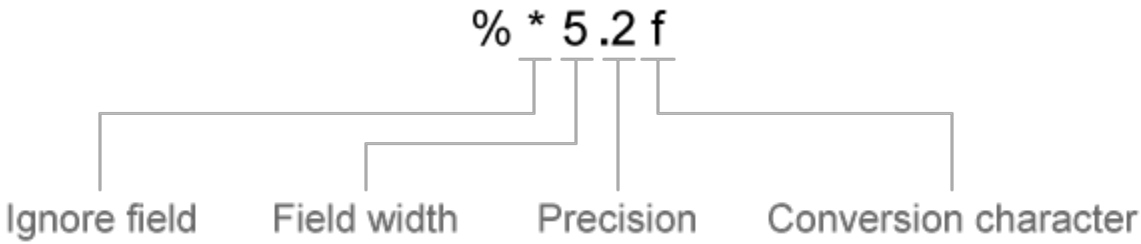
This table lists available conversion specifiers for inputs that include nonnumeric characters.

Strings	Conversion Specifier	Details
Character	`%c`	Read any single character, including a delimiter.
Text string	`%s`	Read a string.
	`%q`	Read a string. If the string begins with a double quotation mark ("), omit the leading quotation mark and its accompanying closing mark, which is the second instance of a lone double quotation mark. Replace escaped double quotation marks (for example, `""abc""`) with lone double quotation marks (`"abc"`). `%q` ignores any double quotation marks that appear after the closing double quotation mark. Example: `'%q'` reads `'"Joe ""Lightning"" Smith, Jr."'` as `'Joe "Lightning" Smith, Jr.'`.
Dates and time	`%D`	Read a string in the same way as `%q` above, and then convert to a datetime value.
	`%{`*`fmt`*`}D`	Read a string in the same way as `%q` above, and then convert to a datetime value. *`fmt`* describes the format of the input string. The *`fmt`* input is a string of letter identifiers that is a valid value for the `Format` property of a datetime. `textscan` converts strings that do not match this format to `NaT` values. For more information about datetime display formats, see the `Format` property for datetime arrays. Example: `'%{dd-MMM-yyyy}D'` specifies the format of a date such as `'01-Jan-2014'` .

Strings	Conversion Specifier	Details
Category string	`%C`	Read a string in the same way as `%q`, and then convert to a category name in a categorical array. `textscan` converts the string, `<undefined>` to an undefined value in the output categorical array.
Pattern-matching strings	`%[...]`	Read only the characters inside the brackets up to the first nonmatching character. To include `]` in the set, specify it first: `%[]...]`. Example: `%[mus]` reads `'summer '` as `'summ'`.
	`%[^...]`	Exclude characters inside the brackets, reading until the first matching character. To exclude `]`, specify it first: `%[^]...]`. Example: `%[^xrg]` reads `'summer '` as `'summe'`.

Optional Operators

Conversion specifiers in `formatSpec` can include optional operators, which appear in the following order (includes spaces for clarity):



Optional operators include:

- Fields and Characters to Ignore

  `textscan` reads all characters in your file in sequence, unless you tell it to ignore a particular field or a portion of a field.

Insert an asterisk character (*) after the percent character (%) to skip a field or a portion of a character field.

Operator	Action Taken
`%*k`	Skip the field. *k* is any conversion specifier identifying the field to skip. `textscan` does not create an output cell for any such fields. Example: `'%s %*s %s %s %*s %*s %s'` (spaces are optional) converts the string `'Blackbird singing in the dead of night'` to four output cells with the strings `'Blackbird' 'in' 'the' 'night'`
`'%*ns'`	Skip up to *n* characters, where *n* is an integer less than or equal to the number of characters in the field. Example: `'%*3s %s'` converts `'abcdefg'` to `'defg'`. When the delimiter is a comma, the same delimiter converts `'abcde,fghijkl'` to a cell array containing `'de';'ijkl'`.
`'%*nc'`	Skip *n* characters, including delimiter characters.

- Field Width

  `textscan` reads the number of characters or digits specified by the field width or precision, or up to the first delimiter, whichever comes first. A decimal point, sign (+ or -), exponent character, and digits in the numeric exponent are counted as characters and digits within the field width. For complex numbers, the field width refers to the individual widths of the real part and the imaginary part. For the imaginary part, the field width includes + or − but not `i` or `j`. Specify the field width by inserting a number after the percent character (%) in the conversion specifier.

  Example: `%5f` reads `'123.456'` as `123.4`.

  Example: `%5c` reads `'abcdefg'` as `'abcde'`.

  When the field width operator is used with single characters (`%c`), `textscan` also reads delimiter, white-space, and end-of-line characters.
  Example: `%7c` reads 7 characters, including white-space, so `'Day and night'` reads as `'Day and'`.

- Precision

For floating-point numbers (`%n`, `%f`, `%f32`, `%f64`), you can specify the number of decimal digits to read.

Example: `%7.2f` reads `'123.456'` as `123.45`.

- Literal Text to Ignore

  `textscan` ignores specified text appended to the `formatSpec` string.

  Example: `Level%u8` reads `'Level1'` as `1`.

  Example: `%u8Step` reads `'2Step'` as `2`.

`N` — Number of times to apply `formatSpec`

`Inf` (default) | positive integer

Number of times to apply `formatSpec`, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`str` — Input string

string

Input string to read.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Names are not case sensitive.

Example: `C = textscan(fileID,formatSpec,'HeaderLines',3,'Delimiter',',')` skips the first three lines of the data, and then reads the remaining data, treating commas as a delimiter.

`'CollectOutput'` — Logical indicator determining data concatenation

`false` (default) | `true`

Logical indicator determining data concatenation, specified as the comma-separated pair consisting of `'CollectOutput'` and either `true` or `false`. If `true`, then `textscan` concatenates consecutive output cells of the same fundamental MATLAB class into a single array.

`'CommentStyle'` — Symbols designating text to ignore

string | cell array of strings

Symbols designating text to ignore, specified as the comma-separated pair consisting of `'CommentStyle'` and a string or cell array of strings.

For example, specify a string such as `'%'` to ignore characters following the string on the same line. Specify a cell array of two strings, such as `{'/*', '*/'}`, to ignore characters between the strings.

`textscan` checks for comments only at the start of each field, not within a field.

Example: `'CommentStyle',{'/*', '*/'}`

`'DateLocale'` — Locale for reading dates

string

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a string in the form *`xx_YY`*, where *`xx`* is a lowercase ISO 639-1 two-letter code that specifies a language, and *`YY`* is an uppercase ISO 3166-1 alpha-2 code that specifies a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

Use `DateLocale` to specify the locale in which textscan should interpret month and day of week names and abbreviations when reading text as dates using the `%D` format specifier.

Example: `'DateLocale','ja_JP'`

`'Delimiter'` — Field delimiter characters

string | cell array of strings

Field delimiter characters, specified as the comma-separated pair consisting of `'Delimiter'` and a string or a cell array of strings. Specify multiple delimiters in a cell array of strings.

Example: `'Delimiter',{';','*'}`

`textscan` interprets repeated delimiter characters as separate delimiters, and returns an empty value to the output cell.

Within each row of data, the default field delimiter is white-space. White-space can be any combination of space (`' '`), backspace (`'\b'`), or tab (`'\t'`) characters. If you do not specify a delimiter, then:

- the delimiter characters are the same as the white-space characters. The default white-space characters are `' '`, `'\b'`, and `'\t'`. Use the `'Whitespace'` name-value pair argument to specify alternate white-space characters.
- `textscan` interprets repeated white-space characters as a single delimiter.

When you specify one of the following escape sequences as a delimiter, `textscan` converts that sequence to the corresponding control character:

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (\)

`'EmptyValue'` — Returned value for empty numeric fields

`NaN` (default) | scalar

Returned value for empty numeric fields in delimited text files, specified as the comma-separated pair consisting of `'EmptyValue'` and a scalar.

`'EndOfLine'` — End-of-line characters

string

End-of-line characters, specified as the comma-separated pair consisting of `'EndOfLine'` and a string. The string must specify a single character or be `'\r\n'`. The default end-of-line sequence depends on the format of your file and can include a newline character (`'\n'`), a carriage return (`'\r'`), or a combination of the two (`'\r\n'`).

If there are missing values and an end-of-line sequence at the end of the last line in a file, then `textscan` returns empty values for those fields. This ensures that individual cells in output cell array, `C`, are the same size.

Example: `'EndOfLine',':'`

'ExpChars' — Exponent characters
'eEdD' (default) | string

Exponent characters, specified as the comma-separated pair consisting of 'ExpChars' and a string. The default exponent characters are e, E, d, and D.

'HeaderLines' — Number of header lines
0 (default) | positive integer

Number of header lines, specified as the comma-separated pair consisting of 'HeaderLines' and a positive integer. textscan skips the header lines, including the remainder of the current line.

'MultipleDelimsAsOne' — Multiple delimiter handling
0 (false) (default) | 1 (true)

Multiple delimiter handling, specified as the comma-separated pair consisting of 'MultipleDelimsAsOne' and either true or false. If true, textscan treats consecutive delimiters as a single delimiter. Repeated delimiters separated by white-space are also treated as a single delimiter. You must also specify the Delimiter option.

Example: 'MultipleDelimsAsOne',1

'ReturnOnError' — Behavior when textscan fails to read or convert
1 (true) (default) | 0 (false)

Behavior when textscan fails to read or convert, specified as the comma-separated pair consisting of 'ReturnOnError' and either true or false. If true, textscan terminates without an error and returns all fields read. If false, textscan terminates with an error and does not return an output cell array.

'TreatAsEmpty' — Strings to treat as empty value
string | cell array of strings

Strings to treat as empty values, specified as the comma-separated pair consisting of 'TreatAsEmpty' and a single string or cell array of strings. This option only applies to numeric fields.

'Whitespace' — White-space characters
' \b\t' (default) | string

White-space characters, specified as the comma-separated pair consisting of 'Whitespace' and a string of one or more characters. textscan adds a space

character, `char(32)`, to any specified `Whitespace`, unless `Whitespace` is empty (`''`) and `formatSpec` includes any string conversion specifier.

When you specify one of the following escape sequences as any white-space character, `textscan` converts that sequence to the corresponding control character:

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (\)

Output Arguments

`C` — File or string data

cell array

File or string data, returned as a cell array.

For each numeric conversion specifier in `formatSpec`, the `textscan` function returns a `K`-by-1 MATLAB numeric vector to the output cell array, `C`, where `K` is the number of times that `textscan` finds a field matching the specifier.

For each string conversion specifier in `formatSpec`, the `textscan` function returns a `K`-by-1 cell vector of strings, where `K` is the number of times that `textscan` finds a field matching the specifier. For each character conversion that includes a field width operator, `textscan` returns a `K`-by-`M` character array, where `M` is the field width.

For each datetime or categorical conversion specifier in `formatSpec`, the `textscan` function returns a `K`-by-1 datetime or categorical vector to the output cell array, `C`, where `K` is the number of times that `textscan` finds a field matching the specifier.

`position` — File or string position

integer

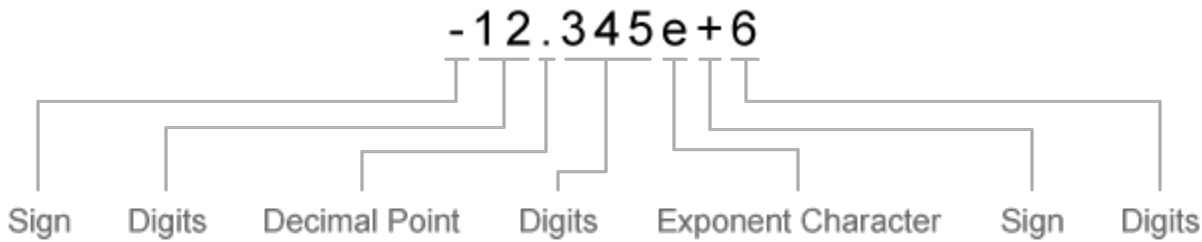
File or string position at the end of the scan, returned as an integer of class `double`. For a file, `ftell(fileID)` would return the same value after calling `textscan`. For a string, `position` indicates how many characters `textscan` read.

More About

Algorithms

`textscan` converts numeric fields to the specified output type according to MATLAB rules regarding overflow, truncation, and the use of `NaN`, `Inf`, and `-Inf`. For example, MATLAB represents an integer `NaN` as zero. If `textscan` finds an empty field associated with an integer format specifier (such as `%d` or `%u`), it returns the empty value as zero and not `NaN`.

When matching data to a string conversion specifier, `textscan` reads until it finds a delimiter or an end-of-line character. When matching data to a numeric conversion specifier, `textscan` reads until it finds a nonnumeric character. When `textscan` can no longer match the data to a particular conversion specifier, it attempts to match the data to the next conversion specifier in the `formatSpec` string. Sign (`+` or `-`), exponent characters, and decimal points are considered numeric characters.



Sign	Digits	Decimal Point	Digits	Exponent Character	Sign	Digits
Read one sign character if it exists.	Read one or more digits.	Read one decimal point if it exists.	If there is a decimal point, read one or more digits that immediately follow it.	Read one exponent character if it exists.	If there is an exponent character, read one sign character.	If there is an exponent character, read one or more digits that follow it.

`textscan` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type (such as `%d` or `%f`). Valid forms for a complex number are:

`±<real>±<imag>i|j`	Example: `5.7-3.1i`
`±<imag>i|j`	Example: `-7j`

Do not include embedded white space in a complex number. `textscan` interprets embedded white space as a field delimiter.

- "Ways to Import Text Files"

See Also

`dlmread` | `fopen` | `fread` | `fscanf` | `load` | `readtable` | `uiimport` | `xlsread`

Introduced before R2006a

textwrap

Wrapped string matrix for given uicontrol

Syntax

```
outstring = textwrap(h,instring)
outstring = textwrap(h,instring,columns)
[outstring,position] = textwrap(...)
```

Description

`outstring = textwrap(h,instring)` returns a wrapped string cell array, `outstring`, that fits inside the uicontrol with handle `h`. `instring` is a cell array, with each cell containing a single line of text. `outstring` is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.

`outstring = textwrap(h,instring,columns)` returns an `outstring` with each line wrapped at `columns` characters. Spaces are included in the character count.

`[outstring,position] = textwrap(...)` returns the recommended position of the uicontrol in the units of the uicontrol. `position` considers the extent of the multiline text in the *x* and *y* directions.

`textwrap` maintains the original line breaks in the input cell array and adds new ones. It can calculate uicontrol positions with any type of `Units`, including normalized units.

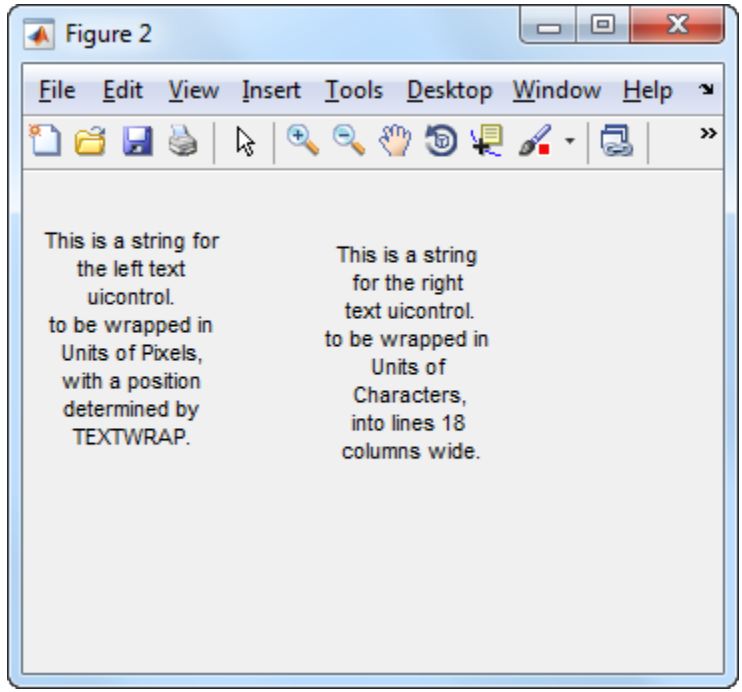
Examples

Place two text-wrapped strings in text uicontrols. The left one has a `Position` calculated by `textwrap` in `Units` of `pixels`; the right one's `Position` is calculated manually in `Units` of `characters`:

```
figure('Position',[560 528 350 250]);
% Make a text uicontrol to wrap in Units of Pixels
% Create it in Units of Pixels, 100 wide, 10 high
pos = [10 100 100 10];
ht = uicontrol('Style','Text','Position',pos);
```

```
string = {'This is a string for the left text uicontrol.',...
          'to be wrapped in Units of Pixels,',...
          'with a position determined by TEXTWRAP.'};
% Wrap string, also returning a new position for ht
[outstring,newpos] = textwrap(ht,string) %#ok<NOPRT>

outstring =
    'This is a string for'
    'the left text'
    'uicontrol.'
    'to be wrapped in'
    'Units of Pixels,'
    'with a position'
    'determined by'
    'TEXTWRAP.'

newpos =
    10   100    91   124

set(ht,'String',outstring,'Position',newpos)

% Make another text uicontrol to wrap to a column width of 18
colwidth = 18;
% Create it in Units of Pixels, 100 wide, 10 high
pos1 = [150 100 100 10];
ht1 = uicontrol('Style','Text','Position',pos1);
string1 = {'This is a string for the right text uicontrol.',...
          'to be wrapped in Units of Characters,',...
          'into lines 18 columns wide.'};
outstring1 = textwrap(ht1,string1,colwidth);
% Reset Units of ht1 to Characters to use the result
set(ht1,'Units','characters')
newpos1 = get(ht1,'Position')

newpos1 =
   29.8000    7.6154   20.0000    0.7692

% Set new Position in Characters to be specified colwidth
% with height the length of the outstring1 cell array + 1.
newpos1(3) = colwidth;
newpos1(4) = length(outstring1)+1

newpos1 =
   29.8000    7.6154   15.0000   10.0000
```

```
set(ht1,'String',outstring1,'Position',newpos1)
```



More About

Tips

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uicontrol object fires the CreateFcn callback repeatedly, which raises a series of error messages.

See Also

`align` | `uicontrol`

Introduced before R2006a

tfqmr

Transpose-free quasi-minimal residual method

Syntax

```
x = tfqmr(A,b)
x = tfqmr(afun,b)
x = tfqmr(a,b,tol)
x = tfqmr(a,b,tol,maxit)
x = tfqmr(a,b,tol,maxit,m)
x = tfqmr(a,b,tol,maxit,m1,m2,x0)
[x,flag] = tfqmr(A,B,...)
[x,flag,relres] = tfqmr(A,b,...)
[x,flag,relres,y]y(A,b,...)
[x,flag,relres,iter,resvec] = tfqmr(A,b,...)
```

Description

`x = tfqmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the right-hand side column vector `b` must have length `n`.

`x = tfqmr(afun,b)` accepts a function handle, `afun`, instead of the matrix `A`. The function, `afun(x)`, accepts a vector input `x` and returns the matrix-vector product `A*x`. In all of the following syntaxes, you can replace `A` by `afun`. "Parameterizing Functions" explains how to provide additional parameters to the function `afun`.

`x = tfqmr(a,b,tol)` specifies the tolerance of the method. If `tol` is `[]` then `tfqmr` uses the default, 1e-6.

`x = tfqmr(a,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]` then `tfqmr` uses the default, `min(N,20)`.

`x = tfqmr(a,b,tol,maxit,m)` and `x = tfqmr(a,b,tol,maxit,m1,m2)` use preconditioners `m` or `m=m1*m2` and effectively solve the system `A*inv(M)*x = B` for `x`. If

M is [] then a preconditioner is not applied. M may be a function handle mfun such that mfun(x) returns m\x.

x = tfqmr(a,b,tol,maxit,m1,m2,x0) specifies the initial guess. If x0 is [] then tfqmr uses the default, an all zero vector.

[x,flag] = tfqmr(A,B,...) also returns a convergence flag:

Flag	Convergence
0	tfqmr converged to the desired tolerance tol within maxit iterations.
1	tfqmr iterated maxit times but did not converge.
2	Preconditioner m was ill-conditioned.
3	tfqmr stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during tfqmr became too small or too large to continue computing.

[x,flag,relres] = tfqmr(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, then relres <= tol.

[x,flag,relres,y]y(A,b,...) also returns the iteration number at which x was computed: 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = tfqmr(A,b,...) also returns a vector of the residual norms at each iteration, including norm(b-A*x0).

Examples

Using tfqmr with Matrix or Function Handle Input

This example shows how to use tfqmr with a matrix input and with a function input.

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
```

```
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = tfqmr(A,b,tol,maxit,M1,M2,[]);
```

You can also use a matrix-vector product function as input:

```
function y = afun(x,n)
y = 4 * x;
y(2:n) = y(2:n) - 2 * x(1:n-1);
y(1:n-1) = y(1:n-1) - x(2:n);
x1 = tfqmr(@(x)afun(x,n),b,tol,maxit,M1,M2);
```

If `applyOp` is a function suitable for use with `qmr`, it may be used with `tfqmr` by wrapping it in an anonymous function:

```
x1 = tfqmr(@(x)applyOp(x,'notransp'),b,tol,maxit,M1,M2);
```

Using tfqmr with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

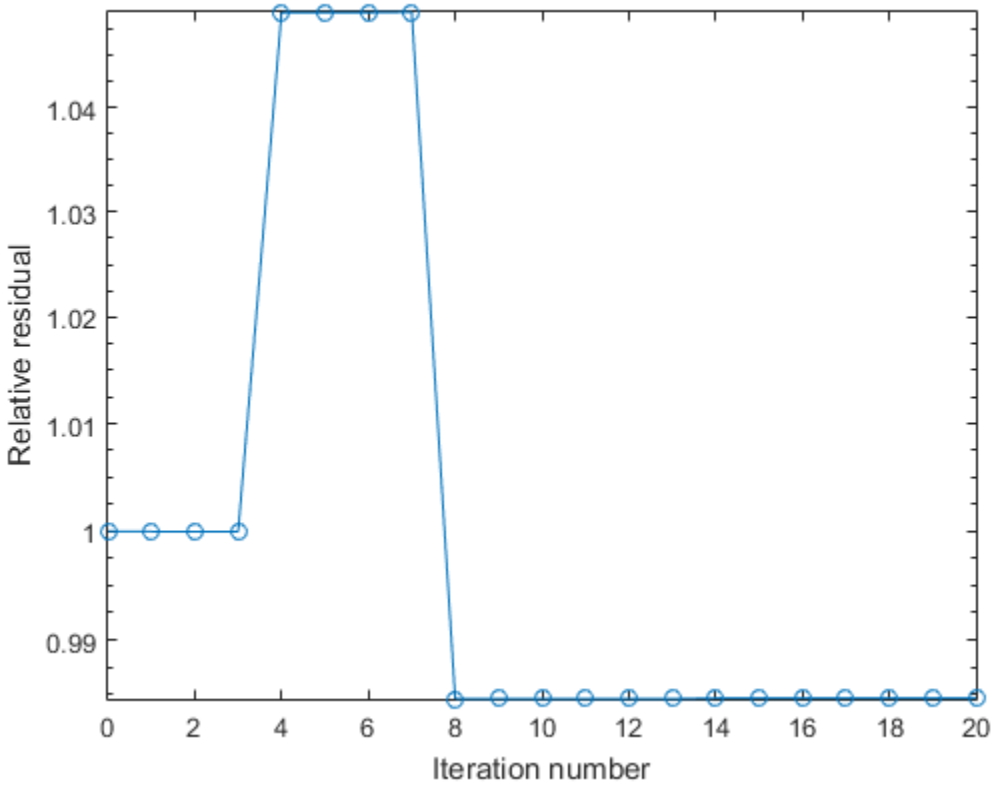
Use `tfqmr` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = tfqmr(A,b,tol,maxit);
```

`fl0` is 1 because `tfqmr` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The seventeenth iterate is the best approximate solution and is the one returned as indicated by `it0 = 17`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `tfqmr`.

```
semilogy(0:maxit,rv0(1:maxit+1)/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



Note that like `bicgstab`, `tfqmr` keeps track of half iterations. The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
```

```
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

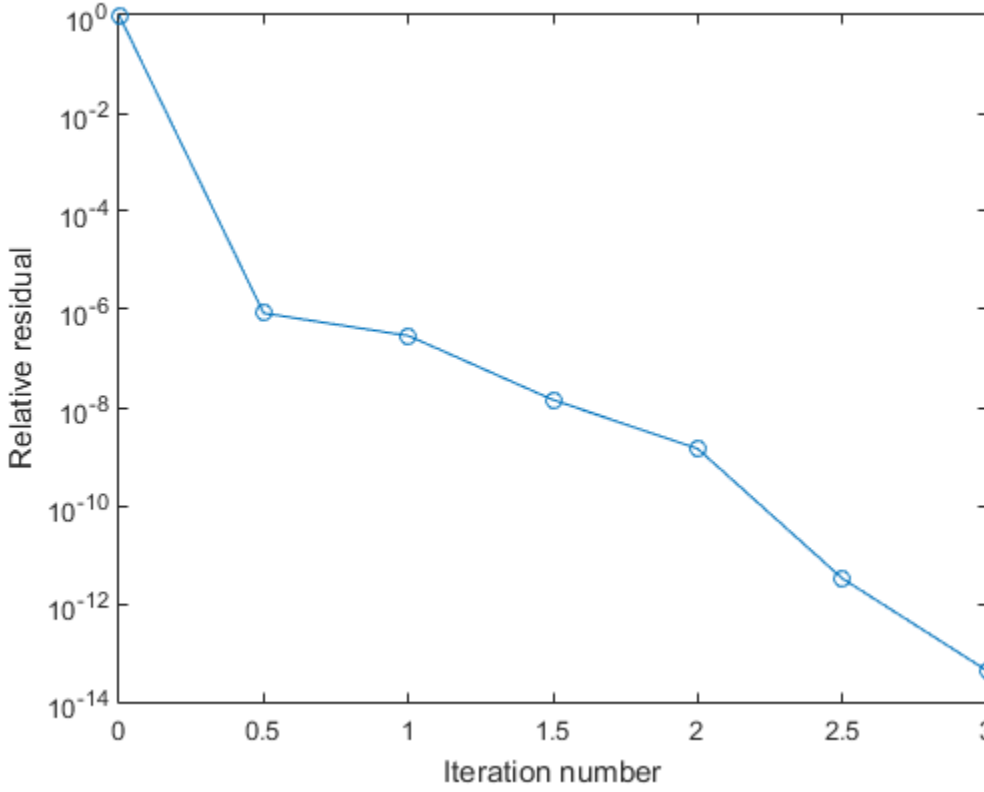
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = tfqmr(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `tfqmr` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

You can follow the progress of `tfqmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:0.5:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



See Also

`bicg` | `bicgstab` | `bicgstabl` | `cgs` | `function_handle` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

tic

Start stopwatch timer

Syntax

```
tic
timerVal = tic
```

Description

`tic` starts a stopwatch timer to measure performance. The function records the internal time at execution of the `tic` command. Display the elapsed time with the `toc` function.

`timerVal = tic` returns the value of the internal timer at the execution of the `tic` command, so that you can record time for simultaneous time spans.

Output Arguments

`timerVal`

Value of the internal timer at the execution of the `tic` command. This value is used as an input argument for a subsequent call to `toc`. You should not rely on the meaning of this value.

Examples

Measure time to generate two random matrices and compute element-by-element multiplication of their transposes.

```
tic
A = rand(12000, 4400);
B = rand(12000, 4400);
toc
C = A'.*B';
```

```
toc
```

Measure how the time required to solve a linear system varies with the order of a matrix:

```
t = zeros(1,100);
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic;
    x = A\b;
    t(n) = toc;
end
plot(t)
```

Measure multiple time spans simultaneously using two pairs of `tic/toc` calls. To do this, measure the minimum and average time to compute a summation of Bessel functions:

```
REPS = 1000;   minTime = Inf;   nsum = 10;
tic;  % TIC, pair 1

for i=1:REPS
   tStart = tic;  % TIC, pair 2
   total = 0;
   for j=1:nsum
      total = total + besselj(j,REPS);
   end

   tElapsed = toc(tStart);  % TOC, pair 2
   minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;  % TOC, pair 1
```

More About

Tips

- Consecutive `tic` commands overwrite the internally recorded starting time.
- The `clear` function does not reset the starting time recorded by a `tic` command.
- The following actions result in unexpected output:
  - Using `tic` and `toc` to time `timeit`

- Using `tic` and `toc` within a function timed by `timeit`

See Also

`clock` | `cputime` | `etime` | `profile` | `timeit` | `toc`

Introduced before R2006a

timeit

Measure time required to run function

Syntax

```
t = timeit(f)
t = timeit(f,numOutputs)
```

Description

`t = timeit(f)` measures the typical time (in seconds) required to run the function specified by the function handle `f`.

`t = timeit(f,numOutputs)` calls `f` with the desired number of outputs, `numOutputs`. By default, `timeit` calls the function `f` with one output (or no outputs, if the function does not return any outputs).

Examples

Determine Time to Obtain Current Date

Use `timeit` to time a function call to `date`. This example uses a handle to a function that accepts no input.

```
f = @date;
t = timeit(f)
```

```
t =

   4.1257e-05
```

Determine Time to Compute Matrix Summation

Time the combination of several mathematical matrix operations: matrix transposition, element-by-element multiplication, and summation of columns.

```
A = rand(12000,400);
B = rand(400,12000);
f = @() sum(A.'.*B, 1);
timeit(f)
```

```
ans =

    0.0659
```

Compare Time to Run svd with Multiple Outputs

Determine how long it takes to run `svd` with one output argument, `s = svd(X)`.

```
X = rand(100);
f = @() svd(X);
t1 = timeit(f)
```

```
t1 =

    0.0021
```

Compare the results to `svd` with three output arguments, `[U,S,V] = svd(X)`.

```
t2 = timeit(f,3)
```

```
t2 =

    0.0043
```

Compare Time to Execute Custom Preallocation to Calling `zeros`

Create a short function to allocate a matrix using nested loops. Preallocating an array using a nested loop is inefficient, but is shown here for illustrative purposes.

```
function mArr = preAllocFcn(x,y)
for m = 1:x
    for n = 1:y
        mArr(m,n) = 0;
    end
```

```
end
end
```

Compare the time to allocate zeros to a matrix using nested loops and using the `zeros` function.

```
x = 1000;
y = 500;
g = @() preAllocFcn(x,y);
h = @() zeros(x,y);
diffRunTime = timeit(g)-timeit(h)
```

```
diffRunTime =

    0.2004
```

Input Arguments

`f` — function to be measured

function handle

Function to be measured, specified as a function handle. `f` is either a handle to a function that takes no input, or a handle to an anonymous function with an empty argument list.

`numOutputs` — Number of desired outputs from `f`

integer

Number of desired outputs from `f`, specified as an integer. If the function specified by `f` has a variable number of outputs, `numOutputs` specifies which syntax `timeit` uses to call the function. For example, the `svd` function returns a single output, `s`, or three outputs, `[U,S,V]`. Set `numOutputs` to `1` to time the `s = svd(X)` syntax, or set it to `3` to time the `[U,S,V] = svd(X)` syntax.

More About

Tips

- The following actions result in unexpected output:
  - Using `timeit` between `tic` and `toc`

- Using `timeit` to time a function that includes calls to `tic` and `toc`
- Using `timeit` recursively

Algorithms

`timeit` calls the specified function multiple times, and computes the median of the measurements.

- Anonymous Functions
- Analyzing Your Program's Performance
- MATLAB Performance Measurement White Paper on MATLAB Central File Exchange

See Also

`cputime` | `function_handle` | `tic` | `toc`

toc

Read elapsed time from stopwatch

Syntax

```
toc
elapsedTime = toc
toc(timerVal)
elapsedTime = toc(timerVal)
```

Description

`toc` reads the elapsed time from the stopwatch timer started by the `tic` function. The function reads the internal time at the execution of the `toc` command, and displays the elapsed time since the most recent call to the `tic` function that had no output, in seconds.

`elapsedTime = toc` returns the elapsed time in a variable.

`toc(timerVal)` displays the time elapsed since the `tic` command corresponding to `timerVal`.

`elapsedTime = toc(timerVal)` returns the elapsed time since the `tic` command corresponding to `timerVal`.

Input Arguments

`timerVal`

Value of the internal timer saved from a previous call to the `tic` command.

Output Arguments

`elapsedTime`

Scalar `double` representing the time elapsed between `tic` and `toc` commands, in seconds.

Examples

Measure time to generate two random matrices and compute element-by-element multiplication of their transposes.

```
tic
A = rand(12000, 4400);
B = rand(12000, 4400);
toc
C = A'.*B';
toc
```

Measure how the time required to solve a linear system varies with the order of a matrix:

```
t = zeros(1,100);
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic;
    x = A\b;
    t(n) = toc;
end
plot(t)
```

Measure multiple time spans simultaneously using two pairs of `tic/toc` calls. To do this, measure the minimum and average time to compute a summation of Bessel functions:

```
REPS = 1000;   minTime = Inf;   nsum = 10;
tic;  % TIC, pair 1

for i=1:REPS
   tStart = tic;  % TIC, pair 2
   total = 0;
   for j=1:nsum
```

```
      total = total + besselj(j,REPS);
   end

   tElapsed = toc(tStart);  % TOC, pair 2
   minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;  % TOC, pair 1
```

More About

Tips

- Consecutive calls to the `toc` function with no input return the elapsed since the most recent `tic`. Therefore, you can take multiple measurements from a single point in time.

  Consecutive calls to the `toc` function with the same `timerVal` input return the elapsed time since the `tic` function call that corresponds to that input.
- The following actions result in unexpected output:
  - Using `tic` and `toc` to time `timeit`
  - Using `tic` and `toc` within a function timed by `timeit`

See Also

`clock` | `cputime` | `etime` | `profile` | `tic` | `timeit`

Introduced before R2006a

Tiff class

MATLAB Gateway to LibTIFF library routines

Description

The `Tiff` class represents a connection to a Tagged Image File Format (TIFF) file and provides access to many of the capabilities of the LibTIFF library. Use the methods of the `Tiff` object to call routines in the LibTIFF library. While you can use the `imread` and `imwrite` functions to read and write TIFF files, the `Tiff` class offers capabilities that these functions don't provide, such as reading subimages, writing tiles and strips of image data, and modifying individual TIFF tags.

In most cases, the syntax of the `Tiff` method is similar to the syntax of the corresponding LibTIFF library function. To get the most out of the `Tiff` object, you must be familiar with the LibTIFF API, as well as the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

MATLAB supports LibTIFF version 4.0.0.

For copyright information, see the `libtiffcopyright.txt` file.

Construction

`obj = Tiff(filename,mode)` creates a `Tiff` object associated with the TIFF file `filename`. `mode` specifies the type of access to the file.

A TIFF file is made up of one or more image file directories (IFDs). An IFD contains image data and associated metadata. IFDs can also contain subIFDs which also contain image data and metadata. When you open a TIFF file for reading, the Tiff object makes the first IFD in the file the *current* IFD. `Tiff` methods operate on the current IFD. You can use `Tiff` object methods to navigate among the IFDs and the subIFDs in a TIFF file.

When you open a TIFF file for writing or appending, the `Tiff` object automatically creates a IFD in the file for writing subsequent data. This IFD has all the default values specified in TIFF Revision 6.0.

When creating a new TIFF file, before writing any image to the file, you must create certain required fields (tags) in the file. These tags include `ImageWidth`, `ImageHeight`,

`BitsPerSample`, `SamplesPerPixel`, `Compression`, `PlanarConfiguration`, and `Photometric`. If the image data has a stripped layout, the IFD contains the `RowsPerStrip` tag. If the image data has a tiled layout, the IFD contains the `TileWidth` and `TileHeight` tags. Use the `setTag` method to define values for these tags.

Input Arguments

`filename`

Text string specifying name of file.

`mode`

One of the following text strings specifying the type of access to the TIFF file.

Supported Values

Parameter	Description
`'r'`	Open file for reading (default)
`'w'`	Open file for writing; discard existing contents
`'w8'`	Open file for writing a BigTIFF file; discard existing contents
`'a'`	Open or create file for writing; append data to end of file.
`'r+'`	Open (do not create) file for reading and writing

Properties

`Compression`

Specify scheme used to compress image data

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

`None`
`CCITTRLE (Read-only)`

CCITTFax3
CCITTFax4
LZW
JPEG
CCITTRLEW (Read-only)
PackBits
SGILog
SGILog24
Deflate
AdobeDeflate (Same as deflate)

Example: Set the `Compression` tag to the value `JPEG`. Note how you use the property to specify the value.

```
tiffobj.setTag('Compression', Tiff.Compression.JPEG);
```

`ExtraSamples`

Describe extra components

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

This field is required if there are extra channels in addition to the usual colormetric channels.

Supported Values

Value of `ExtraSamples`	Description
`Unspecified`	Unspecified data
`AssociatedAlpha`	Associated alpha (pre-multiplied)
`UnassociatedAlpha`	Unassociated alpha data

Example: Set the `ExtraSamples` tag to the value `AssociatedAlpha`. Note how you use the property to specify the value.

```
tiffobj.setTag('ExtraSamples', Tiff.ExtraSamples.AssociatedAlpha);
```

See Also “Specify Tiff object properties and describe alpha channel” on page 1-8221

Group3Options

Options for Group 3 Fax Compression

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

This property is also referred to as `Fax3` and `T4Options`. This value is a bit mask controlled by the first three bits.

Supported Values

Supported values include the following.

`Encoding2D`	Bit 0 is 1. This specifies two-dimensional coding. If more than one strip is specified, each strip must begin with a one-dimensionally coded line. That is, `RowsPerStrip` should be a multiple of Parameter K, as documented in the CCITT specification.
`Uncompressed`	Bit 1 is 1. This specifies an uncompressed mode when encoding.
`FillBits`	Bit 2 is 1. Fill bits are added as necessary before EOL codes such that EOL always ends on a byte boundary. This ensures an EOL-sequence of 1 byte preceded by a zero nibble, for example, `xxxx-0000 0000-0001`.

Example:

```
mask = bitor(Tiff.Group3Options.Encoding2D,Tiff.Group3Options.Uncompressed)
tiffobj.setTag('Group3Options',mask);
```

InkSet

Specify set of inks used in separated image

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

In this context, separated refers to photometric interpretation, not the planar configuration.

Supported Values

`CMYK`	Order of components: cyan, magenta, yellow, black. Usually, a value of 0 represents 0% ink coverage and a value of 255 represents 100% ink coverage for that component, but consult the TIFF specification for `DotRange`. When you specify CMYK, do not set the `InkNames` tag.
`MultiInk`	Any ordering other than CMYK. Consult the TIFF specification for `InkNames` field for a description of the inks used.

Example:

```
tiffobj.setTag('InkSet', Tiff.InkSet.CMYK);
```

`JPEGColorMode`

Specify control of YCbCr/RGB conversion

Use these values only when the photometric interpretation is YCbCr.

This property should not be used for the purpose of reading YCbCr imagery as RGB. In this case, use the RGBA interface provided by the `readRGBAImage`, `readRGBAStrip`, and `readRGBATile` methods.

Supported Values

`Raw` (default)	Keep input as separate Y, Cb, and Cr matrices.
`RGB`	Convert RGB input to YCbCr.

Example:

```
tiffobj.setTag('JPEGColorMode',Tiff/JPEGColorMode.RGB);
```

See also “Create YCbCr/JPEG image from RGB data” on page 1-8222

`Orientation`

Specify visual orientation of the image data.

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

The first row represents the top of the image, and the first column represents the left side. Support for this tag is for informational purposes only, and it does not affect how MATLAB reads or writes the image data.

Supported Values

`TopLeft`
`TopRight`
`BottomRight`
`BottomLeft`
`LeftTop`
`RightTop`
`RightBottom`
`LeftBottom`

Example:

```
tiffobj.setTag('Orientation', Tiff.Orientation.TopRight);
```

`Photometric`

Specify color space of image data

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

`MinIsWhite`
`MinIsBlack`
`RGB`
`Palette`
`Mask`
`Separated (CMYK)`
`YCbCr`
`CIELab`

ICCLab
ITULab
LogL
LogLUV
CFA
LinearRaw

Example:

```
tiffobj.setTag('Photometric', Tiff.Photometric.RGB);
```

PlanarConfiguration

Specifies how image data components are stored on disk

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

Chunky	Store component values for each pixel contiguously. For example, in the case of RGB data, the first three pixels would be stored in the file as RGBRGBRGB etc. Almost all TIFF images have contiguous planar configurations.
Separate	Store component values for each pixel separately. For example, in the case of RGB data, the red component would be stored separately in the file from the green and blue components.

Example:

```
tiffobj.setTag('PlanarConfiguration', Tiff.PlanarConfiguration.Chunky);
```

ResolutionUnit

Specify unit of measure used to interpret the `XResolution` and `YResolution` tags.

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

`None` (default)
`Inch`
`Centimeter`

Example: Set `ResolutionUnit` tag to the value `Inch`. Then, setting `XResolution` tag to 300 means pixels per inch.

```
tiffObj.setTag('ResolutionUnit', Tiff.ResolutionUnit.Inch);
tiffObj.setTag('XResolution', 300);
tiffObj.setTag('YResolution', 300);
```

`SampleFormat`

Specify how to interpret each pixel sample

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the examples.

Supported Values

`UInt` (default)
`Int`
`IEEEFP`

`Void`, `ComplexInt`, and `ComplexIEEEFP` are not supported.

Example:

```
tiffobj.setTag('SampleFormat', Tiff.SampleFormat.IEEEFP);
```

`SGILogDataFmt`

Specify control of client data for SGILog codec

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

These enumerated values should only be used when the photometric interpretation value is `LogL` or `LogLUV`. The `BitsPerSample`, `SamplesPerPixel`, and `SampleFormat` tags should not be set if the image type is `LogL` or `LogLuv`. The choice of `SGILogDataFmt` will set these tags automatically. The `Float` and `Bits8` settings imply a `SamplesPerPixel` value of 3 for `LogLUV` images, but only 1 for `LogL` images.

Supported Values

`Float`	Single precision samples
`Bits8`	`uint8` samples (read only)

This tag can be set only once per instance of a `LogL`/`LogLuv` Tiff image object instance.

Example:

```
tiffobj = Tiff('example.tif','r');
tiffobj.setDirectory(3); % image three is a LogLuv image
tiffobj.setTag('SGILogDataFmt', Tiff.SGILogDataFmt.Float);
imdata = tiffobj.read();
```

`SubFileType`

Specify type of image

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

`SubFileType` is a bitmask that indicates the type of the image.

Supported Values

`Default`	Default value for single image file or first image.
`ReducedImage`	The current image is a thumbnail or reduced-resolution image that typically would be found in a sub-IFD.
`Page`	The image is a single image of a multi-image (or multipage) file.
`Mask`	The image is a transparency mask for another image in the file. The photometric interpretation value must be Photometric.Mask.

Example:

```
tiffobj.setTag('SubFileType', Tiff.SubFileType.Mask);
```

TagID

List of recognized TIFF tag names with their ID numbers

This property identifies all the supported TIFF tags with their ID numbers. Use this property to specify a tag when using the `setTag` method. For example, `Tiff.TagID.ImageWidth` returns the ID of the `ImageWidth` tag. To get a list of the names of supported tags, use the `getTagNames` method.

Example:

```
tiffobj.setTag(Tiff.TagID.ImageWidth, 300);
```

Thresholding

Specifies technique used to convert from gray to black and white pixels.

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

`BiLevel` (default)
`HalfTone`
`ErrorDiffuse`

Example:

```
tiffobj.setTag('Thresholding', Tiff.Thresholding.HalfTone);
```

YCbCrPositioning

Specify relative positioning of chrominance samples

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

This property identifies all supported values for the `YCbCrPositioning` tag.

Supported Values

`Centered`	Specify for compatibility with industry standards such as PostScript Level 2

`Cosited`	Specify for compatibility with most digital video standards such as CCIR Recommendation 601-1.

Example:

```
tiffobj.setTag('YCbCrPositioning', Tiff.YCbCrPositioning.Centered);
```

Methods

close	Close Tiff object
computeStrip	Index number of strip containing specified coordinate
computeTile	Index number of tile containing specified coordinates
currentDirectory	Index of current IFD
getTag	Value of specified tag
getTagNames	List of recognized TIFF tags
getVersion	LibTIFF library version
isTiled	Determine if tiled image
lastDirectory	Determine if current IFD is last in file
nextDirectory	Make next IFD current IFD
numberOfStrips	Total number of strips in image
numberOfTiles	Total number of tiles in image

read	Read entire image
readEncodedStrip	Read data from specified strip
readEncodedTile	Read data from specified tile
readRGBAImage	Read image using RGBA interface
readRGBAStrip	Read strip data using RGBA interface
readRGBATile	Read tile data using RGBA interface
rewriteDirectory	Write modified metadata to existing IFD
setDirectory	Make specified IFD current IFD
setSubDirectory	Make subIFD specified by byte offset current IFD
setTag	Set value of tag
write	Write entire image
writeDirectory	Create new IFD and make it current IFD
writeEncodedStrip	Write data to specified strip
writeEncodedTile	Write data to specified tile

Examples

Create New TIFF File Using `Tiff` object

Create a new file called `myfile.tif`. To run this example, your directory must be writable.

```
t = Tiff('myfile.tif', 'w');
```

Close the Tiff object.

```
t.close();
```

Specify Tiff object properties and describe alpha channel

Create an array of data, `data`, that contains colormetric channels and an alpha channel.

```
rgb = imread('example.tif');
numrows = size(rgb,1);
numcols = size(rgb,2);
alpha = 255*ones([numrows numcols], 'uint8');
data = cat(3,rgb,alpha);
```

Create a `Tiff` object, `t`, and set the object properties. Set the value of the `ExtraSamples` tag because the data contains the alpha channel in addition to the colormetric channels.

```
t = Tiff('myfile.tif','w');
t.setTag('Photometric',Tiff.Photometric.RGB);
t.setTag('Compression',Tiff.Compression.None);
t.setTag('BitsPerSample',8);
t.setTag('SamplesPerPixel',4);
t.setTag('SampleFormat',Tiff.SampleFormat.UInt);
t.setTag('ExtraSamples',Tiff.ExtraSamples.Unspecified);
t.setTag('ImageLength',numrows);
t.setTag('ImageWidth',numcols);
t.setTag('TileLength',32);
t.setTag('TileWidth',32);
t.setTag('PlanarConfiguration',Tiff.PlanarConfiguration.Chunky);
```

Write the data to the `Tiff` object.

```
t.write(data);
```

```
t.close();
```

Create YCbCr/JPEG image from RGB data

Get RGB data.

```
rgb = imread('example.tif');
```

Create a `Tiff` object, `t`, and set the object properties. Specify RGB input data using the `JPEGColorMode` property.

```
t = Tiff('myfile.tif','w');
t.setTag('Photometric',Tiff.Photometric.YCbCr);
t.setTag('Compression',Tiff.Compression.JPEG);
t.setTag('YCbCrSubSampling',[2 2]);
t.setTag('BitsPerSample',8);
t.setTag('SamplesPerPixel',3);
t.setTag('SampleFormat',Tiff.SampleFormat.UInt);
t.setTag('ImageLength',size(rgb,1));
t.setTag('ImageWidth',size(rgb,2));
t.setTag('TileLength',32);
t.setTag('TileWidth',32);
t.setTag('PlanarConfiguration',Tiff.PlanarConfiguration.Chunky);
t.setTag('JPEGColorMode',Tiff.JPEGColorMode.RGB);
t.setTag('JPEGQuality',75);
```

Write the data to the `Tiff` object.

```
t.write(rgb);
t.close();
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

See Also

`imread` | `imwrite` | `imfinfo`

time

Convert time of calendar duration to duration

Syntax

```
t = time(d)
```

Description

`t = time(d)` returns the time components of the calendar duration values in `d` as durations. The `t` output is the same size as `d`.

Examples

Convert Time of Calendar Durations to Durations

Create an array of calendar durations.

```
d = caldays(8:10) + hours(1.2345)
```

```
d =

    8d 1h 14m 4.2s    9d 1h 14m 4.2s   10d 1h 14m 4.2s
```

Convert the time components of the array to durations.

```
t = time(d)
```

```
t =

   01:14:04   01:14:04   01:14:04
```

View the data type of `t`.

```
whos t
  Name      Size            Bytes  Class       Attributes

  t         1x3               144  duration
```

Input Arguments

d — Input calendar duration

scalar | vector | matrix | multidimensional array

Input calendar duration, specified as a scalar, vector, matrix, or multidimensional `calendarDuration` array.

See Also

`caldays` | `calmonths` | `calquarters` | `calweeks` | `calyears` | `duration`

Introduced in R2014b

timeofday

Convert time of datetime to duration

Syntax

```
d = timeofday(t)
```

Description

`d = timeofday(t)` returns an array of durations equal to the time components of the datetime values in `t`. The `d` output is the same size as `t`.

```
timeofday(t) = t - dateshift(t,'start','day')
```

Examples

Convert Time of Datetimes to Durations

Create a `datetime` array.

```
t = datetime('now') + hours(1:3)
```

```
t =

   23-Feb-2015 10:58:45   23-Feb-2015 11:58:45   23-Feb-2015 12:58:45
```

Convert the time components of the array to durations.

```
d = timeofday(t)
```

```
d =

   10:58:45   11:58:45   12:58:45
```

View the data type of `d`.

```
whos d
```

```
  Name      Size            Bytes  Class       Attributes

  d         1x3               144  duration
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

`dateshift` | `duration` | `hms` | `hour` | `minute` | `second`

Introduced in R2014b

timer class

Create object to schedule execution of MATLAB commands

Description

Use a `timer` object to schedule the execution of MATLAB commands one or multiple times. If you schedule the timer to execute multiple times, you can define the time between executions and how to handle queuing conflicts.

The `timer` object uses callback functions to execute commands. Callback functions execute code during some event. For the `timer` object, you can specify the callback function as a function handle or as a string. If the callback function is a string, MATLAB evaluates it as executable code. The timer object supports callback functions when a timer starts (`StartFcn`), executes (`TimerFcn`), stops (`StopFcn`), or encounters an error (`ErrorFcn`).

Note: The `timer` object is subject to the limitations of your hardware, operating system, and software. Avoid using timer objects for real-time applications.

Construction

`t = timer` creates an empty `timer` object to schedule execution of MATLAB commands. An error occurs if the timer starts and `TimerFcn` is not defined.

`t = timer(Name,Value)` creates a `timer` object with additional options that you specify using one or more Name,Value pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The argument name, `Name`, corresponds to a `timer` property name. In the constructor, the property values are specified using Name,Value pair arguments.

`'BusyMode'`

String that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue
`'queue'`	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts `Period` property to manage tasks in execution queue

See "Handling Timer Queuing Conflicts" for more information.

Default: `'drop'`

`'ErrorFcn'`

String, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls `StopFcn`.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.

- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

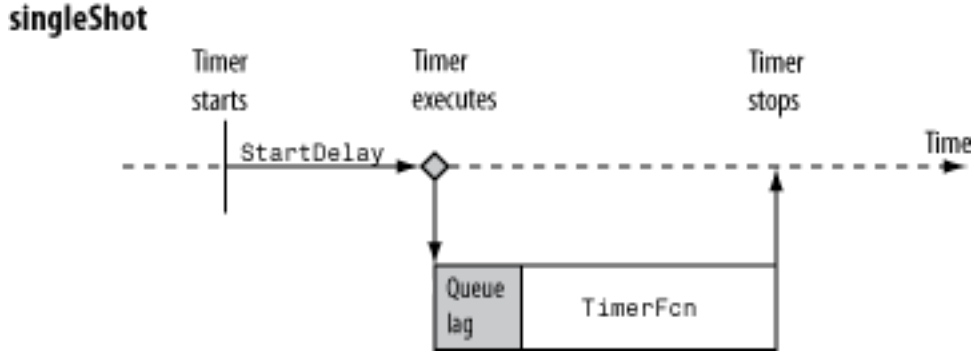
For more information, see "Timer Callback Functions".

`'ExecutionMode'`

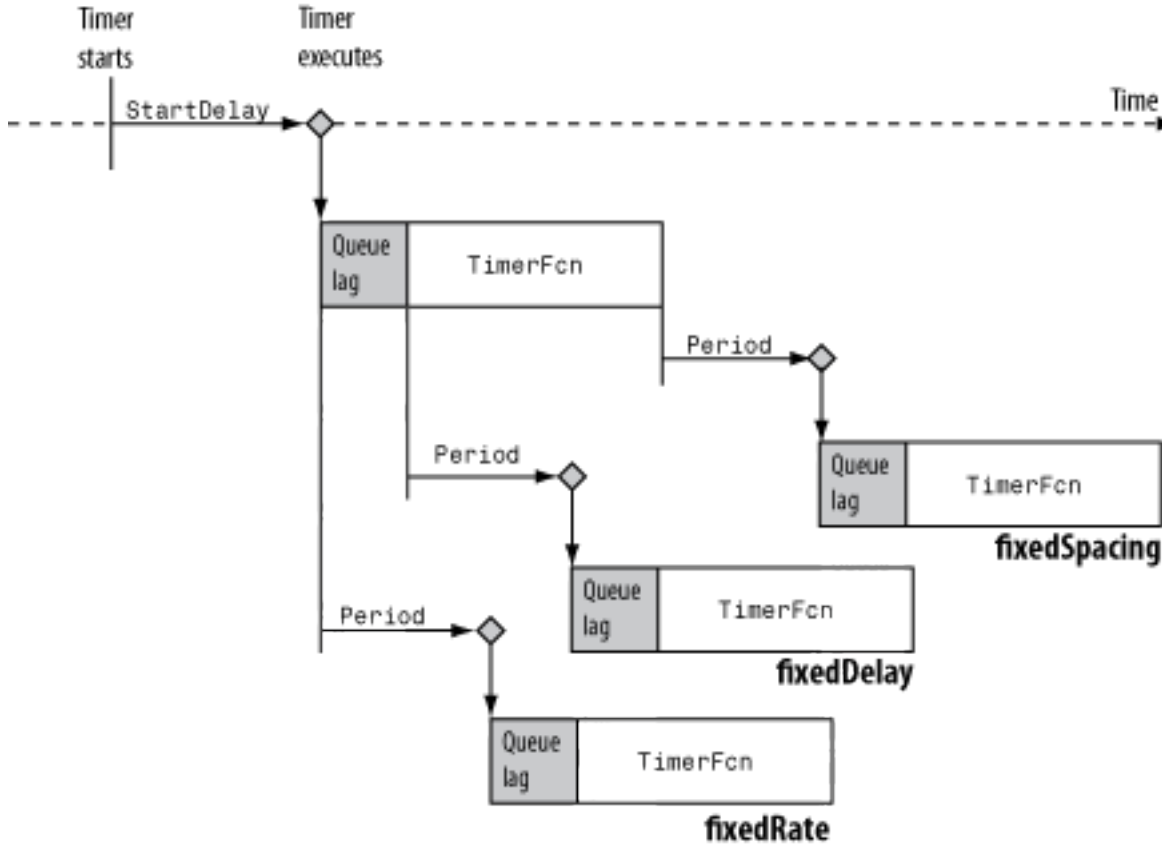
String that defines how the `timer` object schedules timer events. When `Running='on'`, `ExecutionMode` is read only. This table summarizes the execution modes.

Execution Mode	Time `Period` Start Point
`'singleShot'`	In this mode, the timer callback function is only executed once. Therefore, the `Period` property has no effect. This is the default execution mode.
`'fixedRate'`	Starts immediately after the timer callback function is added to the MATLAB execution queue
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`'Name'`

String representing the `timer` name.

Default: `'timer-`*`i`*`'`, where *`i`* is a number indicating the *`i`*th timer object created this session. To reset *`i`* to 1, execute the `clear classes` command.

`'ObjectVisibility'`

String with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates. The `timerfind` function

does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

`'Period'`

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

`'StartDelay'`

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

`'StartFcn'`

String, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

`'StopFcn'`

String, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

`'Tag'`

String that represents a label for the object.

`'TasksToExecute'`

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

`'TimerFcn'`

String, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

`'UserData'`

Generic field for data that you want to add to the object.

Properties

`AveragePeriod`

Average time in seconds between `TimerFcn` executions since the timer started. Value is `NaN` until timer executes two timer callbacks.

`InstantPeriod`

The time in seconds between the last two executions of `TimerFcn`. Value is `NaN` until timer executes two timer callbacks.

`Running`

String defined as `'off'` or `'on'`, indicating whether the timer is currently executing callback functions.

`TasksExecuted`

The number of times the timer called `TimerFcn` since the timer started.

`Type`

String that identifies the object type.

Methods

delete	Remove timer object from memory
get	Query property values for timer object
isvalid	Determine timer object validity
set	Set property values for timer object
start	Start timer object
startat	Schedule timer to fire at specified time
stop	Stop timer object
timerfind	Find timer object
timerfindall	Find timer object, regardless of visibility
wait	Block command prompt until timer stops running

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Display Message Using Timer

Display a message using an anonymous function as a callback function. It is important to note that the first two arguments the callback function passes are a handle to the

timer object and an event structure. Even if the function doesn't use these arguments, the function definition requires them.

Wait 3 seconds, and then display the message '3 seconds have elapsed'.

```
t = timer;
t.StartDelay = 3;
t.TimerFcn = @(myTimerObj, thisEvent)disp('3 seconds have elapsed');
start(t)
```

```
3 seconds have elapsed
```

Suppose the function does not require the timer or event object. Use the tilde (~) operator to ignore the inputs.

```
t.TimerFcn = @(~,~) disp('3 seconds have elapsed');
start(t)
```

```
3 seconds have elapsed
```

Delete the timer object.

```
delete(t)
```

Execute Callback Function Multiple Times

Display the event and date/time output when the timer starts, fires, and stops. The timer callback function will be executed 3 times with 2 seconds between calls. The first two arguments the callback function passes are a handle to the timer object and an event structure. The event structure contains two fields: `Type` is a string that identifies the type of event that caused the callback, and `Data` is a structure that contains a date time vector of when the event occurred.

```
t = timer;
t.StartFcn = @(~,thisEvent)disp([thisEvent.Type ' executed '...
    datestr(thisEvent.Data.time,'dd-mmm-yyyy HH:MM:SS.FFF')]);
t.TimerFcn = @(~,thisEvent)disp([thisEvent.Type ' executed '...
     datestr(thisEvent.Data.time,'dd-mmm-yyyy HH:MM:SS.FFF')]);
t.StopFcn = @(~,thisEvent)disp([thisEvent.Type ' executed '...
    datestr(thisEvent.Data.time,'dd-mmm-yyyy HH:MM:SS.FFF')]);
t.Period = 2;
t.TasksToExecute = 3;
t.ExecutionMode = 'fixedRate';
start(t)
```

```
StartFcn executed 14-Mar-2013 09:08:50.865
TimerFcn executed 14-Mar-2013 09:08:50.865
TimerFcn executed 14-Mar-2013 09:08:52.865
TimerFcn executed 14-Mar-2013 09:08:54.866
StopFcn executed 14-Mar-2013 09:08:54.869
```

Delete the timer object.

```
delete(t)
```

Define Custom Callback Functions

Create a timer object to remind yourself to take 30-second ergonomic breaks every 10 minutes over the course of 8 hours.

Create a function in a file named `createErgoTimer.m` that returns a timer object. Have this file include three local functions to handle timer start, execute, and stop tasks.

```
function t = createErgoTimer()
secondsBreak = 30;
secondsBreakInterval = 600;
secondsPerHour = 60^2;
secondsWorkTime = 8*secondsPerHour;

t = timer;
t.UserData = secondsBreak;
t.StartFcn = @ergoTimerStart;
t.TimerFcn = @takeBreak;
t.StopFcn = @ergoTimerCleanup;
t.Period = secondsBreakInterval+secondsBreak;
t.StartDelay = t.Period-secondsBreak;
t.TasksToExecute = ceil(secondsWorkTime/t.Period);
t.ExecutionMode = 'fixedSpacing';
end
```

Using `StartDelay` allows the timer to start without directing you to take a break immediately. Set the execution mode to `'fixedSpacing'` so that `10` minutes and `30` seconds (`t.Period`) elapses after the completion of a `TimerFcn` execution. This allows you to stretch for `30` seconds before the start of the next `10` minute interval.

In the `createErgoTimer.m` file, add a local function to handle the tasks associated with starting the timer. By default, the `timer` object passes itself and event data to the callback function. The function disregards the event data.

```
function ergoTimerStart(mTimer,~)
secondsPerMinute = 60;
secondsPerHour = 60*secondsPerMinute;
str1 = 'Starting Ergonomic Break Timer.  ';
str2 = sprintf('For the next %d hours you will be notified',...
    round(mTimer.TasksToExecute*(mTimer.Period + ...
    mTimer.UserData)/secondsPerHour));
str3 = sprintf(' to take a %d second break every %d minutes.',...
    mTimer.UserData, (mTimer.Period - ...
    mTimer.UserData)/secondsPerMinute);
disp([str1 str2 str3])
end
```

Add a local function to handle the tasks associated with executing the timer. The `TimerFcn` callback should tell you to take a 30 second break.

```
function takeBreak(mTimer,~)
disp('Take a 30 second break.')
end
```

Add a local function to handle the tasks associated with stopping the timer.

```
function ergoTimerCleanup(mTimer,~)
disp('Stopping Ergonomic Break Timer.')
delete(mTimer)
end
```

Deleting the timer object removes it from memory.

From the command line, call the `createErgoTimer` function to create and start a timer.

```
t = createErgoTimer;
start(t)
```

```
Starting Ergonomic Break Timer.  For the next 8 hours you will be notified to take a 30
```

Every 10 minutes, you will be reminded to take a 30 second break.

```
Take a break.
```

You can leave the timer running for 8 hours or stop it manually. Recall that you included the task of deleting the timer from memory in the `StopFcn` callback.

```
stop(t)
```

```
Stopping Ergonomic Break Timer.
```

Tips

- To force the execution of the callback functions in the event queue, include a call to the `drawnow` function in your code. The `drawnow` function flushes the event queue.

See Also

`function_handle`

More About

- Timer Callback Functions
- "Handling Timer Queuing Conflicts"
- "Ignore Function Outputs"
- Property Attributes

Introduced before R2006a

delete

Class: timer

Remove timer object from memory

Syntax

```
delete(t)
```

Description

`delete(t)` removes the timer object, `t`, from memory. If `t` is an array of timer objects, `delete` removes all the objects from memory.

When you delete a timer object, it becomes invalid and you cannot reuse it. If multiple references to a timer object exist in the workspace, deleting the timer object invalidates the remaining references. To remove invalid timer objects references from the workspace, use the `clear` command.

Tips

- Use the `isvalid` method to determine if a timer object exists in memory, but is not cleared from the workspace.
- Use the `timerfind` and `timerfindall` methods to return timer objects currently existing in memory. This approach is useful if the reference to the timer object is cleared from the workspace (using the `clear` command), but not deleted from memory.

Input Arguments

`t`

Object of class `timer`.

Examples

Delete a Timer

Create and start a timer that generates a 10-by-10 array of random numbers.

```
t = timer('TimerFcn','rand(10);');
start(t)
```

Delete the timer from memory.

```
delete(t)
```

Call the `whos` function to see if a reference still exists in the workspace.

```
whos
```

```
  Name        Size              Bytes  Class     Attributes

  ans        10x10               800  double
  t           1x1                104  timer
```

Try to restart the timer.

```
start(t)
```

```
Error using timer/start (line 27)
Invalid timer object. This object has been deleted and should be removed from your work
```

The timer cannot be restarted.

Clear the timer object reference from the workspace.

```
clear t
```

Delete Multiple Timers Using timerfind

Use `delete` with the `timerfind` method to remove all visible timers from memory. This is an alternative to deleting individual timers by variable name.

Create and start three timers that compute the sine, cosine and tangent of `pi/4`.

```
t1 = timer('TimerFcn','sin(pi/4);');
t2 = timer('TimerFcn','cos(pi/4);');
t3 = timer('TimerFcn','tan(pi/4);');
```

Delete the timers from memory using `timerfind`. This removes all visible timer objects from memory.

```
delete(timerfind)
```

See Also

`timer` | `isvalid` | `timerfind` | `timerfindall`

Introduced before R2006a

get

Class: timer

Query property values for timer object

Syntax

```
get(t)
V = get(t)
V = get(t,propName)
```

Description

`get(t)` queries property values for timer object, `t`, and displays all property names and current values. `t` must be a scalar timer object.

`V = get(t)` queries property values for timer object, `t` and returns a structure, `V`, where each field name is the name of a property of `t` and each field contains the value of that property. If `t` is an M-by-1 vector of timer objects, `V` is an M-by-1 array of structures.

`V = get(t,propName)` returns the value, `V`, of the timer object property specified in `propName`. If `propNames` is a vector cell array of N property names, and `t` is a vector of M timer objects, `v` is an M-by-N cell array of property values.

Input Arguments

`t`

Object of class `timer`.

`propName`

String enclosed in single quotation marks that specifies a `timer` property.

Examples

Display All Properties of Timer

```
t = timer;
get(t)
```

```
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-1'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 0
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
       TasksExecuted: 0
      TasksToExecute: Inf
            TimerFcn: ''
                Type: 'timer'
            UserData: []
```

Delete the timer from memory.

```
delete(t)
```

Obtain Properties for Array of Timers

Create three timers.

```
t1 = timer;
t2 = timer;
t3 = timer;
```

Get properties of an array of timers.

```
V = get([t1,t2,t3])
```

```
V =
```

```
3x1 struct array with fields:

    AveragePeriod
    BusyMode
    ErrorFcn
    ExecutionMode
    InstantPeriod
    Name
    ObjectVisibility
    Period
    Running
    StartDelay
    StartFcn
    StopFcn
    Tag
    TasksExecuted
    TasksToExecute
    TimerFcn
    Type
    UserData
```

Delete the timers from memory.

```
delete([t1,t2,t3])
```

Obtain Single Property for Timer

Create a timer and determine if it is running.

```
t = timer;
get(t,'Running')
```

```
ans =

off
```

Delete the timer from memory.

```
delete(t)
```

Obtain Specified Properties for Array of Timers

Create three timers.

```
t1 = timer;
t2 = timer;
t3 = timer;
```

Obtain name, period, and running property values from the array of timers.

```
V = get([t1,t2,t3],{'Name','Running','Period'})
```

```
V =

    'timer-1'    'off'    [1]
    'timer-2'    'off'    [1]
    'timer-3'    'off'    [1]
```

Delete the timers

```
delete([t1,t2,t3])
```

Alternatives

You can also use dot notation can also be used to query timer object properties. For example, `t.Running` returns the same value as `get(t,'Running')`.

See Also

`timer` | `set`

Introduced before R2006a

isvalid

Class: timer

Determine timer object validity

Syntax

```
validCheck = isvalid(t)
```

Description

`validCheck = isvalid(t)` determines timer object validity and returns a logical array, `validCheck`, that contains a `0` where the elements of `t` are invalid timer objects and a `1` where the elements of `t` are valid timer objects.

An invalid timer object is an object that is deleted from memory using `delete` and cannot be reused. Use the `clear` command to remove an invalid timer object from the workspace.

Tips

- To return timer objects existing in memory, use the `timerfind` and `timerfindall` methods. This practice is useful if the reference to the timer object is cleared from the workspace (using the `clear` function), but the object has not been deleted from memory.

Input Arguments

`t`

Object or array of objects of class `timer`

Examples

Check Timer Object Validity

Create a timer object. While this timer object is valid, it cannot start because a `TimerFcn` is not defined.

```
t = timer;
out = isvalid(t)
```

```
out =

     1
```

Delete the timer object, making it invalid.

```
delete(t)
out1 = isvalid(t)
```

```
out1 =

     0
```

See Also

`timer` | `delete` | `timerfind` | `timerfindall`

Introduced before R2006a

set

Class: timer

Set property values for timer object

Syntax

```
set(t)
propStruct = set(t)

set(t,Name)
propCell = set(t,Name)
```

Description

`set(t)` displays the property names and their possible values for all configurable properties of timer object `t`.

`propStruct = set(t)` returns the values in a `struct`.

`set(t,Name)` displays the possible values for the specified property, `Name`, of timer object, `t`.

`propCell = set(t,Name)` returns the values in a cell.

`set(t,Name,Value)` sets the properties specified by one or more Name,Value pair arguments. `t` can be a single timer object or a vector of timer objects, in which case `set` configures the property values for all timer objects, `t`.

`set(t,S)` configures the properties of `t`, with the values specified in `S`, where `S` is a structure whose field names are object property names.

`set(t,PN,PV)` configures the properties specified in the cell array of strings, `PN`, to the corresponding values in the cell array `PV`, for the timer object `t`.

Input Arguments

`t`

Object of class `timer`.

`S`

Structure whose field names are `timer` property names.

`PN,PV`

Cell array of strings, `PN`, and corresponding cell array of values, `PV`. The cell array of `PN` must be a 1-by-N or N-by-1 array. If `t` is an array of timer objects, `PV` can be an M-by-N cell array, where M is equal to the length of `t` and N is equal to the length of `PN`. In this case, each timer object is updated with a different set of values for the list of property names contained in `PN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'BusyMode'`

String that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

`BusyMode` Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
'queue'	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts Period property to manage tasks in execution queue

See "Handling Timer Queuing Conflicts" for more information.

Default: 'drop'

'ErrorFcn'

String, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls StopFcn.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the timer object and an event structure to the callback function. The event structure contains the type of event in the Type field and the time of the event in the Data field.
- If your callback function accepts arguments in addition to the timer object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

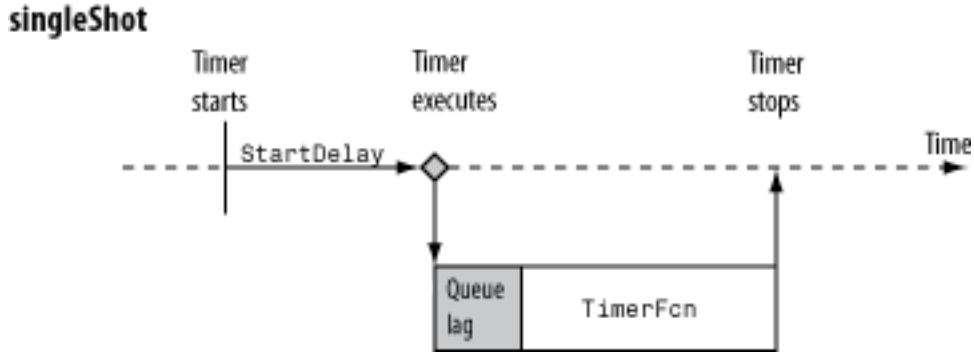
'ExecutionMode'

String that defines how the timer object schedules timer events. When Running='on', ExecutionMode is read only. This table summarizes the execution modes.

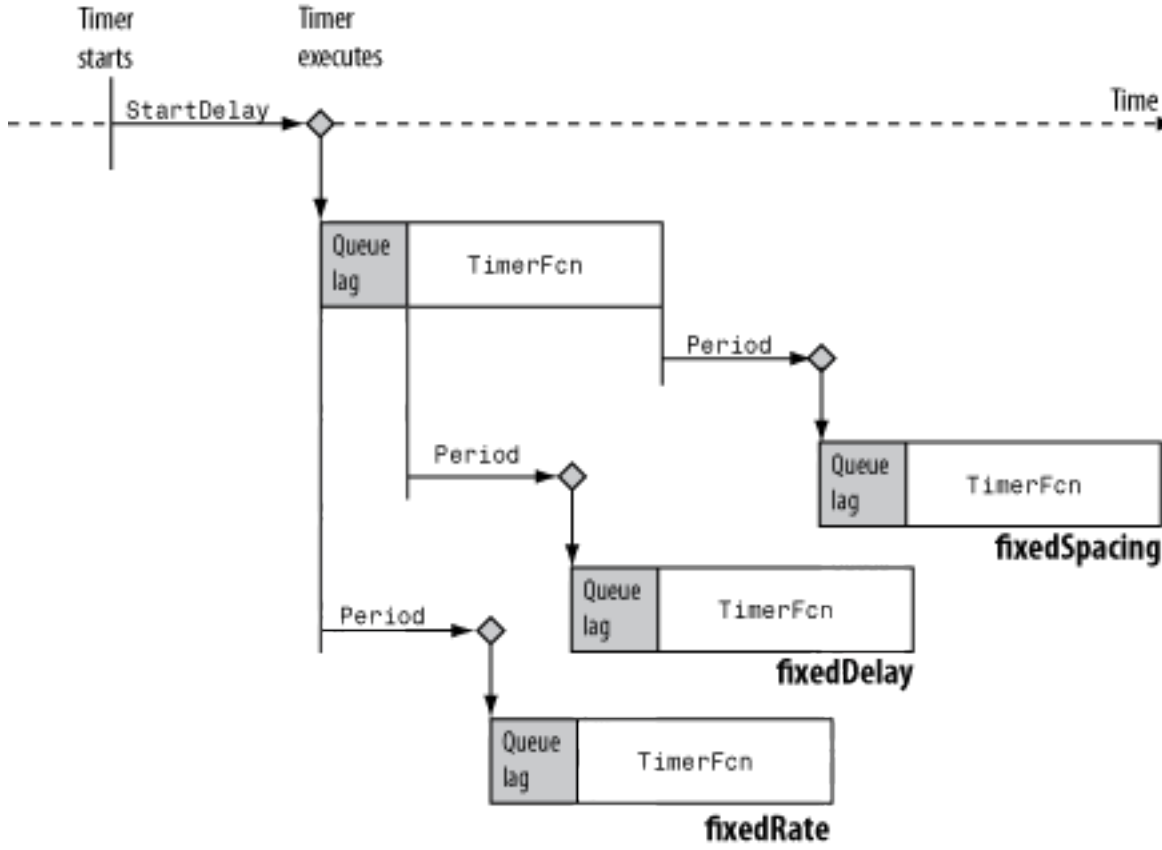
Execution Mode	Time Period Start Point
'singleShot'	In this mode, the timer callback function is only executed once. Therefore, the Period property has no effect. This is the default execution mode.
'fixedRate'	Starts immediately after the timer callback function is added to the MATLAB execution queue

Execution Mode	Time `Period` Start Point
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`'Name'`

String representing the `timer` name.

Default: `'timer-`*`i`*`'`, where *`i`* is a number indicating the *`i`*th timer object created this session. To reset *`i`* to 1, execute the `clear classes` command.

`'ObjectVisibility'`

String with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates. The `timerfind` function

does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

`'Period'`

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

`'StartDelay'`

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

`'StartFcn'`

String, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

`'StopFcn'`

String, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

`'Tag'`

String that represents a label for the object.

`'TasksToExecute'`

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

`'TimerFcn'`

String, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

`'UserData'`

Generic field for data that you want to add to the object.

Output Arguments

`propStruct`

Configurable properties of `t`, returned as a structure. The field names of `propStruct` are the property names of `t`, and the associated values of `propStruct` are cell arrays of the possible property values. If the property in `t` does not have a finite set of possible values, the property value in `propStruct` is an empty cell array.

`propCell`

Possible values of a given property name, returned as cell array of strings. If the property does not have a finite set of possible string values, `set` returns an empty cell array.

Examples

Display Configurable Timer Object Properties

Instantiate a timer object and call the `set` method.

```
t = timer;
set(t)
```

```
    BusyMode: [ {drop} | queue | error ]
```

```
    ErrorFcn: string -or- function handle -or- cell array
    ExecutionMode: [ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
    Name
    ObjectVisibility: [ {on} | off ]
    Period
    StartDelay
    StartFcn: string -or- function handle -or- cell array
    StopFcn: string -or- function handle -or- cell array
    Tag
    TasksToExecute
    TimerFcn: string -or- function handle -or- cell array
    UserData
```

Some of the timer properties, such as `Running`, are not displayed because they are read only.

Use the `set` method to output a structure.

```
out = set(t)
```

```
out =

            BusyMode: {3x1 cell}
            ErrorFcn: {}
       ExecutionMode: {4x1 cell}
                Name: {}
    ObjectVisibility: {2x1 cell}
              Period: {}
          StartDelay: {}
            StartFcn: {}
             StopFcn: {}
                 Tag: {}
      TasksToExecute: {}
             TimerFcn: {}
             UserData: {}
```

Delete the timer object from memory.

```
delete(t)
```

Display Possible Property Values

Instantiate a timer object and display possible values for the `BusyMode` property.

```
t = timer;
set(t,'BusyMode')
```

```
[ {drop} | queue | error ]
```

The output shows the three possible values for `BusyMode`. The default value, `drop`, is indicated by curly braces.

Display the possible values for `ErrorFcn`.

```
set(t,'ErrorFcn')
```

```
string -or- function handle -or- cell array
```

A description of the possible values is displayed since `ErrorFcn` does not have a set list of possible values.

Output the possible property values.

```
out1 = set(t,'BusyMode')
out2 = set(t,'ErrorFcn')
```

```
out1 =

    'drop'
    'queue'
    'error'


out2 =

     {}
```

While `set(t,'ErrorFcn')` displays a description of the possible values, `out2 = set(t,'ErrorFcn')` returns an empty cell.

Delete the timer from memory.

```
delete(t)
```

Set Timer Object Property

Instantiate a timer object and make the object invisible. Display the objects's visibility and delete the object.

```
t = timer;
set(t,'ObjectVisibility','off')
get(t,'ObjectVisibility')
delete(t)
```

```
ans =

off
```

Set Timer Object Properties Using Structure

Construct a structure to modify several timer object properties.

```
s.BusyMode = 'queue';
s.ExecutionMode = 'fixedDelay';
s.ObjectVisibility = 'off'
```

```
s =

            BusyMode: 'queue'
       ExecutionMode: 'fixedDelay'
    ObjectVisibility: 'off'
```

Create a timer, display the properties in `s`, modify the timer, display the new values of the properties, and delete the timer.

```
t = timer;
get(t,{'BusyMode','ExecutionMode','ObjectVisibility'})
set(t,s)
get(t,{'BusyMode','ExecutionMode','ObjectVisibility'})
delete(t)
```

```
ans =

    'drop'    'singleShot'    'on'


ans =

    'queue'    'fixedDelay'    'off'
```

Set Timer Objects Using Cell Arrays

Create a cell array of properties to modify, and a cell array of the values of the corresponding properties. Instantiate a timer, and display the initial values of the properties in the property name cell array, `nameArr`.

```
nameArr = {'BusyMode','ExecutionMode','Period'};
valArr = {'queue','fixedDelay',3};
t = timer;
get(t,nameArr)
```

```
ans =

    'drop'    'singleShot'    [1]
```

Modify the timer object and display the new property values. Delete the timer.

```
set(t,nameArr,valArr)
get(t,nameArr)
delete(t)
```

```
ans =

    'queue'    'fixedDelay'    [3]
```

Instantiate an array of three timers. Create a new property name cell array to modify the `BusyMode`, `ExecutionMode`, and `UserData` properties. Display the initial values of the properties for each timer.

```
tArr = [timer timer timer];
nameArr = {'BusyMode','ExecutionMode','UserData'};
get(tArr,nameArr)
```

```
ans =

    'drop'    'singleShot'    []
    'drop'    'singleShot'    []
    'drop'    'singleShot'    []
```

Assign each property a different value in each timer. Create a cell array containing the new values. Each row indicates the values for the properties in the corresponding timer.

```
valArr = {'queue','fixedDelay',3;...
            'error','fixedSpacing',42;...
            'drop','fixedRate','hello'};
```

Modify the timer object properties and display the updated values.

```
set(tArr,nameArr,valArr)
get(tArr,nameArr)
```

```
ans =

    'queue'    'fixedDelay'      [    3]
    'error'    'fixedSpacing'    [   42]
    'drop'     'fixedRate'       'hello'
```

Delete the timers from memory.

```
delete(tArr)
```

Alternatives

You can also use dot notation to set timer object properties. For example, `t.ObjectVisibility = 'off'` sets the property to the same value as `set(t,'ObjectVisibility','off')`.

See Also

`timer` | `get`

Introduced before R2006a

start

Class: timer

Start timer object

Syntax

```
start(t)
```

Description

`start(t)` starts the timer object, `t`. If `t` is an array of timer objects, `start` starts all the timers.

The `start` method sets the `Running` property of the timer object to `'on'`, executes the `StartFcn` callback, and initiates `TimerFcn` callback .

Input Arguments

`t`

Object of class `timer`.

Examples

Start Timer

Create and start a timer that displays the message 'timer started.' as the `StartFcn` callback and generates a random number as the `TimerFcn` callback. Delete the timer.

```
t = timer('StartFcn',@(~,~)disp('timer started.'),'TimerFcn',@(~,~)disp(rand(1)));
start(t)
delete(t)
```

```
timer started.
```

```
    0.9706
```

Your output from `rand` will vary.

Start Several Timers

Create and start three timers that displays a message for the `StartFcn` callbacks and compute the sine, cosine, and tangent of `pi/4` as the `TimerFcn` callbacks. Delete the timers.

```
t1 = timer('StartFcn',@(~,~)disp('t1 started.'),'TimerFcn',@(~,~)sin(pi/4));
t2 = timer('StartFcn',@(~,~)disp('t2 started.'),'TimerFcn',@(~,~)cos(pi/4));
t3 = timer('StartFcn',@(~,~)disp('t3 started.'),'TimerFcn',@(~,~)tan(pi/4));
start([t1 t2 t3]);
delete([t1 t2 t3]);
```

```
t1 started.
t2 started.
t3 started.
```

See Also

`timer` | `delete` | `startat` | `stop`

Introduced before R2006a

startat

Class: timer

Schedule timer to fire at specified time

Syntax

```
startat(t,firingTime)
startat(t,Y,M,D)
startat(t,Y,M,D,H,MI,S)
```

Description

`startat(t,firingTime)` schedules timer, `t`, to fire at specified time, `firingTime`. A timer fires by executing the callback function, `timerFcn`. `firingTime` must be within 25 days of the current time.

- If `t` is an array of timer objects and `firingTime` is a scalar, `startat` sets all the timers to fire at the specified time.
- If `t` is an array of timer objects and `firingTime` is an array of the same size as `t`, `startat` sets each timer to fire at the corresponding time.

`startat(t,Y,M,D)` starts the timer and schedules execution of `TimerFcn` at the year (`Y`), month (`M`), and day (`D`) specified.

`startat(t,Y,M,D,H,MI,S)` also specifies the hour (`H`), minute (`MI`), and second (`S`) specified.

Algorithms

- The `startat` method specifies when the timer object executes the `TimerFcn` callback, not when the timer starts running. The timer starts running with the call to the `startat` method.
- Based on the specified time, `startat` computes and sets the required `StartDelay` property of the timer object, `t`. Additionally, it sets the `Running` property of the timer object to `'on'`, and executes the `StartFcn` callback.

- `startat` modifies the `timer` object's `startDelay` property. As such, `startat` overrides specified values of the timer's `startDelay` property.

Input Arguments

`t`

Object of class `timer`.

`firingTime`

Time at which the timer object is to fire, specified as a serial date number, date string, or a date vector. `firingTime` can be a single date or an array of dates with the same number of rows as timer objects in `t`.

- A serial date number indicates the number of days that have elapsed since 1-Jan-0000 (starting at 1). See `datenum` for additional information about serial date numbers.
- To specify date strings, use the following date formats defined by the `datestr` function: 0, 1, 2, 6, 13, 14, 15, 16, or 23. These numeric identifiers correspond to formats defined by the `formatOut` property of the `datestr` function. Date strings with two-character years are interpreted to be within the 100 years centered on the current year.
- Date vectors are specified as an `m-by-6` or `m-by-3` matrix containing `m` full or partial date vectors, respectively. A full date vector has six elements indicating year, month, day, hour, minute, and second, in that order. A partial date vector has three elements indicating year, month, and day, in that order.

`Y,M,D`

Time at which the timer object is to fire, specified as numbers indicating the year (`Y`), month (`M`), and day (`D`). Month values less than 1 are set to 1; other arguments can wrap and have negative values.

`Y,M,D,H,MI,S`

Time at which the timer object is to fire, specified as numbers indicating the year (`Y`), month (`M`), day (`D`), hour (`H`), minute (`MI`), and second (`S`) specified. Month values less than 1 are set to 1; other arguments can wrap and have negative values.

Examples

Start Timer in 2 Seconds

Create a timer that displays messages at start time and firing time.

```
t = timer('TimerFcn', @(~,~)disp('Fired.'), ...
    'StartFcn', @(~,~)disp('Started.'));
```

Set the timer to fire 2 seconds from the present time using a serial date. A serial date is specified in days.

```
two = 2/(60^2*24); % two seconds in serial time
fTime = now + two
startat(t,fTime);
```

```
fTime =

   7.3527e+05

Started.
Fired.
```

Wait for the timer to fire, and then delete the timer.

```
delete(t)
```

Start Timer Using Year, Month, Day

Create a timer that displays messages at start time and firing time.

```
t = timer('TimerFcn', @(~,~)disp('Fired.'), ...
    'StartFcn', @(~,~)disp('Started.'));
```

Schedule the timer to start 2 days from present at 00:00:00

```
[Y, M, D, H, MI, S] = datevec(now+2);
startat(t,Y,M,D)
```

```
Started.
```

Manually stop and delete the timer.

```
stop(t)
```

```
delete(t)
```

See Also

`timer` | `delete` | `start` | `stop`

More About

- “Carryover in Date Vectors and Strings”

Introduced before R2006a

stop

Class: timer

Stop timer object

Syntax

```
stop(t)
```

Description

`stop(t)` stops the timer object, `t`. If `t` is an array of timer objects, the `stop` method stops each timer.

The `stop` method sets the `Running` property of the timer object to `'off'` and executes the `StopFcn` callback .

Tips

- Use the `stop` method to stop a timer manually. The timer automatically stops when the `TimerFcn` callback executes the number of times specified by the `ExecutionMode` and `TasksToExecute` properties or when an error occurs while executing a `TimerFcn` callback.

Input Arguments

`t`

Object of class `timer`.

Examples

Stop Timer

Create a timer object that generates 100 random numbers and executes one million times. Define a `StopFcn` callback that displays the message 'Timer has stopped.' Start the timer and verify the timer is running

```
t = timer('TimerFcn','rand(100,1);',...
    'ExecutionMode','fixedSpacing','TasksToExecute',1e6,...
    'StopFcn','disp(''Timer has stopped.'')');
start(t)
t.Running
```

```
ans =

on
```

Manually stop the timer and verify it is no longer running. Delete the timer.

```
stop(t)
t.Running
delete(t)
```

```
Timer has stopped.

ans =

off
```

See Also

`timer` | `delete` | `start` | `startat`

Introduced before R2006a

timerfind

Class: timer

Find timer object

Syntax

```
out = timerfind
out = timerfind(Name,Value)
out = timerfind(t,Name,Value)
out = timerfind(S)
```

Description

`out = timerfind` finds the "visible timer objects" on page 1-8277and returns an array, `out`.

`out = timerfind(Name,Value)` finds "visible timer objects" on page 1-8277 with property values matching those passed as Name,Value pair arguments and returns an array, `out`.

`out = timerfind(t,Name,Value)` matches Name,Value pair arguments to the timer objects listed in `t`, where `t` can be an array of timer objects, and returns an array, `out`.

`out = timerfind(S)` matches property values defined in the structure, `S`, and returns an array, `out`. The field names of `S` are timer object property names and the field values are the corresponding property values.

Tips

- `timerfind` only finds "visible timer objects" on page 1-8277. Visible timer objects are those that are in memory and have the `ObjectVisibility` property set to `'on'`. To find objects that are hidden, but still valid, use `timerfindall`.

Input Arguments

`t`

Array of objects of class `timer`

`s`

Structure with field names corresponding to `timer` object property names. Field values are the corresponding property values.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'BusyMode'`

String that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

`BusyMode` Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue
`'queue'`	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts `Period` property to manage tasks in execution queue

See “Handling Timer Queuing Conflicts” for more information.

Default: `'drop'`

`'ErrorFcn'`

String, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls `StopFcn`.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

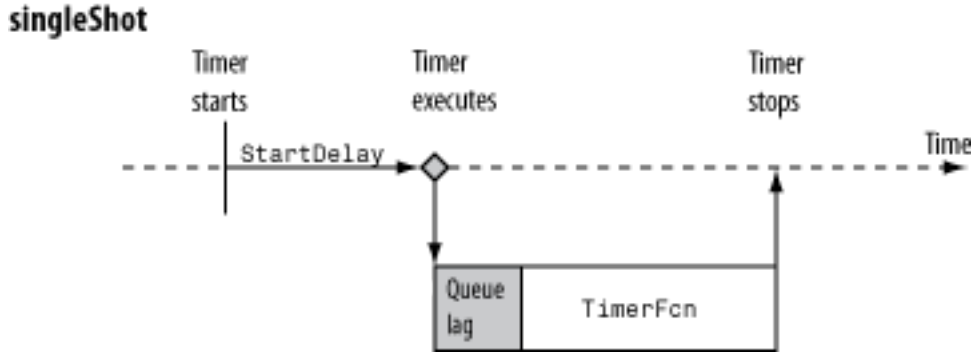
For more information, see “Timer Callback Functions”.

`'ExecutionMode'`

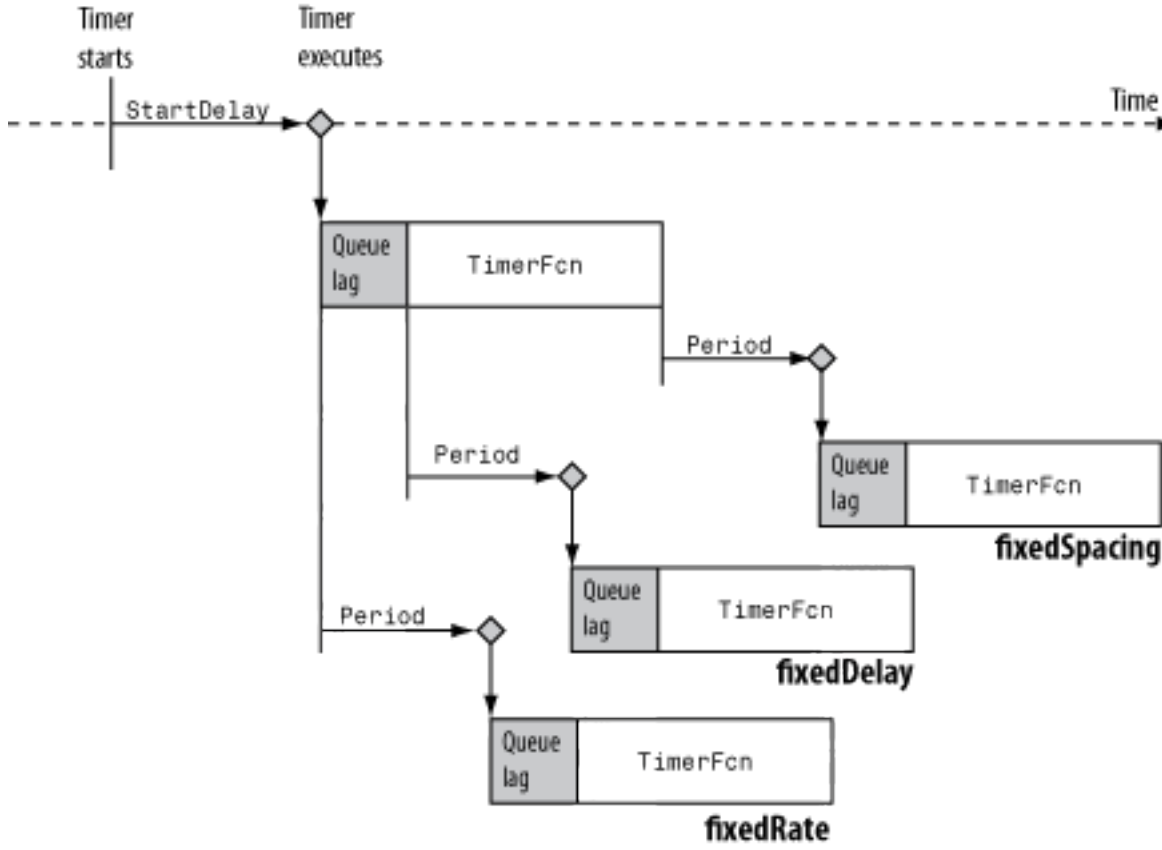
String that defines how the `timer` object schedules timer events. When `Running='on'`, `ExecutionMode` is read only. This table summarizes the execution modes.

Execution Mode	Time `Period` Start Point
`'singleShot'`	In this mode, the timer callback function is only executed once. Therefore, the `Period` property has no effect. This is the default execution mode.
`'fixedRate'`	Starts immediately after the timer callback function is added to the MATLAB execution queue
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`'Name'`

String representing the `timer` name.

Default: `'timer-`*`i`*`'`, where *`i`* is a number indicating the *`i`*th timer object created this session. To reset *`i`* to 1, execute the `clear classes` command.

`'ObjectVisibility'`

String with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates. The `timerfind` function

does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

`'Period'`

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

`'StartDelay'`

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

`'StartFcn'`

String, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

`'StopFcn'`

String, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

`'Tag'`

String that represents a label for the object.

`'TasksToExecute'`

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

`'TimerFcn'`

String, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

`'UserData'`

Generic field for data that you want to add to the object.

Read Only Name-Value Pair Arguments

`AveragePeriod`

Average time in seconds between `TimerFcn` executions since the timer started. Value is `NaN` until timer executes two timer callbacks.

`InstantPeriod`

The time in seconds between the last two executions of `TimerFcn`. Value is `NaN` until timer executes two timer callbacks.

`Running`

String defined as `'off'` or `'on'`, indicating whether the timer is currently executing callback functions.

`TasksExecuted`

The number of times the timer called `TimerFcn` since the timer started.

`Type`

String that identifies the object type.

Definitions

visible timer objects

Visible timer objects are timer objects that are in memory and have the `ObjectVisibility` property set to 'on'.

Examples

Find Timer Objects Existing in Memory

Create several individual timers and an array of timers.

```
t1 = timer('Tag','broadcastProgress','UserData','Monday');
t2 = timer('Tag','displayProgress','UserData','Monday');
timerArr = [timer('Tag','broadcastProgress','UserData','Tuesday');
    timer('Tag','displayProgress','UserData','Tuesday');
    timer('Tag','displayProgress','UserData','Wednesday');];
```

Find all the timers in memory.

```
out1 = timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                    timer-1
   2       singleShot      1        ''                    timer-2
   3       singleShot      1        ''                    timer-3
   4       singleShot      1        ''                    timer-4
   5       singleShot      1        ''                    timer-5
```

Find only those timers in memory that have the string value, `'displayProgress'`, as the `Tag` property.

```
out2 = timerfind('Tag','displayProgress')
```

```
Timer Object Array
```

```
   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                    timer-2
   2       singleShot      1        ''                    timer-4
   3       singleShot      1        ''                    timer-5
```

Limit the search for timers with the string value, `'displayProgress'`, as the `Tag` property to timer objects in `timerArr`.

```
out3 = timerfind(timerArr,'Tag','displayProgress')
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                    timer-4
   2       singleShot      1        ''                    timer-5
```

Define a `struct` containing the `Tag` and `UserData` properties of interest.

```
searchStruct = struct('Tag','broadcastProgress','UserData','Monday')
```

```
searchStruct =

         Tag: 'broadcastProgress'
    UserData: 'Monday'
```

Use the `struct` as the search criteria to find timer objects in memory.

```
out4 = timerfind(searchStruct)
```

```
   Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
          BusyMode: drop
           Running: off

   Callbacks
```

```
          TimerFcn: ''
          ErrorFcn: ''
          StartFcn: ''
           StopFcn: ''
```

Delete the timer objects.

```
delete(t1)
delete(t2)
delete(timerArr)
```

Delete Timer by Name

Simulate having existing timers in memory by creating an array of timers. Create a new timer with a custom name. List all visible timers.

```
existingTimers = [timer timer timer];

myTimerName = 'myTimer';
anotherTimer = timer('Name',myTimerName);

timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                     timer-1
   2       singleShot      1        ''                     timer-2
   3       singleShot      1        ''                     timer-3
   4       singleShot      1        ''                     myTimer
```

Delete the specified timer and list all visible timers.

```
delete(timerfind('Name',myTimerName));
timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                     timer-1
   2       singleShot      1        ''                     timer-2
   3       singleShot      1        ''                     timer-3
```

Delete all visible timers from memory.

```
delete(timerfind)
```

Find Valid Timer Objects Cleared from Workspace

Use `timerfind` to find 'lost' timer object references. References are lost when you clear the timer object from the workspace, but do not delete it from memory.

Create two timer objects. Since the callback function does not require the timer or event object, you can use the tilde (~) operator to ignore the inputs in the function handle.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
whos
```

```
  Name      Size            Bytes  Class    Attributes

  t1        1x1               104  timer
  t2        1x1               104  timer
```

Clear one of the timer objects from the workspace. To actually remove the timer from memory, you need to both clear it and delete it.

```
clear t1
whos
```

```
  Name      Size            Bytes  Class    Attributes

  t2        1x1               104  timer
```

Try to delete the timer, `t1`.

```
delete(t1)
```

```
Undefined function or variable 't1'.
```

The timer, `t1`, can not be removed from memory using delete because its reference has been cleared.

Find valid timer objects in memory.

```
out = timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:               Name:
   1       singleShot      1        1x1 function_handle arraytimer-1
```

```
   2       singleShot      1        1x1 function_handle arraytimer-2
```

Since two timers were found, determine which timer does not exist in the workspace.

```
out ~= t2
```

```
ans =

     1     0
```

The first timer object in `out` is not equal to `t2`. This was previously `t1`. It is reassigned to `t1`. Since it is still valid, the timer can be started.

```
t1 = out(1);
start(t1)
```

```
Timer 1 Fired!
```

Delete timer objects. `timerfind` provides a way of accessing timer objects in memory. It does not copy the objects; therefore you do not need to delete `out` from memory. To verify, use `timerfind`.

```
delete(t1)
delete(t2)
timerfind
```

```
ans =

     []
```

Delete All Timer Objects in Memory

Create four timer objects.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
t3 = timer('TimerFcn',@(~,~)disp('Timer 3 Fired!'));
t4 = timer('TimerFcn',@(~,~)disp('Timer 4 Fired!'));
```

Clear two timers from the workspace.

```
clear t2 t3
```

Pass `timerfind` to `delete` to remove all timer objects from memory, whether or not they exist in the workspace.

```
delete(timerfind)
timerfind

ans =

     []
```

See Also

`timer` | `delete` | `timerfindall`

Introduced before R2006a

timerfindall

Class: timer

Find timer object, regardless of visibility

Syntax

```
out = timerfindall
out = timerfindall(Name,Value)
out = timerfindall(t,Name,Value)
out = timerfindall(S)
```

Description

`out = timerfindall` finds timer objects existing in memory, regardless of visibility and returns an array, `out`. Use the `ObjectVisibility` property to set the object's visibility.

`out = timerfindall(Name,Value)` finds timer objects existing in memory, regardless of visibility whose property values match those passed as Name,Value pair arguments and returns an array, `out`.

`out = timerfindall(t,Name,Value)` matches Name,Value pair arguments to the timer objects listed in `t`, where `t` can be an array of timer objects, and returns an array, `out`.

`out = timerfindall(S)` matches property values defined in the structure, `S` and returns an array, `out`. The field names of `S` are timer object property names and the field values are the corresponding property values.

Tips

- `timerfindall` finds timer objects in memory, regardless of the value of the `ObjectVisibility` property. To limit the search to objects with `ObjectVisibility` set to `'on'`, use `timerfind`. .

Input Arguments

t

Array of objects of class `timer`.

s

Structure with field names corresponding to `timer` object property names. Field values are the corresponding property values.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'BusyMode'

String that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue
`'queue'`	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts `Period` property to manage tasks in execution queue

See “Handling Timer Queuing Conflicts” for more information.

Default: `'drop'`

`'ErrorFcn'`

String, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls `StopFcn`.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

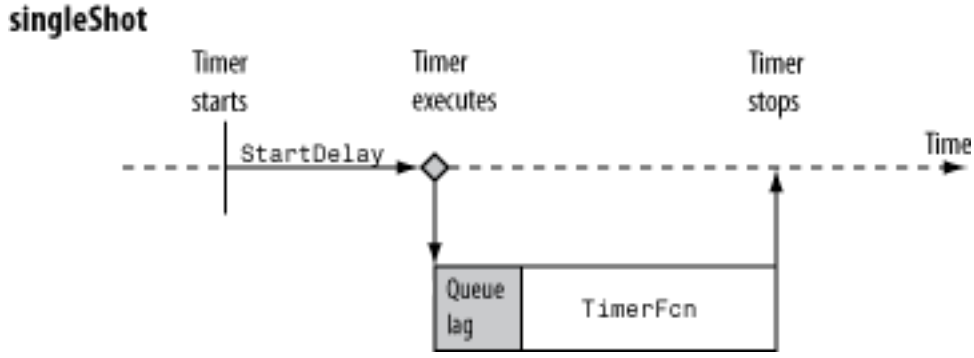
For more information, see “Timer Callback Functions”.

`'ExecutionMode'`

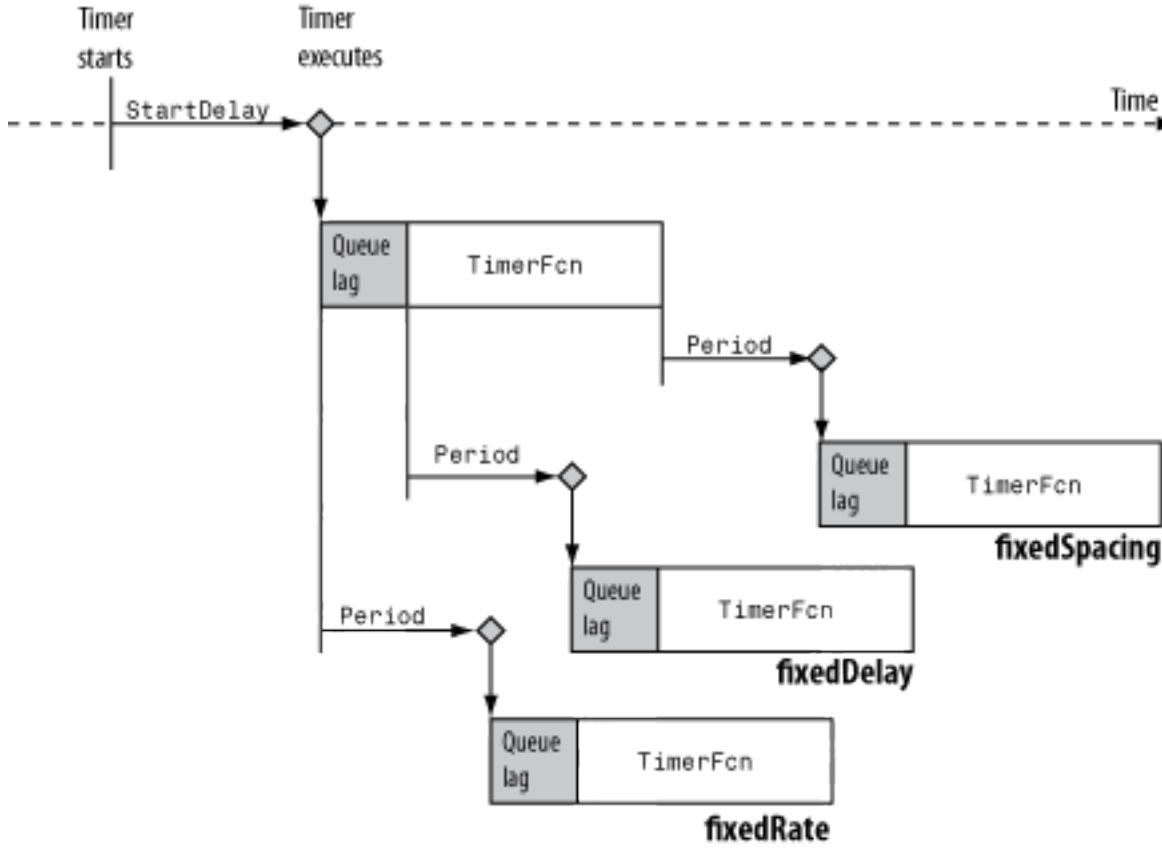
String that defines how the `timer` object schedules timer events. When `Running='on'`, `ExecutionMode` is read only. This table summarizes the execution modes.

Execution Mode	Time `Period` Start Point
`'singleShot'`	In this mode, the timer callback function is only executed once. Therefore, the `Period` property has no effect. This is the default execution mode.
`'fixedRate'`	Starts immediately after the timer callback function is added to the MATLAB execution queue
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`'Name'`

String representing the `timer` name.

Default: `'timer-`*`i`*`'`, where *`i`* is a number indicating the *`i`*th timer object created this session. To reset *`i`* to 1, execute the `clear classes` command.

`'ObjectVisibility'`

String with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates. The `timerfind` function

does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

`'Period'`

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

`'StartDelay'`

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

`'StartFcn'`

String, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

`'StopFcn'`

String, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

`'Tag'`

String that represents a label for the object.

`'TasksToExecute'`

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

`'TimerFcn'`

String, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a string, when MATLAB executes the callback it evaluates the MATLAB code contained in the string.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see "Timer Callback Functions".

`'UserData'`

Generic field for data that you want to add to the object.

Read Only Name-Value Pair Arguments

`AveragePeriod`

Average time in seconds between `TimerFcn` executions since the timer started. Value is `NaN` until timer executes two timer callbacks.

`InstantPeriod`

The time in seconds between the last two executions of `TimerFcn`. Value is `NaN` until timer executes two timer callbacks.

`Running`

String defined as `'off'` or `'on'`, indicating whether the timer is currently executing callback functions.

`TasksExecuted`

The number of times the timer called `TimerFcn` since the timer started.

`Type`

String that identifies the object type.

Examples

Find and Delete All Timers From Memory

Create four timer objects.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
t3 = timer('TimerFcn',@(~,~)disp('Timer 3 Fired!'));
t4 = timer('TimerFcn',@(~,~)disp('Timer 4 Fired!'));
```

Set timers `t2` and `t4` to be invisible.

```
t2.ObjectVisibility = 'off';
t4.ObjectVisibility = 'off';
```

Clear timers `t1` and `t2` from the workspace.

```
clear t1 t2
whos
```

```
  Name      Size            Bytes  Class    Attributes

  t3        1x1               104  timer
  t4        1x1               104  timer
```

Find all visible timers in memory.

```
timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        1x1 function_handle arraytimer-1
   2       singleShot      1        1x1 function_handle arraytimer-3
```

`timerfind` finds only timers `t1` and `t2`, since they are visible. Timer `t2` is still valid and in memory even though it was cleared from the workspace

Find all timers in memory.

```
timerfindall
```

```
Timer Object Array
```

```
   Index:  ExecutionMode:  Period:  TimerFcn:               Name:
   1       singleShot      1        1x1 function_handle arraytimer-1
   2       singleShot      1        1x1 function_handle arraytimer-2
   3       singleShot      1        1x1 function_handle arraytimer-3
   4       singleShot      1        1x1 function_handle arraytimer-4
```

`timerfindall` finds all four valid timers in memory even though `t2` and `t4` are invisible and `t1` and `t2` were cleared from the workspace.

Delete all timers from memory.

```
delete(timerfindall)
```

Find Timer Objects Existing in Memory

Create several individual timers and an array of timers.

```
t1 = timer('Tag', 'broadcastProgress','UserData','Monday');
t2 = timer('Tag', 'displayProgress','UserData','Monday');
timerArr = [timer('Tag', 'broadcastProgress','UserData','Tuesday');
    timer('Tag', 'displayProgress','UserData','Tuesday');
    timer('Tag', 'displayProgress','UserData','Wednesday');];
```

Make timer `t1` and `timerArr(2)` invisible.

```
t1.ObjectVisibility = 'off';
timerArr(2).ObjectVisibility = 'off';
```

Find all the timers in memory using `timerfind`.

```
out1 = timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:               Name:
   1       singleShot      1        ''                     timer-2
   2       singleShot      1        ''                     timer-3
   3       singleShot      1        ''                     timer-5
```

`timerfind` does not find the hidden timers.

Find all the timers in memory using `timerfindall`.

```
out2 = timerfindall
```

```
Timer Object Array
```

```
   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                    timer-1
   2       singleShot      1        ''                    timer-2
   3       singleShot      1        ''                    timer-3
   4       singleShot      1        ''                    timer-4
   5       singleShot      1        ''                    timer-5
```

`timerfindall` finds all timers, even the invisible ones.

Find only those timers in memory that have the string value, `'displayProgress'`, as the `Tag` property.

```
out3 = timerfindall('Tag','displayProgress')
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                    timer-2
   2       singleShot      1        ''                    timer-4
   3       singleShot      1        ''                    timer-5
```

Limit the search for timers that have the string value, `'displayProgress'`, as the `Tag` property to timer objects in `timerArr`.

```
out4 = timerfindall(timerArr,'Tag','displayProgress')
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:              Name:
   1       singleShot      1        ''                    timer-4
   2       singleShot      1        ''                    timer-5
```

Define a `struct` containing the `Tag` and `UserData` properties of interest.

```
searchStruct = struct('Tag','broadcastProgress','UserData','Monday')
```

```
searchStruct =

         Tag: 'broadcastProgress'
    UserData: 'Monday'
```

Use the `struct` as the search criteria to find timer objects in memory.

```
out5 = timerfindall(searchStruct)
```

```
   Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: ''
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''
```

Delete the timer objects.

```
delete(timerfindall)
```

Find Invisible Timers

Create four timer objects.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
t3 = timer('TimerFcn',@(~,~)disp('Timer 3 Fired!'));
t4 = timer('TimerFcn',@(~,~)disp('Timer 4 Fired!'));
```

Set timers `t2` and `t4` to be invisible, and clear timers `t1` and `t2` from the workspace.

```
t2.ObjectVisibility = 'off';
t4.ObjectVisibility = 'off';
clear t1 t2;
whos
```

```
  Name      Size            Bytes  Class    Attributes

  t3        1x1               104  timer
  t4        1x1               104  timer
```

Find all valid invisible timers.

```
out = timerfindall('ObjectVisibility','off')
```

```
Timer Object Array
```

```
Index:  ExecutionMode:  Period:  TimerFcn:               Name:
1       singleShot      1        1x1 function_handle arraytimer-2
2       singleShot      1        1x1 function_handle arraytimer-4
```

Both valid invisible timers were found by `timerfindall`, regardless of whether they were in the workspace.

See Also

`timer` | `delete` | `timerfind`

Introduced before R2006a

wait

Class: timer

Block command prompt until timer stops running

Syntax

```
wait(t)
```

Description

`wait(t)` blocks the command prompt until timer, `t`, stops running. If `t` is an array of timer objects, `wait` blocks the MATLAB command line until each timer in `t` has stopped running.

To block the command line, the timer object must first start via `start` or `startat` before calling the `wait` method. If the timer is not running, `wait` returns immediately.

Input Arguments

`t`

Array of `timer` objects

Default:

Examples

Block Command Prompt While Timer Runs

Create a timer that waits 10 seconds, and then displays a message. Start the timer and wait for it to finish.

```
T = timer('TimerFcn',@(~,~)disp('Fired.'),'StartDelay',10);
start(T)
```

```
Fired.
```

Notice that after the timer starts, the MATLAB prompt returns.

Start the timer and use the wait method to block anyone from entering commands at the MATLAB command line. You must start the timer before calling the wait command.

```
start(T)
wait(T)
```

```
Fired.
```

Notice that after the timer starts, the MATLAB prompt disappears until the timer stops.

Delete the timer.

```
delete(T)
```

See Also

`timer` | `start`

Introduced before R2006a

times, .*

Element-wise multiplication

Syntax

```
C = A.*B
C = times(A,B)
```

Description

`C = A.*B` multiplies arrays `A` and `B` element by element and returns the result in `C`.

`C = times(A,B)` is an alternate way to execute `A.*B`, but is rarely used. It enables operator overloading for classes.

Examples

Multiply Two Vectors

Create two vectors, `A` and `B`, and multiply them element by element.

```
A = [1 0 3];
B = [2 3 7];
C = A.*B
```

```
C =

     2     0    21
```

Multiply Two Arrays

Create two 3-by-3 arrays, `A` and `B`, and multiply them element by element.

```
A = [1 0 3; 5 3 8; 2 4 6];
B = [2 3 7; 9 1 5; 8 8 3];
C = A.*B
```

```
C =
```

```
     2     0    21
    45     3    40
    16    32    18
```

- “Combine Categorical Arrays Using Multiplication”

Input Arguments

A — Left array

scalar | vector | matrix | multidimensional array

Left array, specified as a scalar, vector, matrix, or multidimensional array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `categorical` | `duration` | `calendarDuration`
Complex Number Support: Yes

B — Right array

scalar | vector | matrix | multidimensional array

Right array, specified as a scalar, vector, matrix, or multidimensional array. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `categorical` | `duration` | `calendarDuration`
Complex Number Support: Yes

More About

- “Array vs. Matrix Operations”
- “Operator Precedence”

See Also

`mtimes`

Introduced before R2006a

title

Add title to current axes

Syntax

```
title(str)
title(str,Name,Value)

title(ax, ___ )

h = title( ___ )
```

Description

`title(str)` adds the title consisting of a string, `str`, at the top and in the center of the current axes. Reissuing the `title` command causes the new title to replace the old title.

`title(str,Name,Value)` additionally specifies the title properties using one or more `Name,Value` pair arguments.

`title(ax, ___ )` adds the title to the axes specified by `ax`. This syntax allows you to specify the axes to which to add a title. `ax` can precede any of the input argument combinations in the previous syntaxes.

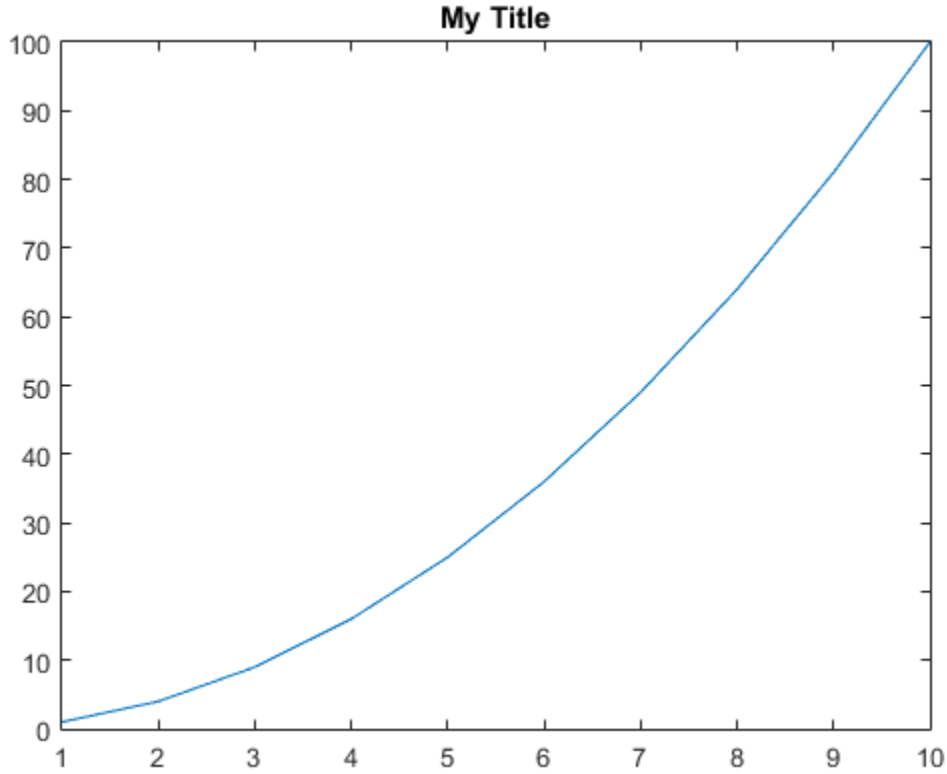
`h = title( ___ )` returns the handle to the text object used as the title. The handle is useful when making future modifications to the title.

Examples

Add Title to Current Figure

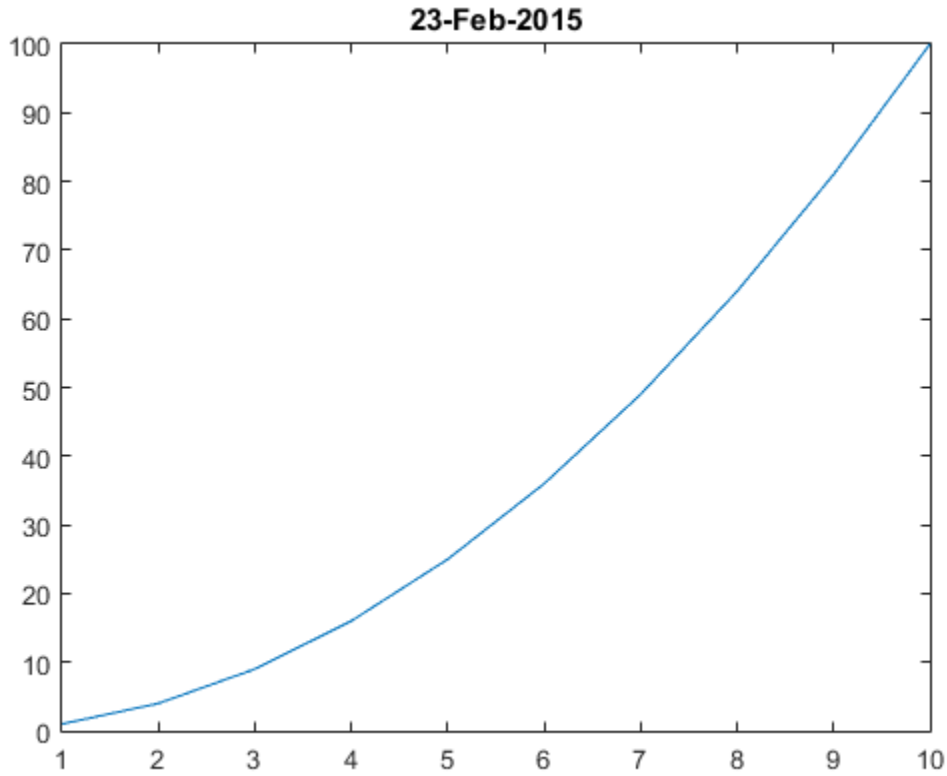
Create a figure and display a title in the current axes.

```
figure
plot((1:10).^2)
title('My Title')
```



You also can call `title` with a function that returns a string. For example, the `date` function returns a string containing today's date.

```
title(date)
```

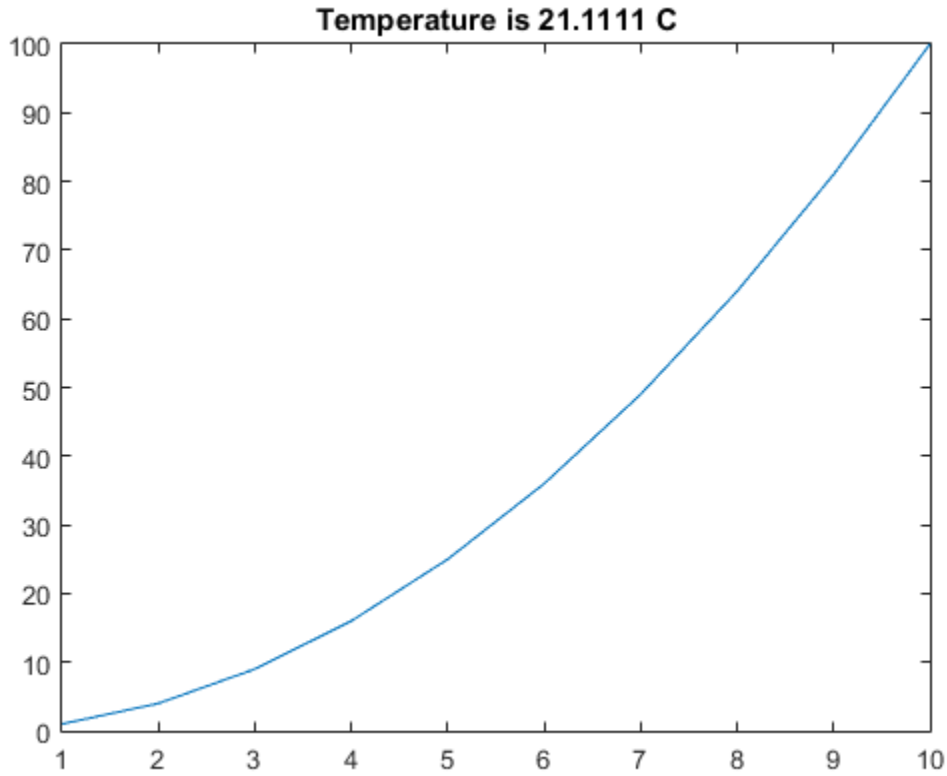


MATLAB® sets the output of `date` as the axes title.

Include Variable's Value in Title

Include the value of variable `c` in a title.

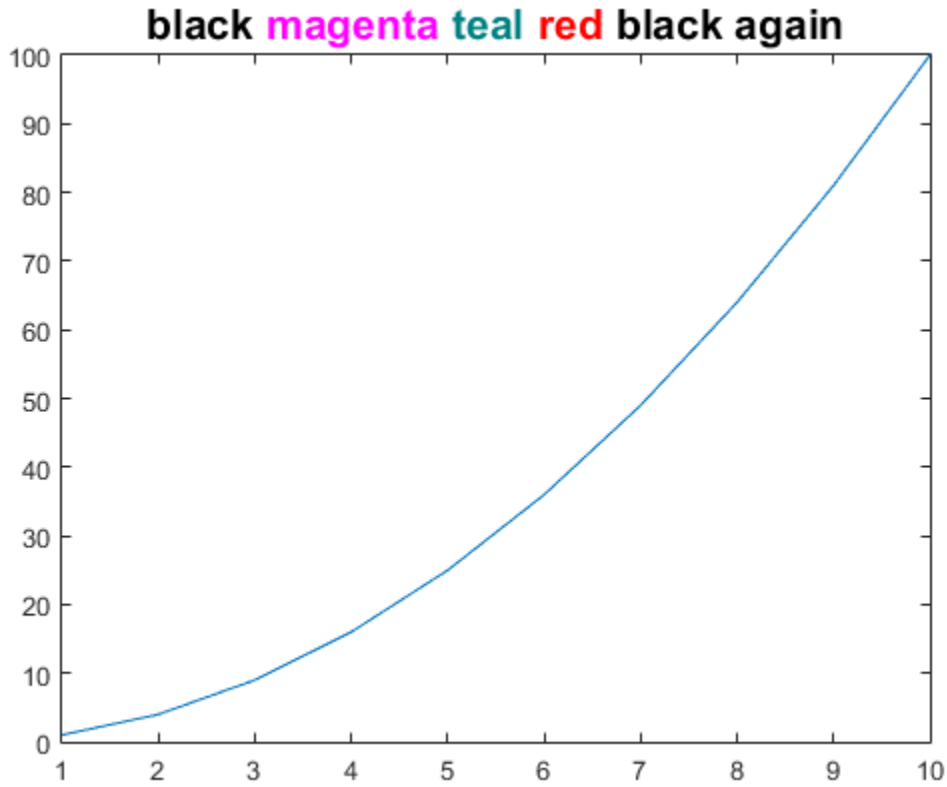
```
figure
plot((1:10).^2)
f = 70;
c = (f-32)/1.8;
title(['Temperature is ',num2str(c),' C'])
```



Create Multicolored Title Using TeX Markup

In a TeX string, use the color modifier `\color` to change the color of characters following it from the previous color.

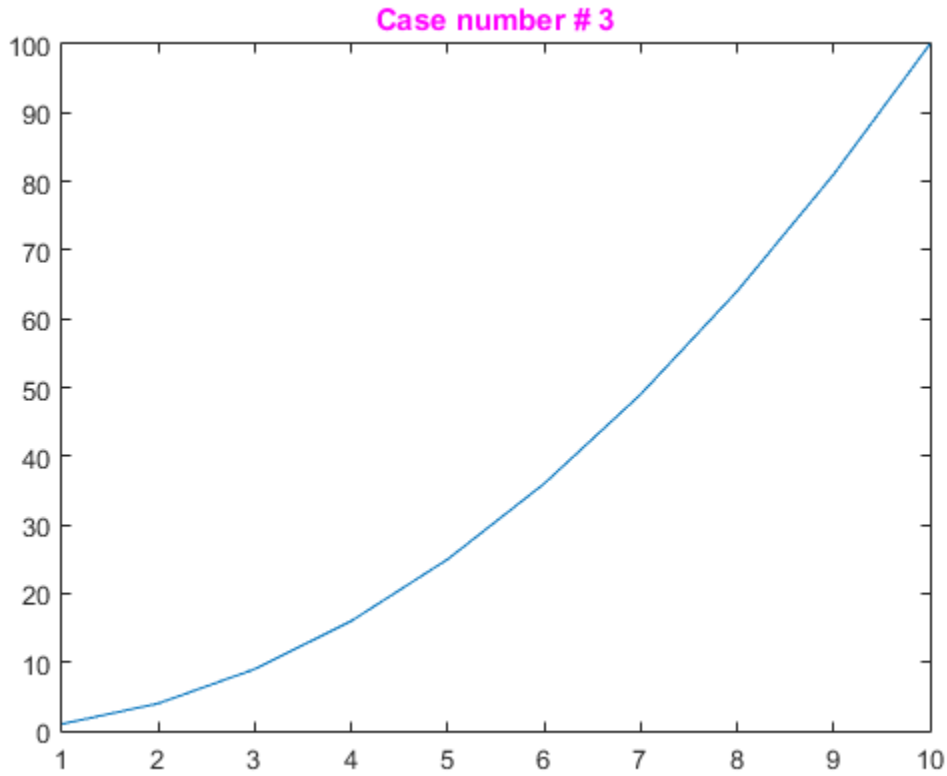
```
figure
plot((1:10).^2)
title(['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```



Create Colored Title Using Name,Value Pair Argument

Use the `Name,Value` pair `'Color','m'` to set the color of the title to magenta.

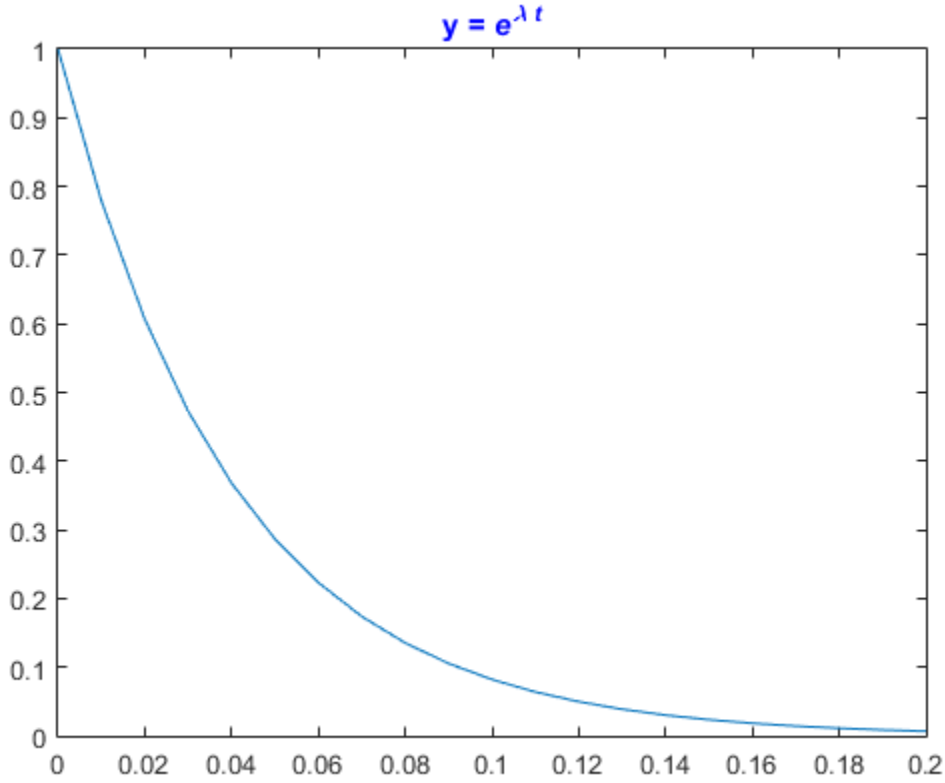
```
figure
plot((1:10).^2)
title('Case number # 3','Color', 'm')
```



Include Greek Symbols in Title

Use a TeX string to include Greek symbols in a title.

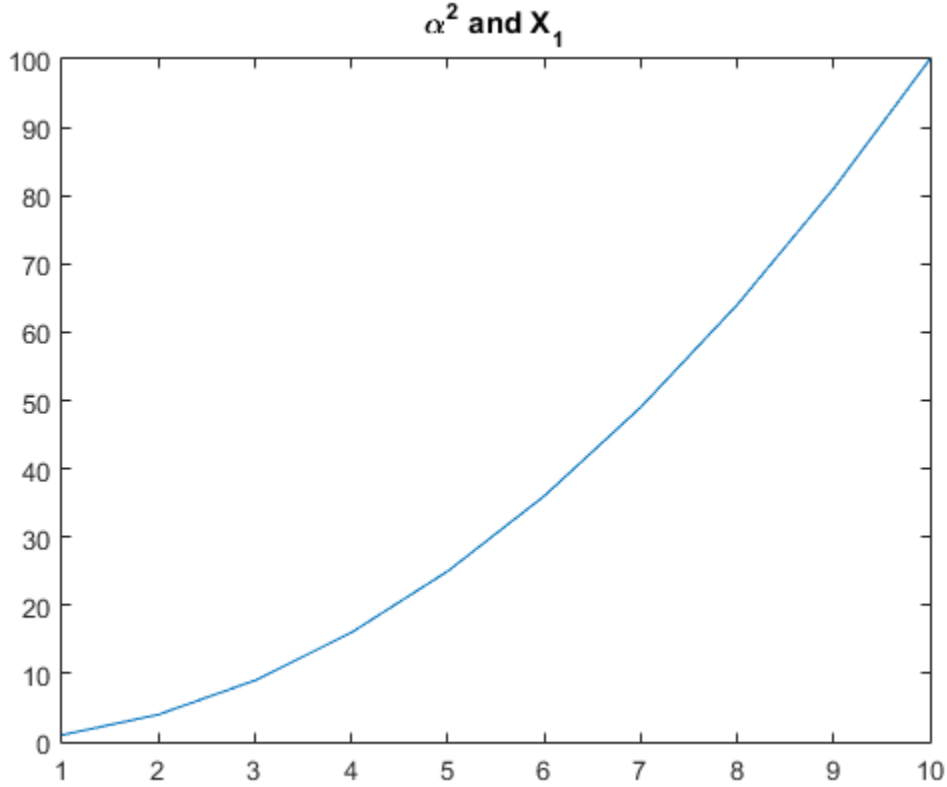
```
t = (0:0.01:0.2);
y = exp(-25*t);
figure
plot(t,y)
title('y = \ite^{\lambda t}','Color','b')
```



The `'Interpreter'` property must be `'tex'` (the default).

Include Superscript or Subscript Character in Title

```
figure
plot((1:10).^2)
title('\alpha^2 and X_1')
```

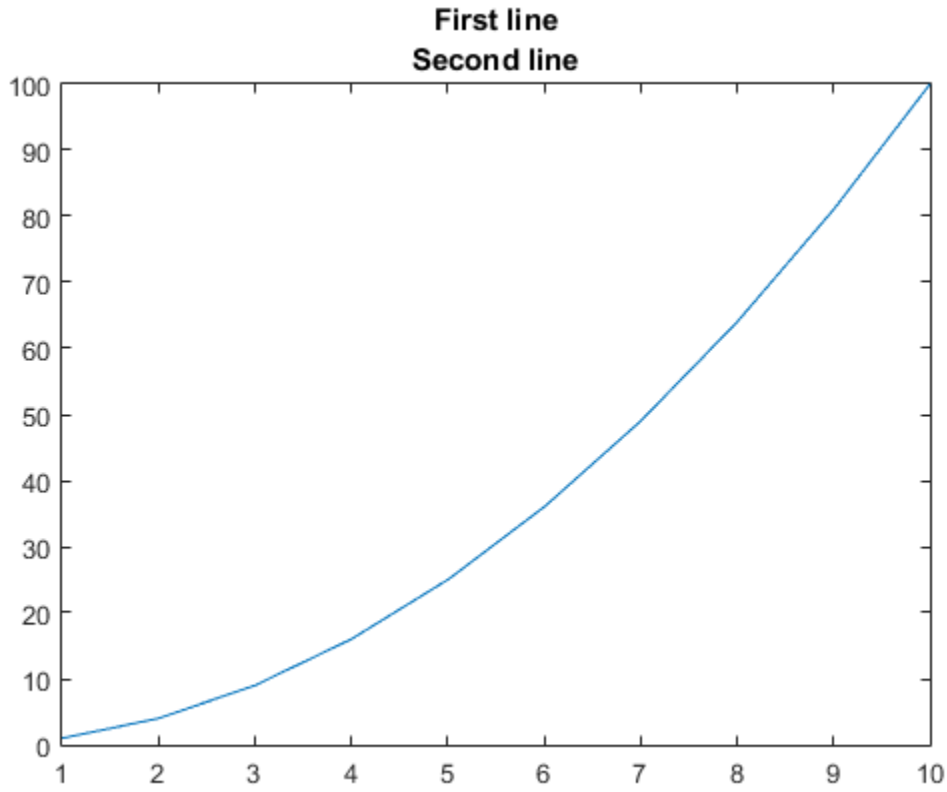


The superscript character, "^", and the subscript character, "_", modify the character or substring defined in braces immediately following.

Create Multiline Title

Create a multiline title using a multiline cell array.

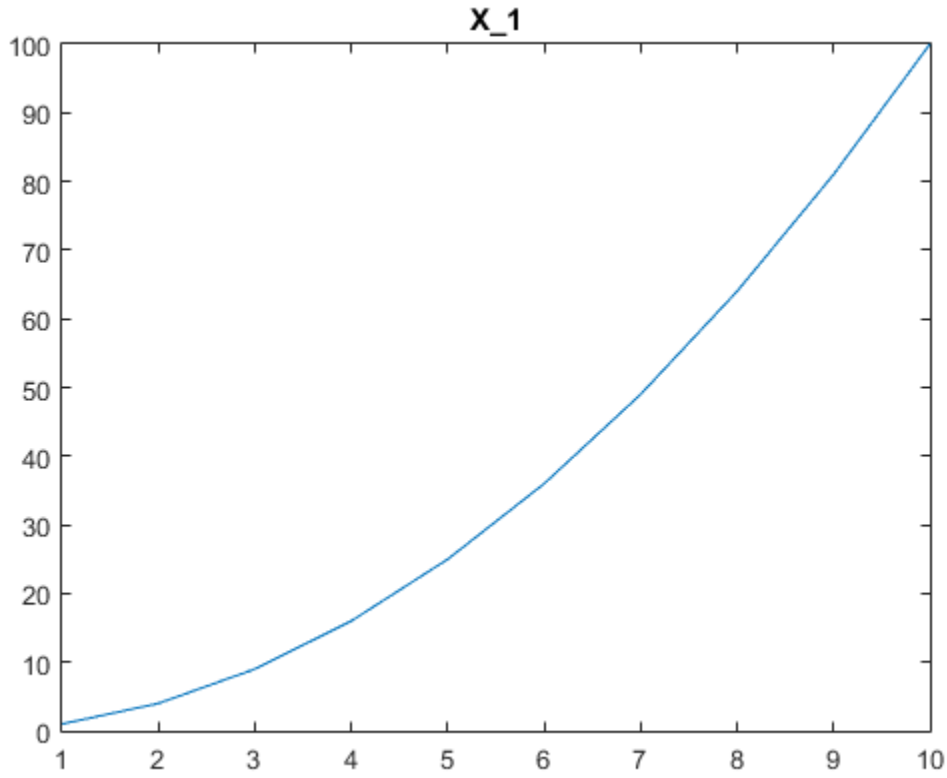
```
figure
plot((1:10).^2)
title({'First line';'Second line'})
```



Display Text As Typed

Set the `Interpreter` property as `'none'` so that the string `X_1` is displayed in the figure as typed, without making `1` a subscript of `X`.

```
figure
plot((1:10).^2)
title('X_1','Interpreter','none')
```

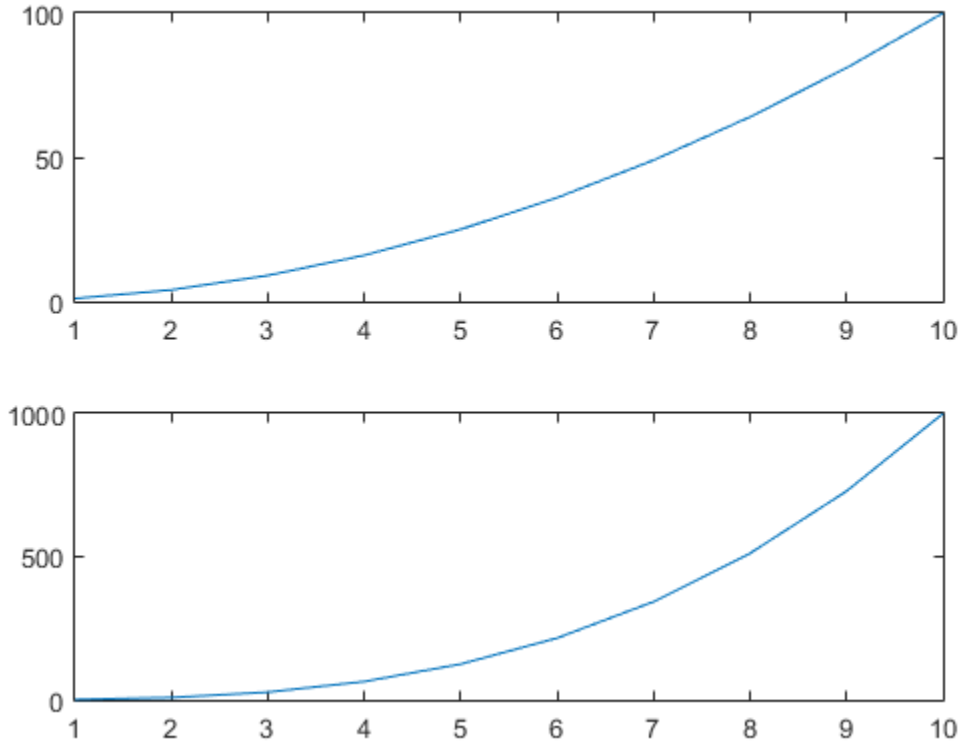


MATLAB® displays the string `X_1` in the title of the figure.

Add Title to Specific Axes

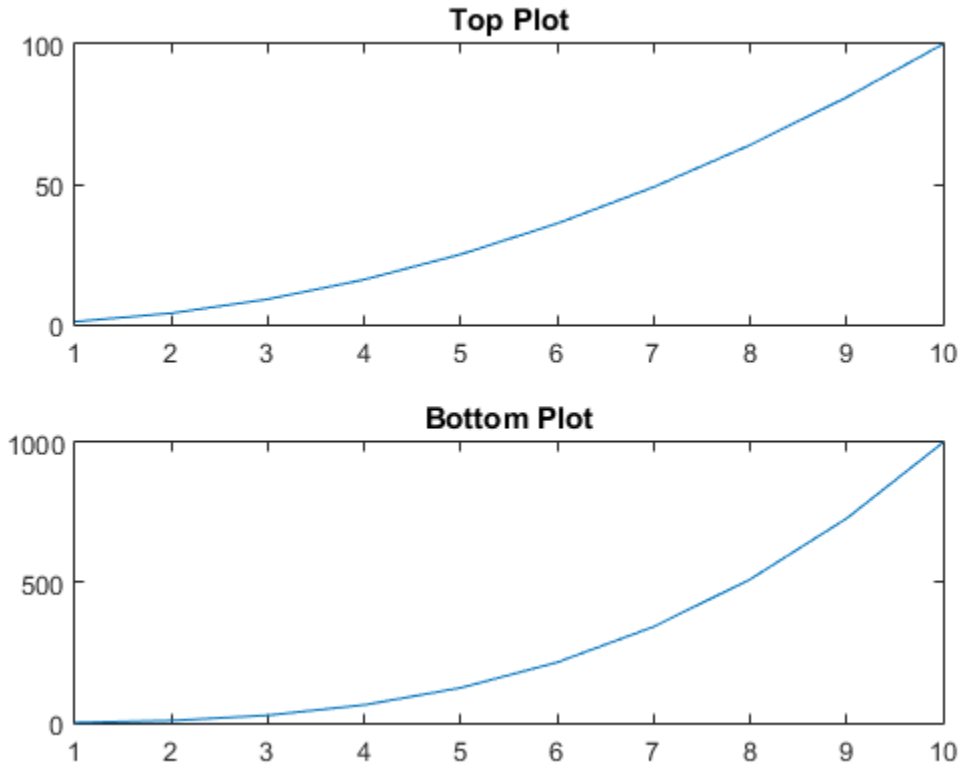
Create two subplots and return the handles to the axes objects, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
plot((1:10).^2)
s(2) = subplot(2,1,2);
plot((1:10).^3)
```



Add a title to each subplot by referring to its axes handle, `s(1)`, or `s(2)`.

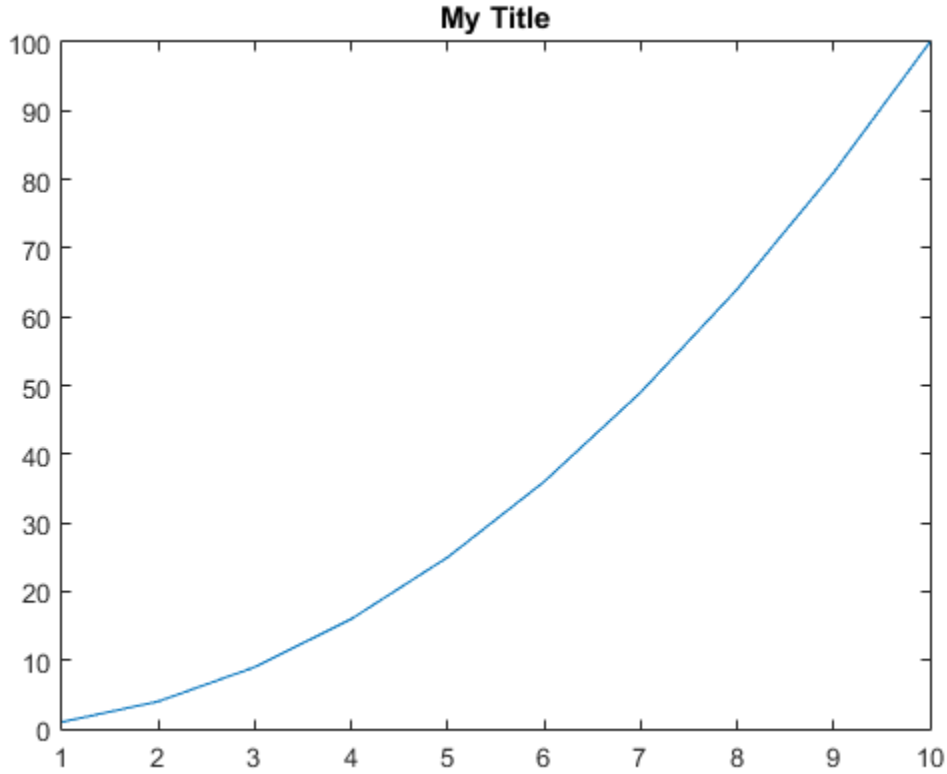
```
title(s(1),'Top Plot')
title(s(2),'Bottom Plot')
```



Add Title and Return Text Handle

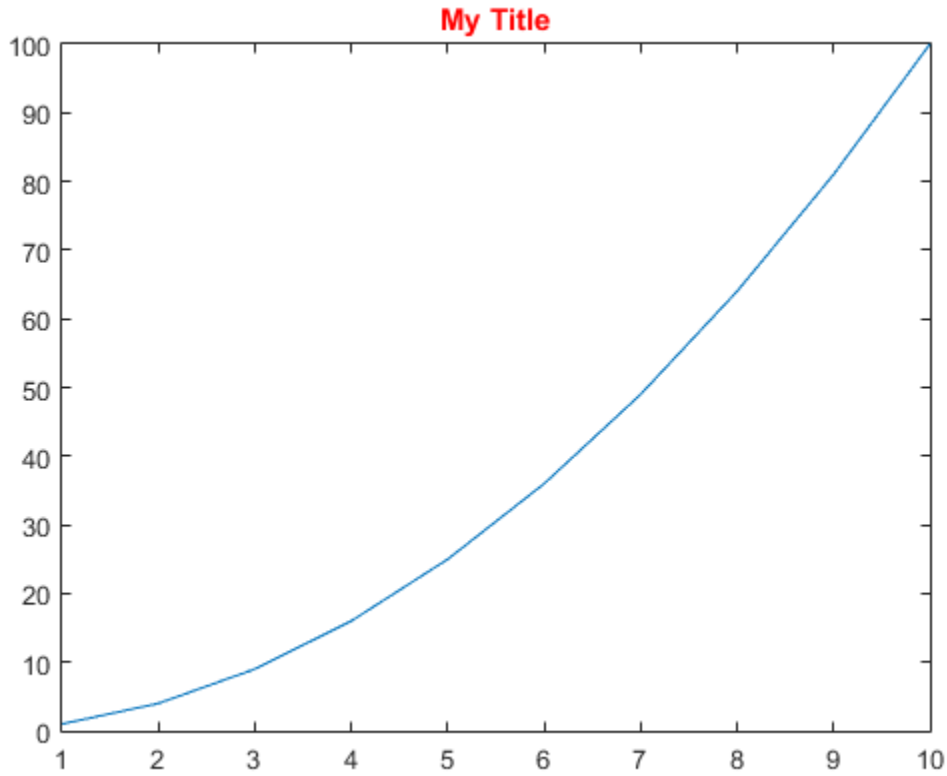
Add a title to a plot and return the text object.

```
plot((1:10).^2)
t = title('My Title');
```



Set the color of the title to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
t.Color = 'red';
```



Input Arguments

`str` — Text to display as title

character array | cell array | numeric value

Text to display as title, specified as a character array, cell array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: `123`

To include numeric variables with text in a title, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the Interpreter property.

To create multiline titles:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create a string with a new line character, such as `sprintf('first line \n second line')`.

Numeric titles are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note: The words `default`, `factory`, and `remove` are reserved words that will not appear in a title when quoted as a normal string. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

---

ax — Axes object

axes object

Axes object. If you do not specify an axes, then the `title` function uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',14` adds a title with red, 14-point font.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text Properties.

`'FontSize'` — Font size

`11` (default) | scalar value greater than `0`

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Setting the font size properties for the associated axes also affects the title font size. The title font size updates to equal the axes font size times the title scale factor. The `FontSize` property of the axes contains the axes font size. The `TitleFontSizeMultiplier` property of the axes contains the scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the title font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'FontWeight'` — Thickness of text characters

`'bold'` (default) | `'normal'`

Thickness of the text characters, specified as one of these values:

- `'bold'` — Thicker characters outlines than normal
- `'normal'` — Normal weight as defined by the particular font

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

The `TitleFontWeight` property for the associated axes affects the `FontWeight` value for the title.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

`'FontName'` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`'Color'` — Text color

`[0 0 0]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. The default color is black with an RGB triplet value of `[0 0 0]`. If you set the color to `'none'`, then the text is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`'Interpreter'` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`

Modifier	Description	Example of String
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsui`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ε	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrighta`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	Θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

Output Arguments

h — Text object

text object

Text object used as the title. Use `h` to access and modify properties of the title after its created.

More About

- Adding Titles to Graphs

See Also

Functions

`num2str` | `text` | `xlabel` | `ylabel` | `zlabel`

Properties

Text Properties

Introduced before R2006a

todatenum

Convert CDF epoch object to MATLAB serial date number

Syntax

```
n = todatenum(obj)
```

Description

`n = todatenum(obj)` converts the CDF epoch object `ep_obj` into a MATLAB serial date number. Note that a CDF epoch is the number of milliseconds since `01-Jan-0000` whereas a MATLAB datenum is the number of days since 00`-Jan-0000`.

Examples

Construct a CDF epoch object from a date string, and then convert the object back into a MATLAB date string:

```
dstr = datestr(today)
dstr =
   08-Oct-2003

obj = cdfepoch(dstr)
obj =
     cdfepoch object:
     08-Oct-2003 00:00:00

dstr2 = datestr(todatenum(obj))
dstr2 =
   08-Oct-2003
```

See Also

`cdfepoch` | `cdfinfo` | `cdfread` | `datenum` | `datetime`

Introduced before R2006a

toeplitz

Toeplitz matrix

Syntax

```
T = toeplitz(c,r)
T = toeplitz(r)
```

Description

A *Toeplitz* matrix is defined by one row and one column. A *symmetric Toeplitz* matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

`T = toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix `T` having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` are different, a message is printed and the column element is used.

For a real vector `r`, `T = toeplitz(r)` returns the symmetric Toeplitz matrix formed from vector `r`, where `r` defines the first row of the matrix. For a complex vector `r` with a real first element, `T = toeplitz(r)` returns the Hermitian Toeplitz matrix formed from `r`, where `r` defines the first row of the matrix and `r'` defines the first column. When the first element of `r` is not real, the resulting matrix is Hermitian off the main diagonal, i.e., $T_{ij} = \mathrm{conj}(T_{ji})$ for $i \neq j$.

Examples

A Toeplitz matrix with diagonal disagreement is

```
c = [1  2  3  4  5];
r = [1.5  2.5  3.5  4.5  5.5];
toeplitz(c,r)
Column wins diagonal conflict:
ans =
    1.000    2.500    3.500    4.500    5.500
```

```
    2.000    1.000    2.500    3.500    4.500
    3.000    2.000    1.000    2.500    3.500
    4.000    3.000    2.000    1.000    2.500
    5.000    4.000    3.000    2.000    1.000
```

See Also

`hankel` | `kron`

Introduced before R2006a

toolboxdir

Root folder for specified toolbox

Syntax

```
toolboxdir('tbxFolderName')
s = toolboxdir('tbxFolderName')
s = toolboxdir tbxFolderName
```

Description

`toolboxdir('tbxFolderName')` returns a string that is the absolute path to the specified toolbox, `tbxFolderName`, where `tbxFolderName` is the folder name for the toolbox.

`s = toolboxdir('tbxFolderName')` returns the absolute path to the specified toolbox to the output argument, `s`.

`s = toolboxdir tbxFolderName` is the command form of the syntax.

Examples

Obtain the path for the Control System Toolbox software:

```
s = toolboxdir('control')
```

MATLAB returns:

```
s = C:\Program Files\MATLAB\R2012a\toolbox\control
```

More About

Tips

`toolboxdir` is particularly useful for MATLAB Compiler software. The base folder of all toolboxes installed with MATLAB software is:

matlabroot/**toolbox**/tbxFolderName

However, in deployed mode, the base folders of the toolboxes are different. `toolboxdir` returns the correct root folder, whether running from MATLAB or from an application deployed with the MATLAB Compiler software.

To determine the folder name for a given toolbox, run the following code, substituting the name of a product function for *toolbxfcn*:

```
n = 'toolbxfcn';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

For example, to determine the product name for Control System Toolbox set `n` to the name of a function unique to Control System Toolbox, such as `dss`:

```
n = 'dss'
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+'
regexp(which(n), pat, 'match', 'once')
```

```
control
```

See Also

`fullfile` | `path` | `matlabroot`

trace

Sum of diagonal elements

Syntax

```
b = trace(A)
```

Description

`b = trace(A)` is the sum of the diagonal elements of the matrix `A`.

More About

Algorithms

```
t = sum(diag(A));
```

See Also

`det` | `eig`

Introduced before R2006a

transpose, .'

Transpose

Syntax

```
B = A.'
B = transpose(A)
```

Description

`B = A.'` computes the nonconjugate transpose of `A`.

`B = transpose(A)` is an alternate way to execute `A.'`, but is rarely used. It enables operator overloading for classes.

Examples

Transpose of Real Matrix

Create a 4-by-4 matrix.

```
A = magic(4)
```

```
A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Find the transpose of `A`.

```
B = A.'
```

```
B =

    16     5     9     4
```

```
     2    11     7    14
     3    10     6    15
    13     8    12     1
```

The result, B, has the same elements as A, but the row and column index for each element are interchanged.

Transpose of Complex Matrix

Create a 2-by-4 matrix containing complex elements.

```
A = [1 3 4-1i 2+2i; 0+1i 1-1i 5 6-1i]
```

```
A =

   1.0000 + 0.0000i   3.0000 + 0.0000i   4.0000 - 1.0000i   2.0000 + 2.0000i
   0.0000 + 1.0000i   1.0000 - 1.0000i   5.0000 + 0.0000i   6.0000 - 1.0000i
```

Find the transpose of A.

```
B = A.'
```

```
B =

   1.0000 + 0.0000i   0.0000 + 1.0000i
   3.0000 + 0.0000i   1.0000 - 1.0000i
   4.0000 - 1.0000i   5.0000 + 0.0000i
   2.0000 + 2.0000i   6.0000 - 1.0000i
```

The result, B, is a 4-by-2 matrix. The non-conjugate transpose does not change the sign of the imaginary parts of the complex elements.

Input Arguments

A — Input array

vector | matrix | cell array | categorical array | datetime array | duration array | calendarDuration array | structure field

Input array, specified as a vector or matrix of any numeric, logical, or `char` data type, or as a cell array, categorical array, datetime array, duration array, calendarDuration array, or structure field.

Complex Number Support: Yes

More About

Nonconjugate Transpose

The nonconjugate transpose of a matrix interchanges the row and column index for each element, reflecting the elements across the main diagonal, with the diagonal elements themselves unchanged. This operation does not affect the sign of the imaginary parts of complex elements.

For example, if `B = A.'` and `A(3,2)` is `1+1i`, then the element `B(2,3)` is `1+1i`.

Tips

- The complex conjugate transpose operator, `A'`, also negates the sign of the imaginary portion of the complex elements in `A`.
- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`conj` | `ctranspose` | `permute`

Introduced before R2006a

trapz

Trapezoidal numerical integration

Syntax

```
Q = trapz(Y)
Q = trapz(X,Y)
Q = trapz( ___ ,dim)
```

Description

`Q = trapz(Y)` returns the approximate integral of `Y` via the trapezoidal method with unit spacing. The size of `Y` determines the dimension to integrate along:

- If `Y` is a vector, then `trapz(Y)` is the approximate integral of `Y`.
- If `Y` is a matrix, then `trapz(Y)` integrates over each column and returns a row vector of integration values.
- If `Y` is a multidimensional array, then `trapz(Y)` integrates over the first dimension whose size does not equal 1. The size of this dimension becomes 1, and the sizes of other dimensions remain unchanged.

`Q = trapz(X,Y)` integrates `Y` with spacing increment `X`. By default, `trapz` operates on the first dimension of `Y` whose size does not equal 1. `length(X)` must be equal to the size of this dimension . If `X` is a scalar, then `trapz(X,Y)` is equivalent to `X*trapz(Y)`.

`Q = trapz( ___ ,dim)` integrates along the dimension `dim` using any of the previous syntaxes. You must specify `Y`, and optionally can specify `X`. The length of `X`, if specified, must be the same as `size(Y,dim)`. For example, if `Y` is a matrix, then `trapz(X,Y,2)` integrates each row of `Y`.

Examples

Integrate Vector of Data with Unit Spacing

Create a numeric vector of data.

```
Y = [1 4 9 16 25];
```

Y contains function values for $f(x) = x^2$ in the domain [1,5].

Use trapz to integrate the data points with unit spacing.

```
Q = trapz(Y)
```

```
Q =

    42
```

This approximate integration yields a value of 42. In this case, the exact answer is a little less, $41\frac{1}{3}$. The trapz function overestimates the value of the integral because *f(x)* is concave up.

Integrate Vector of Data with Nonunit Spacing

Create a domain vector, X.

```
X = 0:pi/100:pi;
```

Calculate the sine of X and store the result in Y.

```
Y = sin(X);
```

Integrate the function values contained in Y using trapz.

```
Q = trapz(X,Y)
```

```
Q =

    1.9998
```

When the spacing between points is constant, but not equal to 1, you can multiply by the spacing value, in this case pi/100*trapz(Y). The answer is the same if you pass the value directly to the function with trapz(X,Y).

Integrate Matrix with Nonuniform Spacing

Create a vector of time values, X. Also create a matrix, Y, containing values evaluated at the irregular intervals in X.

```
X = [1 2.5 7 10]';
```

```
Y = [5.2 4.8 4.9 5.1; 7.7 7.0 6.5 6.8; 9.6 10.5 10.5 9.0; 13.2 14.5 13.8 15.2]
```

```
Y =

    5.2000    4.8000    4.9000    5.1000
    7.7000    7.0000    6.5000    6.8000
    9.6000   10.5000   10.5000    9.0000
   13.2000   14.5000   13.8000   15.2000
```

The columns of `Y` represent velocity data, taken at the times contained in `X`, for several different trials.

Use `trapz` to integrate each column independently and find the total distance traveled in each trial. Since the function values are not evaluated at constant intervals, specify `X` to indicate the spacing between the data points.

```
Q = trapz(X,Y)
```

```
Q =

   82.8000   85.7250   83.2500   80.7750
```

The result is a row vector of integration values, one for each column in `Y`. By default, `trapz` integrates along the first dimension of `Y` whose size does not equal 1.

Alternatively, you can integrate the rows of a matrix by specifying `dim = 2`.

In this case, use `trapz` on `Y'`, which contains the velocity data in the rows.

```
dim = 2;
Q1 = trapz(X,Y',dim)
```

```
Q1 =

   82.8000
   85.7250
   83.2500
   80.7750
```

The result is a column vector of integration values, one for each row in `Y'`.

Multiple Numerical Integrations

Created a grid of domain values.

```
x = -3:.1:3;
```

```
y = -5:.1:5;
[X,Y] = meshgrid(x,y);
```

Calculate the function $f(x,y) = x^2 + y^2$ over the grid.

```
F = X.^2 + Y.^2;
```

`trapz` integrates numeric data rather than functional expressions, so in general the expression does not need to be known to use `trapz` on a matrix of data.

Use `trapz` to approximate the double integral

$$I = \int_{-5}^{5} \int_{-3}^{3} \left(x^2 + y^2\right) dx dy .$$

To perform double or triple integrations on an array of numeric data, nest function calls to `trapz`.

```
I = trapz(y,trapz(x,F,2))
```

```
I =

  680.2000
```

`trapz` performs the integration over *x* first, producing a column vector. Then, the integration over *y* reduces the column vector to a single scalar. `trapz` slightly overestimates the exact answer of 680 because *f(x,y)* is concave up.

Input Arguments

`Y` — Numeric data

vector | matrix | multidimensional array

Numeric data, specified as a vector, matrix, or multidimensional array. By default, `trapz` integrates along the first dimension of `Y` whose size does not equal 1.

Data Types: `single` | `double`
Complex Number Support: Yes

`X` — Point spacing

1 (default) | uniform scalar spacing | vector of nonuniform spacings

Point spacing, specified as `1` (default), a uniform scalar spacing, or a vector of nonuniform spacings. If `X` is a vector, then `length(X)` must be the same as the size of the integration dimension in `Y`.

Data Types: `single` | `double`
Complex Number Support: Yes

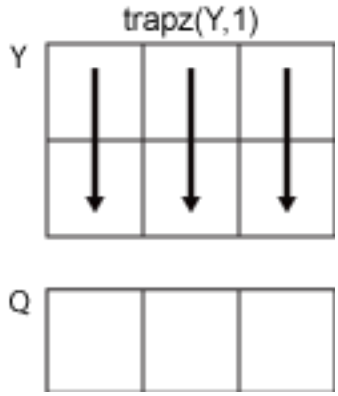
`dim` — Dimension to operate along

positive integer scalar

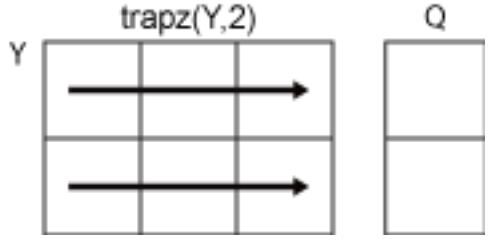
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `Y`:

- `trapz(Y,1)` works on successive elements in the columns of `Y` and returns a row vector of integration values.



- `trapz(Y,2)` works on successive elements in the rows of `Y` and returns a column vector of integration values.



If `dim` is greater than `ndims(Y)`, then `trapz` returns an array of zeros of the same size as `Y`.

More About

Trapezoidal Method

`trapz` performs numerical integration via the trapezoidal method. This method approximates the integration over an interval by breaking the area down into trapezoids with more easily computable areas.

For an integration with `N+1` evenly spaced points, the approximation is

$$\int_a^b f(x)dx \approx \frac{b-a}{2N}\sum_{n=1}^{N}\left(f(x_n)+f(x_{n+1})\right)$$
$$= \frac{b-a}{2N}\left[f(x_1)+2f(x_2)+\ldots+2f(x_N)+f(x_{N+1})\right],$$

where the spacing between each point is equal to the scalar value $\frac{b-a}{N}$.

If the spacing between the points is not constant, then the formula generalizes to

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{n=1}^{N}(x_{n+1}-x_n)\left[f(x_n)+f(x_{n+1})\right],$$

where $(x_{n+1}-x_n)$ is the spacing between each consecutive pair of points.

Tips

- `trapz` reduces the size of the dimension it operates on to 1, and returns only the final integration value. `cumtrapz` also returns the intermediate integration values, preserving the size of the dimension it operates on.
- “Integration of Numeric Data”

See Also

`cumsum` | `cumtrapz` | `integral` | `integral2` | `integral3`

Introduced before R2006a

treelayout

Lay out tree or forest

Syntax

```
[x,y] = treelayout(parent,post)
[x,y,h,s] = treelayout(parent,post)
```

Description

`[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with `0` for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

See Also

`etree` | `treeplot` | `etreeplot` | `symbfact`

Introduced before R2006a

treeplot

Plot picture of tree

Syntax

```
treeplot(p)
treeplot(p,nodeSpec,edgeSpec)
```

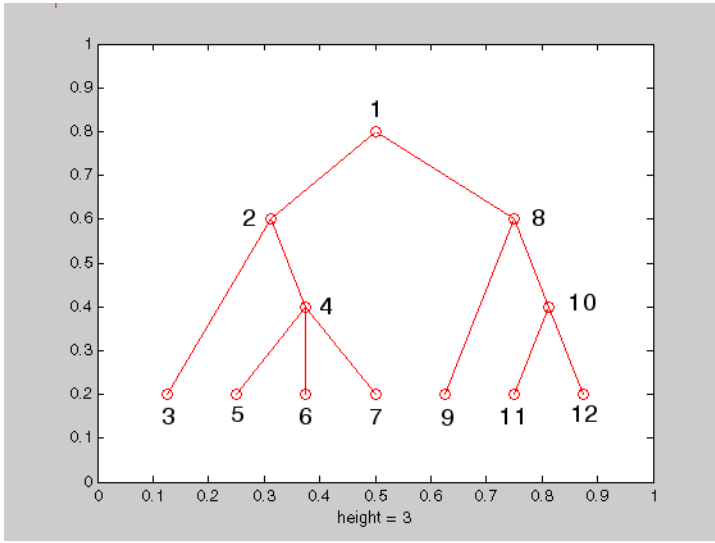
Description

`treeplot(p)` plots a picture of a tree given a vector of parent pointers, with `p(i)` = 0 for a root.

`treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

Examples

To plot a tree with 12 nodes, call `treeplot` with a 12-element input vector. The index of each element in the vector is shown adjacent to each node in the figure below. (These indices are shown only for the point of illustrating the example; they are not part of the `treeplot` output.)



To generate this plot, set the value of each element in the `nodes` vector to the index of its parent, (setting the parent of the root node to zero).

The node marked `1` in the figure is represented by `nodes(1)` in the input vector, and because this is the root node which has a parent of zero, you set its value to zero:

```
nodes(1) = 0;      % Root node
```

`nodes(2)` and `nodes(8)` are children of `nodes(1)`, so set these elements of the input vector to `1`:

```
nodes(2) = 1;      nodes(8) = 1;
```

`nodes(5:7)` are children of `nodes(4)`, so set these elements to `4`:

```
nodes(5) = 4;      nodes(6) = 4;      nodes(7) = 4;
```

Continue in this manner until each element of the vector identifies its parent. For the plot shown above, the `nodes` vector now looks like this:

```
nodes = [0 1 2 2 4 4 4 1 8 8 10 10];
```

Now call `treeplot` to generate the plot:

```
treeplot(nodes)
```

See Also

`etree` | `etreeplot` | `treelayout`

Introduced before R2006a

tril

Lower triangular part of matrix

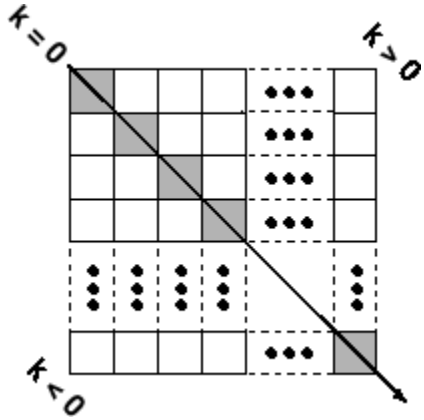
Syntax

```
L = tril(X)
L = tril(X,k)
```

Description

`L = tril(X)` returns the lower triangular part of `X`.

`L = tril(X,k)` returns the elements on and below the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
tril(ones(4,4),-1)

ans =
```

```
     0     0     0     0
     1     0     0     0
     1     1     0     0
     1     1     1     0
```

See Also

`diag` | `triu`

Introduced before R2006a

trimesh

Triangular mesh plot

Syntax

```
trimesh(Tri,X,Y,Z,C)
trimesh(Tri,X,Y,Z)
trimesh(Tri,X,Y)
trimesh(TR)
trimesh(...'PropertyName',PropertyValue...)
h = trimesh(...)
```

Description

`trimesh(Tri,X,Y,Z,C)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a mesh. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices. The edge color is defined by the vector `C`.

`trimesh(Tri,X,Y,Z)` uses `C = Z` so color is proportional to surface height.

`trimesh(Tri,X,Y)` displays the triangles in a 2-D plot.

`trimesh(TR)` displays the triangles in a triangulation representation.

`trimesh(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

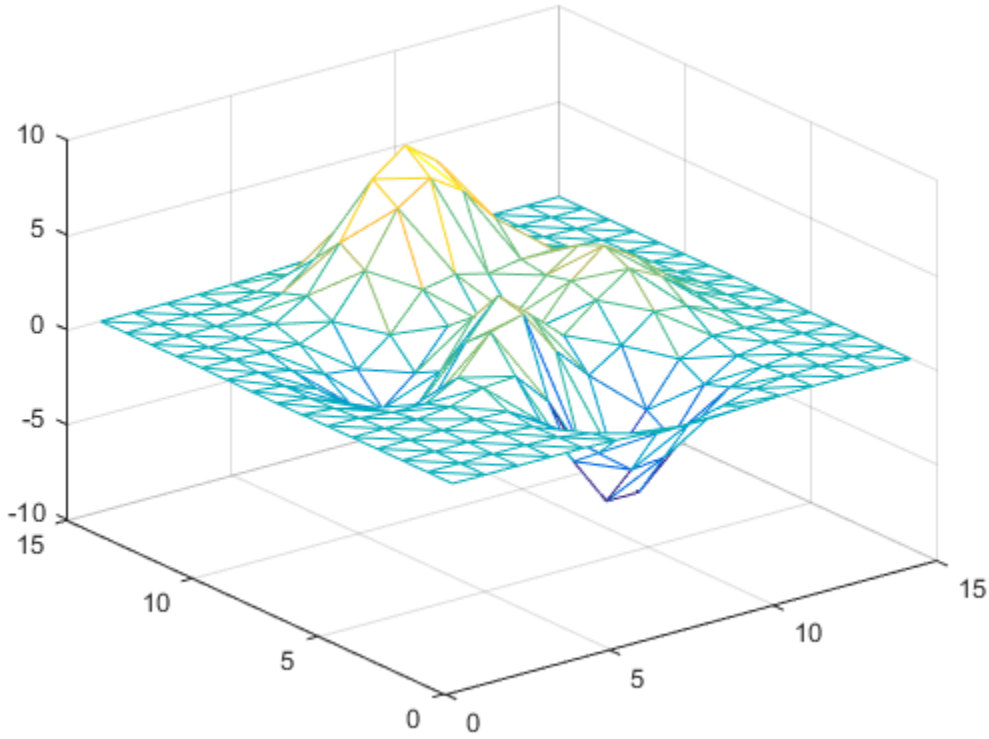
`h = trimesh(...)` returns a handle to the displayed triangles.

Examples

Create Triangular Mesh Plot

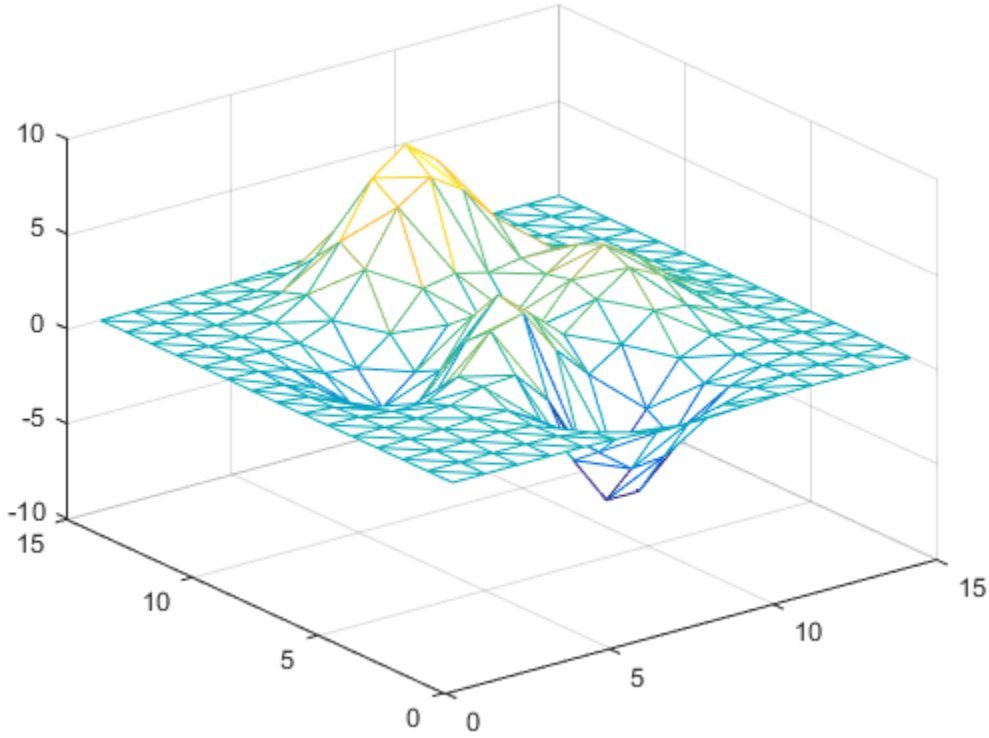
Create vertex vectors and a face matrix, and then create a triangular mesh plot.

```
[x,y] = meshgrid(1:15,1:15);
tri = delaunay(x,y);
z = peaks(15);
trimesh(tri,x,y,z)
```



If the surface is already a triangulation representation, then you can pass the triangulation to `trimesh`:

```
tr = triangulation(tri,x(:),y(:),z(:));
trimesh(tr)
```



See Also

`patch` | `trisurf` | `triangulation` | `delaunay` | `delaunayTriangulation`

Introduced before R2006a

triplequad

Numerically evaluate triple integral

Compatibility

`triplequad` will be removed in a future release. Use `integral3` instead.

Syntax

```
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)
```

Description

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)` evaluates the triple integral `fun(x,y,z)` over the three dimensional rectangular region `xmin <= x <= xmax`, `ymin <= y <= ymax`, `zmin <= z <= zmax`. The first input, `fun`, is a function handle. `fun(x,y,z)` must accept a vector `x` and scalars `y` and `z`, and return a vector of values of the integrand.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples

Pass function handle `@integrnd` to `triplequad`:P

```
Q = triplequad(@integrnd,0,pi,0,1,-1,1);
```

where the file `integrnd.m` is

```
function f = integrnd(x,y,z)
f = y*sin(x)+z*cos(x);
```

Pass anonymous function handle `F` to `triplequad`:

```
F = @(x,y,z)y*sin(x)+z*cos(x);
Q = triplequad(F,0,pi,0,1,-1,1);
```

This example integrates `y*sin(x)+z*cos(x)` over the region `0 <= x <= pi`, `0 <= y <= 1`, `-1 <= z <= 1`. Note that the integrand can be evaluated with a vector `x` and scalars `y` and `z`.

More About

- "Anonymous Functions"

See Also

`dblquad` | `quad2d` | `quad` | `quadgk` | `quadl` | `function_handle` | `integral` | `integral2` | `integral3`

Introduced before R2006a

triplot

2-D triangular plot

Note: The behavior of `h = triplot(...)` has changed. The new behavior returns a single chart line handle.

Syntax

```
triplot(TRI,x,y)
triplot(TRI,x,y,color)
triplot(TR)
h = triplot(...)
triplot(...,'param','value','param','value'...)
```

Description

`triplot(TRI,x,y)` displays the triangles defined in the `m`-by-3 matrix `TRI`. A row of `TRI` contains indices into the vectors `x` and `y` that define a single triangle. The default line color is blue.

`triplot(TRI,x,y,color)` uses the string `color` as the line color. `color` can also be a line specification. See `ColorSpec` for a list of valid color strings. See `LineSpec` for information about line specifications.

`triplot(TR)` displays the triangles in a triangulation representation.

`h = triplot(...)` returns a single chart line handle to the displayed triangles.

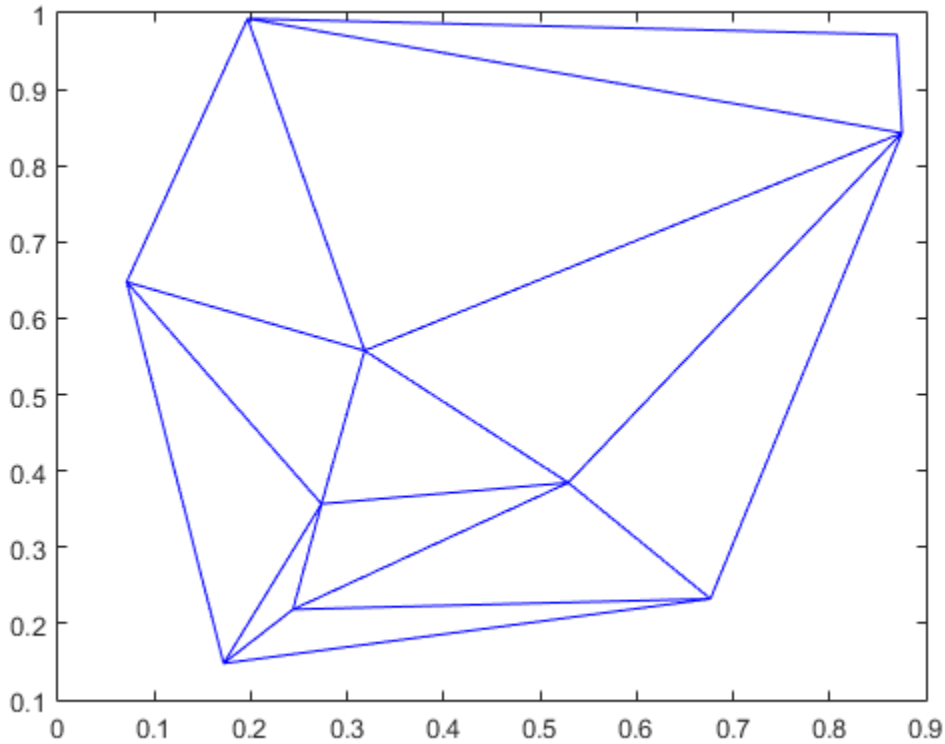
`triplot(...,'param','value','param','value'...)` allows additional line property name/property value pairs to be used when creating the plot. See Chart Line Properties for information about the available properties.

Examples

Plot Delaunay Triangulation

Plot a Delaunay triangulation for 10 randomly generated points.

```
P = gallery('uniformdata',10,2,2);
DT = delaunayTriangulation(P);
triplot(DT)
```



See Also

`delaunayTriangulation` | `triangulation` | `delaunay` | `trimesh` | `trisurf`

Introduced before R2006a

TriRep class

(Will be removed) Triangulation representation

Note: `TriRep` will be removed in a future release. Use `triangulation` instead.

Description

`TriRep` provides topological and geometric queries for triangulations in 2-D and 3-D space. For example, for triangular meshes you can query triangles attached to a vertex, triangles that share an edge, neighbor information, circumcenters, or other features. You can create a `TriRep` directly using existing triangulation data. Alternatively, you can create a Delaunay triangulation, via `DelaunayTri`, which provides access to the `TriRep` functionality.

Construction

.TriRep	(Will be removed) Triangulation representation

Methods

baryToCart	(Will be removed) Convert point coordinates from barycentric to Cartesian
cartToBary	(Will be removed) Convert point coordinates from cartesian to barycentric
circumcenters	(Will be removed) Circumcenters of specified simplices

edgeAttachments	(Will be removed) Simplices attached to specified edges
edges	(Will be removed) Triangulation edges
faceNormals	(Will be removed) Unit normals to specified triangles
featureEdges	(Will be removed) Sharp edges of surface triangulation
freeBoundary	(Will be removed) Facets referenced by only one simplex
incenters	(Will be removed) Incenters of specified simplices
isEdge	(Will be removed) Test if vertices are joined by edge
neighbors	(Will be removed) Simplex neighbor information
size	(Will be removed) Size of triangulation matrix
vertexAttachments	(Will be removed) Return simplices attached to specified vertices

Properties

X	Coordinates of the points in the triangulation
Triangulation	Triangulation data structure

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Indexing

`TriRep` objects support indexing into the triangulation using parentheses (). The syntax is the same as for arrays.

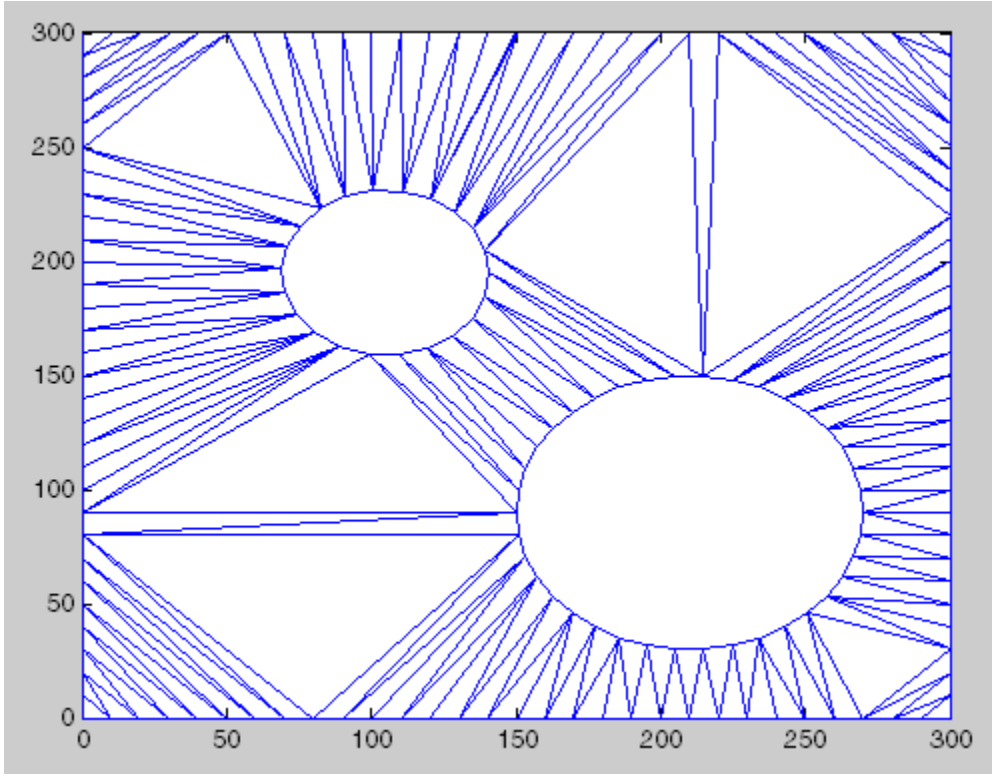
Examples

Load a 2-D triangulation and use the `TriRep` constructor to build an array of the free boundary edges:

```
load trimesh2d
```

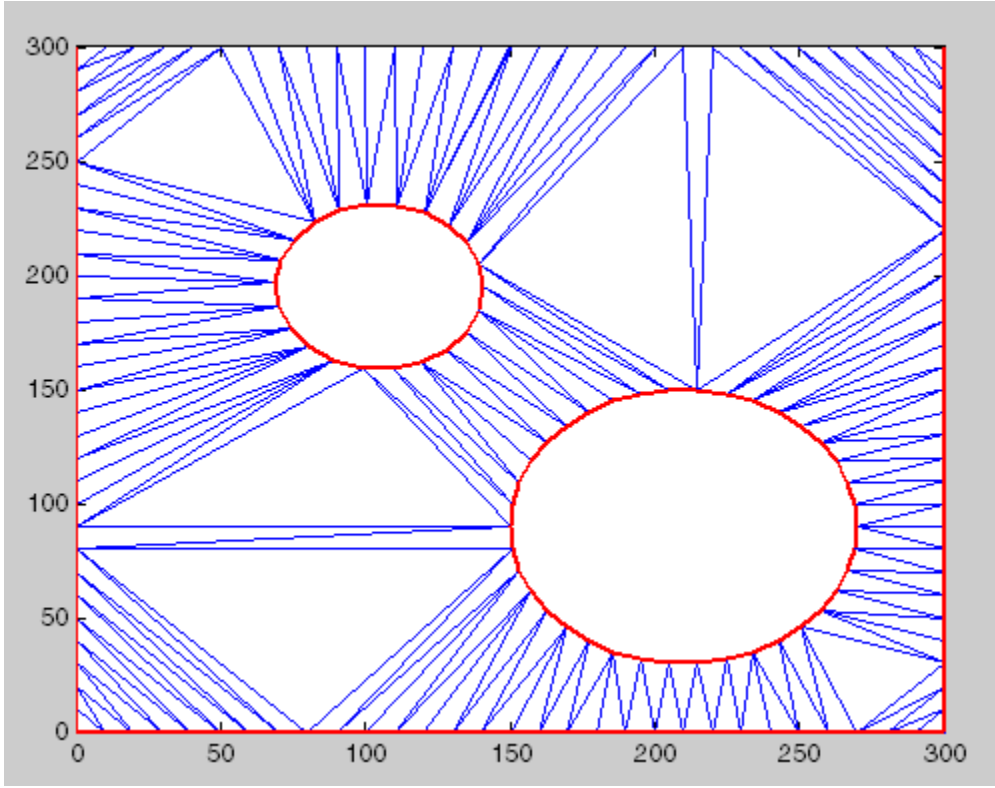
This loads triangulation `tri` and vertex coordinates `x`, `y`:

```
trep = TriRep(tri, x,y);
fe = freeBoundary(trep)';
triplot(trep);
```



You can add the free edges `fe` to the plot:

```
hold on;
plot(x(fe), y(fe), 'r','LineWidth',2);
hold off;
axis([-50 350 -50 350]);
axis equal;
```



See Also

`triangulation` | `delaunayTriangulation` | `scatteredInterpolant`

TriRep

Class: TriRep

(Will be removed) Triangulation representation

Note: `TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
TR = TriRep(TRI, X, Y)
TR = TriRep(TRI, X, Y, Z)
TR = TriRep(TRI, X)
```

Description

`TR = TriRep(TRI, X, Y)` creates a 2-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y)`. `TRI` is an `m`-by-3 matrix that defines the triangulation in face-vertex format, where `m` is the number of triangles. Each row of `TRI` is a triangle defined by indices into the column vector of vertex coordinates `(X, Y)`.

`TR = TriRep(TRI, X, Y, Z)` creates a 3-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y, Z)`. `TRI` is an `m`-by-3 or `m`-by-4 matrix that defines the triangulation in simplex-vertex format, where where m is the number of simplices; triangles or tetrahedra in this case. Each row of `TRI` is a simplex defined by indices into the column vector of vertex coordinates `(X, Y, Z)`.

`TR = TriRep(TRI, X)` creates a triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `X`. `TRI` is an `m`-by-`n` matrix that defines the triangulation in simplex-vertex format, where `m` is the number of simplices and `n` is the number of vertices per simplex. Each row of `TRI` is a simplex defined by indices into the array of vertex coordinates `X`. `X` is an `mpts`-by-`ndim` matrix where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, where $2 \le$ `ndim` ≤ 3.

Examples

Load a 3-D tetrahedral triangulation compute the free boundary. First, load triangulation `tet` and vertex coordinates `X`.

```
load tetmesh
```

Create the triangulation representation and compute the free boundary.

```
trep = TriRep(tet, X);
[tri, Xb] = freeBoundary(trep);
```

See Also

`scatteredInterpolant` | `delaunayTriangulation`

TriScatteredInterp class

(Will be removed) Interpolate scattered data

Note: `TriScatteredInterp` will be removed in a future release. Use `scatteredInterpolant` instead.

Description

`TriScatteredInterp` is used to perform interpolation on a scattered dataset that resides in 2-D or 3-D space. A scattered data set defined by locations `X` and corresponding values `V` can be interpolated using a Delaunay triangulation of `X`. This produces a surface of the form `V = F(X)`. The surface can be evaluated at any query location `QX`, using `QV = F(QX)`, where `QX` lies within the convex hull of `X`. The interpolant `F` always goes through the data points specified by the sample.

Definitions

The *Delaunay triangulation* of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set. The *convex hull* of a set of points is the smallest convex set containing all points of the original set. These definitions extend naturally to higher dimensions.

Construction

.TriScatteredInterp	(Will be removed) Interpolate scattered data

Properties

`X`	Defines locations of scattered data points in 2-D or 3-D space.	
`V`	Defines value associated with each data point.	
`Method`	Defines method used to interpolate the data .	
	`natural`	Natural neighbor interpolation
	`linear`	Linear interpolation (default)
	`nearest`	Nearest neighbor interpolation

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Examples

Create a data set:

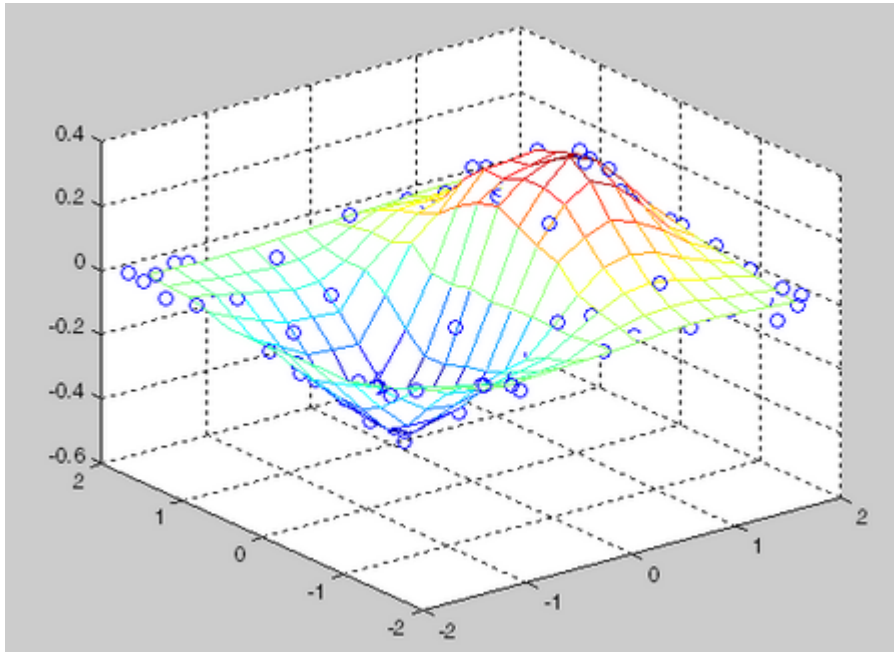
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations `(qx, qy)`. The corresponding value at these locations is `qz`:

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



See Also

`delaunayTriangulation` | `interp1` | `interp2` | `interp3` | `meshgrid`

TriScatteredInterp

Class: TriScatteredInterp

(Will be removed) Interpolate scattered data

Note: `TriScatteredInterp` will be removed in a future release. Use `scatteredInterpolant` instead.

Syntax

```
F = TriScatteredInterp()
F = TriScatteredInterp(X, V)
F = TriScatteredInterp(X, Y, V)
F = TriScatteredInterp(X, Y, Z, V)
F = TriScatteredInterp(DT, V)
F = TriScatteredInterp(..., method)
```

Description

`F = TriScatteredInterp()` creates an empty scattered data interpolant. This can subsequently be initialized with sample data points and values (`Xdata`, `Vdata`) via `F.X = Xdata` and `F.V = Vdata`.

`F = TriScatteredInterp(X, V)` creates an interpolant that fits a surface of the form `V = F(X)` to the scattered data in (`X`, `V`). `X` is a matrix of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside (`ndim` is 2 or 3). The column vector `V` defines the values at `X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(X, Y, V)` and `F = TriScatteredInterp(X, Y, Z, V)` allow the data point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`F = TriScatteredInterp(DT, V)` uses the specified `DelaunayTri` object `DT` as a basis for computing the interpolant. `DT` is a Delaunay triangulation of the scattered data locations, `DT.X`. The matrix `DT.X` is of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, `2 <= ndim <= 3`. `V` is a column vector that defines the values at `DT.X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(..., method)` allows selection of the technique `method` used to interpolate the data.

Input Arguments

`X`	Matrix of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside. Input may also be specified as column vectors (`X`, `Y`) or (`X`, `Y`, `Z`)	
`V`	Column vector that defines the values at `X`, where the length of `V` equals `mpts`.	
`DT`	Delaunay triangulation of the scattered data locations	
`method`	`natural`	Natural neighbor interpolation
	`linear`	Linear interpolation (default)
	`nearest`	Nearest-neighbor interpolation

Output Arguments

`F`	Creates an interpolant that fits a surface of the form `V = F(X)` to the scattered data.

Evaluation

To evaluate the interpolant, express the statement in Monge's form `Vq = F(Xq)`, `Vq = F(Xq,Yq)`, or `Vq = F(Xq,Yq,Zq)` where `Vq` is the value of the interpolant at the query location and `Xq`, `Yq`, and `Zq` are the vectors of point locations.

Definitions

The *Delaunay triangulation* of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set.

Examples

Create a data set:

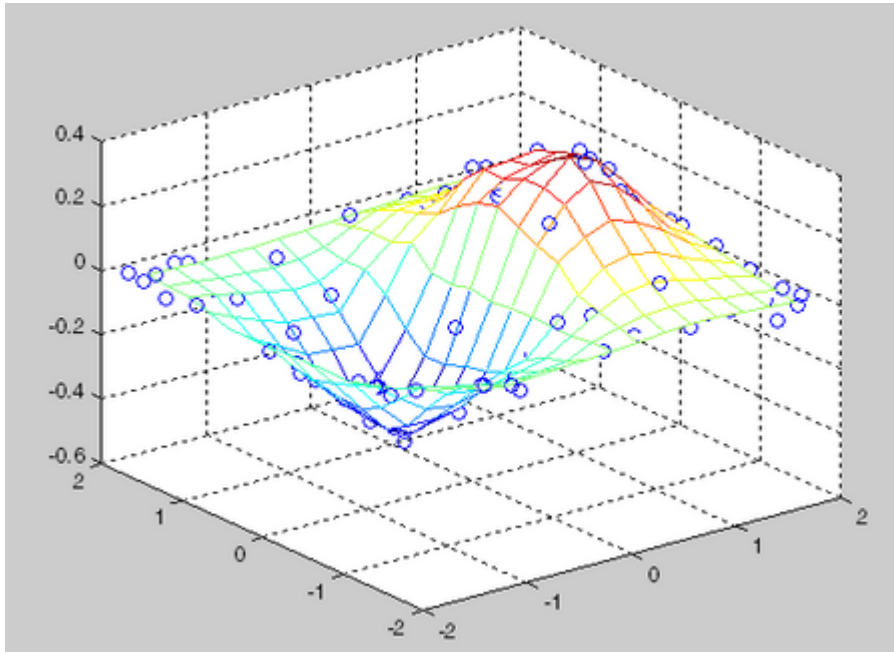
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations `(qx, qy)`. The corresponding value at these locations is `qz` .

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



See Also

`delaunayTriangulation` | `interp1` | `interp2` | `interp3` | `meshgrid`

trisurf

Triangular surface plot

Syntax

```
trisurf(Tri,X,Y,Z,C)
trisurf(Tri,X,Y,Z)
trisurf(TR)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)
```

Description

`trisurf(Tri,X,Y,Z,C)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a surface. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices. The color is defined by the vector `C`.

`trisurf(Tri,X,Y,Z)` uses `C=Z` so color is proportional to surface height.

`trisurf(TR)` displays the triangles in a triangulation representation. It uses `C = TR.Points(:,3)` to make sure the surface color is proportional to height.

`trisurf(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trisurf(...)` returns a patch handle.

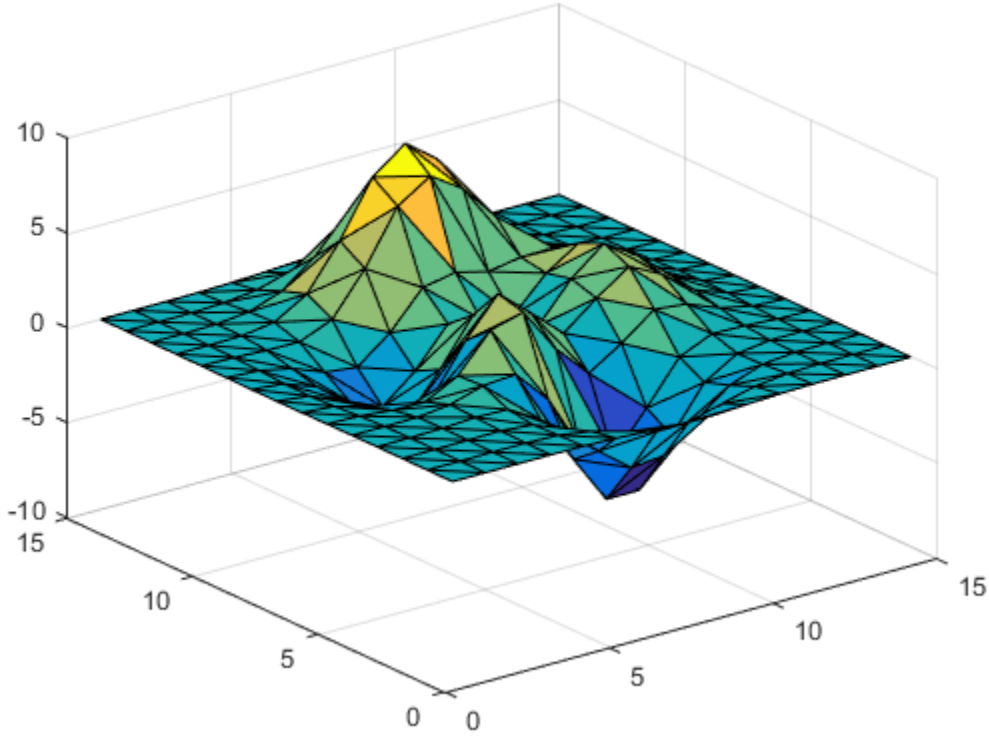
Examples

Create Triangular Surface Plot

Create vertex vectors and a face matrix, then create a triangular surface plot.

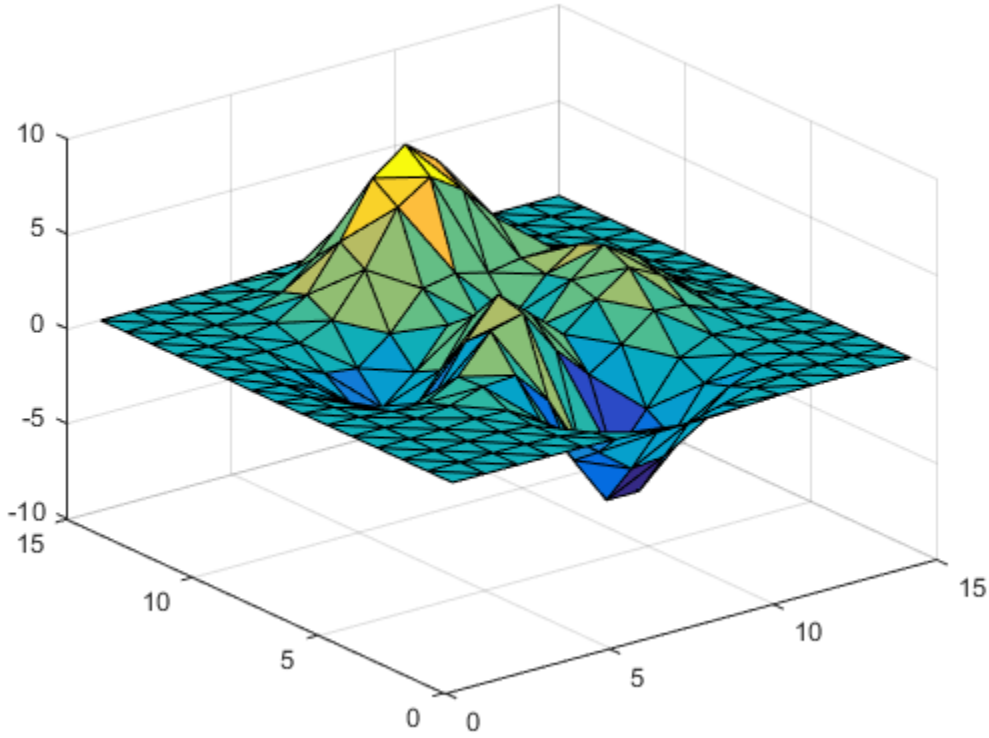
```
[x,y] = meshgrid(1:15,1:15);
```

```
tri = delaunay(x,y);
z = peaks(15);
trisurf(tri,x,y,z)
```



If the surface is in the form of a triangulation representation, you can pass it to `trisurf` alone:

```
tr = triangulation(tri,x(:),y(:),z(:));
trisurf(tr)
```



See Also

`patch` | `surf` | `tetramesh` | `triangulation` | `delaunayTriangulation` | `trimesh` | `triplot` | `delaunay`

Introduced before R2006a

triu

Upper triangular part of matrix

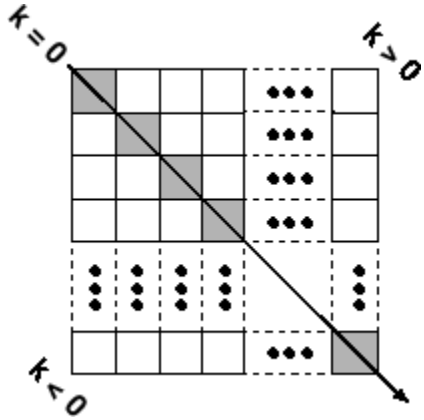
Syntax

```
U = triu(X)
U = triu(X,k)
```

Description

`U = triu(X)` returns the upper triangular part of `X`.

`U = triu(X,k)` returns the element on and above the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
triu(ones(4,4),-1)

ans =
```

```
     1     1     1     1
     1     1     1     1
     0     1     1     1
     0     0     1     1
```

See Also

`diag` | `tril`

Introduced before R2006a

true

Logical 1 (true)

Syntax

```
true
T = true(n)
T = true(sz)
T = true(sz1,...,szN)
T = true( ___ ,'like',p)
```

Description

`true` is shorthand for `logical(1)`.

`T = true(n)` is an `n`-by-`n` matrix of logical ones.

`T = true(sz)` is an array of logical ones where the size vector, `sz`, defines `size(T)`. For example, `true([2 3])` returns a 2-by-3 array of logical ones.

`T = true(sz1,...,szN)` is a `sz1`-by-`...`-by-`szN` array of logical ones where `sz1,...,szN` indicates the size of each dimension. For example, `true(2,3)` returns a 2-by-3 array of logical ones.

`T = true( ___ ,'like',p)` returns an array of logical ones of the same sparsity as the logical variable `p` using any of the previous size syntaxes.

Examples

Generate Square Matrix of Logical Ones

Use `true` to generate a 3-by-3 square matrix of logical ones.

```
A = true(3)
class(A)
```

```
A =
     1     1     1
     1     1     1
     1     1     1
ans =
logical
```

The result is of class `logical`.

Generate Array of Logical Ones with Arbitrary Dimensions

Use `true` to generate a 3-by-2-by-2 matrix of logical ones.

```
true(3,2,2)
```

```
ans(:,:,1) =
     1     1
     1     1
     1     1

ans(:,:,2) =
     1     1
     1     1
     1     1
```

Alternatively, you can use a size vector to specify the size of the matrix.

```
true([3,2,2])
```

```
ans(:,:,1) =
     1     1
     1     1
     1     1

ans(:,:,2) =
     1     1
     1     1
     1     1
```

Note that specifying multiple vector inputs returns an error.

Execute Logic Statement

`true` along with `false` can be used to execute logic statements.

Test the logical statement $\sim(A\ and\ B) \equiv (\sim A)\ or\ (\sim B)$ for A = logical true and B = logical false.

```
~(true & false) == (~true) | (~false)
```

```
ans =
     1
```

The result is logical 1 (true), since the logical statements on both sides of the equation are equivalent. This logical statement is an instance of De Morgan's Law.

Generate Logical Array of Selected Sparsity

Generate a logical array of the same sparsity as the selected array.

```
A = logical(sparse(5,3));
whos A
T = true(4,'like',A);
whos T
```

```
Name      Size            Bytes  Class      Attributes
  A         5x3                41  logical    sparse
Name      Size            Bytes  Class      Attributes
  T         4x4               184  logical    sparse
```

The output array T has the same `sparse` attribute and data-type as the specified array A.

Input Arguments

n — Size of square matrix

integer

Size of square matrix, specified as an integer. `n` sets the output array size to `n`-by-`n`. For example, `true(3)` returns a 3-by-3 array of logical ones.

- If n is `0`, then T is an empty matrix.
- If n is negative, then it is treated as `0`.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Size vector

row vector of integers

Size vector, specified as a row vector of integers. For example, `true([2 3])` returns a 2-by-3 array of logical ones.

- If the size of any dimension is `0`, then T is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, T, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN — Size inputs

comma-separated list of integers

Size inputs, specified by a comma-separated list of integers. For example, `true(2,3)` returns a 2-by-3 array of logical ones.

- If the size of any dimension is `0`, then T is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, T, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

p — Prototype

logical variable

Prototype, specified as a logical variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Output Arguments

T — Output of logical ones

scalar | vector | matrix | N-D array

Output of logical ones, returned as a scalar, vector, matrix, or N-D array.

Data Types: `logical`

More About

Tips

- `true(n)` is much faster and more memory efficient than `logical(true(n))`.
- "Class Support for Array-Creation Functions"

See Also

`false` | `logical`

Introduced before R2006a

try, catch

Execute statements and catch resulting errors

Syntax

```
try
   statements
catch exception
   statements
end
```

Description

`try` *`statements`*`, catch` *`statements`* `end` executes the statements in the `try` block and catches resulting errors in the `catch` block. This approach allows you to override the default error behavior for a set of program statements. If any statement in a `try` block generates an error, program control goes immediately to the `catch` block, which contains your error handling statements.

`exception` is an `MException` object that allows you to identify the error. The `catch` block assigns the current exception object to the variable in *`exception`*.

Both `try` and `catch` blocks can contain nested `try/catch` statements.

Examples

Supplement Error Message

Create two matrices that you cannot concatenate vertically.

```
A = rand(3);
B = ones(5);

C = [A; B];
```

```
Error using vertcat
Dimensions of matrices being concatenated are not consistent.
```

Use `try/catch` to display more information about the dimensions.

```
try
   C = [A; B];
catch ME
   if (strcmp(ME.identifier,'MATLAB:catenate:dimensionMismatch'))
      msg = ['Dimension mismatch occurred: First argument has ', ...
            num2str(size(A,2)),' columns while second has ', ...
            num2str(size(B,2)),' columns.'];
        causeException = MException('MATLAB:myCode:dimensions',msg);
        ME = addCause(ME,causeException);
   end
   rethrow(ME)
end
```

```
Error using vertcat
Dimensions of matrices being concatenated are not consistent.

Caused by:
    Dimension mismatch occurred: First argument has 3 columns while second has 5 columr
```

If matrix dimensions do not agree, MATLAB displays more information about the mismatch. Any other errors appear as usual.

Repackage Error as Warning

Catch any exception generated by calling the nonexistent function, `notaFunction`. If there is an exception, issue a warning and assign the output a value of 0.

```
try
    a = notaFunction(5,6);
catch
    warning('Problem using function.  Assigning a value of 0.');
    a = 0;
end
```

```
Warning: Problem using function.  Assigning a value of 0.
```

By itself, the call to `notaFunction` results in an error. If you use `try` and `catch`, this code catches any exception and repackages it as a warning, allowing MATLAB to continue executing subsequent commands.

Handle Different Types of Errors

Use `try/catch` to handle different types of errors in different ways.

- If the function `notaFunction` is undefined, issue a warning instead of an error and assign the output a value of `NaN`.
- If `notaFunction.m` exists, but is a script instead of a function, issue a warning instead of an error, run the script, and assign the output a value of `0`.
- If MATLAB throws an error for any other reason, rethrow the exception.

```
try
    a = notaFunction(5,6);
catch ME
    switch ME.identifier
        case 'MATLAB:UndefinedFunction'
            warning('Function is undefined.  Assigning a value of NaN.');
            a = NaN;
        case 'MATLAB:scriptNotAFunction'
            warning(['Attempting to execute script as function. '...
                'Running script and assigning output a value of 0.']);
            notaFunction;
            a = 0;
        otherwise
            rethrow(ME)
    end
end
```

```
Warning: Function is undefined.  Assigning a value of NaN.
```

More About

Tips

- You cannot use multiple `catch` blocks within a `try` block, but you can nest complete `try/catch` blocks.
- Unlike some other languages, MATLAB does not allow the use of a `finally` block within `try/catch` statements.

See Also

`MException` | `assert` | `error`

Introduced before R2006a

tscollection

Create `tscollection` object

Syntax

```
tsc = tscollection(TimeSeries)
tsc = tscollection(Time)
tsc = tscollection(...,'Parameter',Value,...)
```

Description

`tsc = tscollection(TimeSeries)` creates a `tscollection` object `tsc` with one or more `timeseries` objects already in the MATLAB workspace. The argument `TimeSeries` can be a

- Single `timeseries` object
- Cell array of `timeseries` objects

`tsc = tscollection(Time)` creates an empty `tscollection` object with the time vector `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

`tsc = tscollection(...,'Parameter',Value,...)` creates a `tscollection` object with optional parameter-value pairs you enter after specifying either a time vector or a `timeseries` object. You can specify the following parameter:

- `Name` — String that specifies the name of this `tscollection` object

Examples

The following example shows how to create a `tscollection` object.

1 Import the sample data.

```
load count.dat
```

2 Create three `timeseries` objects to store each set of data:

```
count1 = timeseries(count(:,1),1:24,'name', 'ts1');
count2 = timeseries(count(:,2),1:24,'name', 'ts2');
count3 = timeseries(count(:,3),1:24,'name', 'ts3');
```

3 Create a `tscollection` object named `tsc` and add to it two out of three time series already in the MATLAB workspace, by using the following syntax:

```
tsc = tscollection({count1 count2},'name','tsc')
```

More About

Tips

Definition: Time Series Collection

A time series collection object is a MATLAB variable that groups several time series with a common time vector. The time series that you include in the collection are called members of this collection.

Properties of Time Series Collection Objects

This table lists the properties of the `tscollection` object. You can specify the `Time`, `TimeSeries`, and `Name` properties as input arguments in the constructor.

Property	Description
`Name`	`tscollection` name as a string. This can differ from the `tscollection` name in the MATLAB workspace.
`Time`	When `TimeInfo.StartDate` is empty, values are measured relative to `0` . When `TimeInfo.StartDate` is defined, values represent date strings measured relative to the `StartDate`. The length of `Time` must be the same as the first or the last dimension of `Data` for each collection .

Property	Description
`TimeInfo`	Contains fields for contextual information about `Time`: • `Units` — Time units with any of the following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, `'nanoseconds'` • `Start` — Start time • `End` — End time (read only) • `Increment` — Interval between subsequent time values. `NaN` when times are not uniformly sampled. • `Length` — Length of the time vector (read only) • `Format` — String defining the date string display format. See `datestr`. • `StartDate` — Date string defining the reference date. See `setabstime (tscollection)`. • `UserData` — Any additional user-defined information

See Also

`addts` | `datestr` | `setabstime (tscollection)` | `timeseries`

Introduced before R2006a

tsdata.event

Construct event object for `timeseries` object

Syntax

```
e = tsdata.event(Name,Time)
e = tsdata.event(Name,Time,'Datenum')
```

Description

`e = tsdata.event(Name,Time)` creates an event object with the specified Name that occurs at the time Time. Time can either be a real value or a date string.

`e = tsdata.event(Name,Time,'Datenum')` uses `'Datenum'` to indicate that the `Time` value is a serial date number generated by the `datenum` function. The `Time` value is converted to a date string after the event is created.

More About

Tips

You add events by using the `addevent` method.

Fields of the `tsdata.event` object include the following:

- `EventData` — MATLAB array that stores any user-defined information about the event
- `Name` — String that specifies the name of the event
- `Time` — Time value when this event occurs, specified as a real number
- `Units` — Time units
- `StartDate` — A reference date, specified in MATLAB `datestr` format. `StartDate` is empty when you have a numerical (non-date-string) time vector.

Introduced before R2006a

tsearchn

N-D closest simplex search

Syntax

```
t = tsearchn(X,TRI,XI)
[t,P] = tsearchn(X,TRI,XI)
```

Description

`t = tsearchn(X,TRI,XI)` returns the indices `t` of the enclosing simplex of the Delaunay triangulation `TRI` for each point in `XI`. `X` is an `m`-by-`n` matrix, representing `m` points in N-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in N-dimensional space. `tsearchn` returns `NaN` for all points outside the convex hull of `X`. `tsearchn` requires a triangulation `TRI` of the points `X` obtained from `delaunayn`.

`[t,P] = tsearchn(X,TRI,XI)` also returns the barycentric coordinate `P` of `XI` in the simplex `TRI`. `P` is a `p`-by-`n+1` matrix. Each row of `P` is the barycentric coordinate of the corresponding point in `XI`. It is useful for interpolation.

See Also

`delaunayTriangulation`

Introduced before R2006a

type

Display contents of file

Syntax

```
type('filename')
type filename
```

Description

`type('filename')` displays the contents of the specified file in the MATLAB Command Window. Use the full path for `filename`, or use a MATLAB relative partial path.

If you do not specify a file extension and there is no `filename` file without an extension, the `type` function adds the `.m` extension by default. The `type` function checks the folders specified in the MATLAB search path, which makes it convenient for listing the contents of files on the screen. Use `type` with `more on` to see the listing one screen at a time.

`type filename` is the command form of the syntax.

Examples

`type('foo.bar')` lists the contents of the file `foo.bar`.

`type foo` lists the contents of the file `foo`. If `foo` does not exist, `type foo` lists the contents of the file `foo.m`.

See Also

`cd` | `dbtype` | `delete` | `dir` | `more` | `path` | `what` | `who`

Introduced before R2006a

typecast

Convert data types without changing underlying data

Syntax

```
Y = typecast(X, type)
```

Description

`Y = typecast(X, type)` converts a numeric value in `X` to the data type specified by `type`. Input `X` must be a full, noncomplex, numeric scalar or vector. The `type` input is a string set to one of the following: `'uint8'`, `'int8'`, `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, or `'double'`.

`typecast` is different from the MATLAB `cast` function in that it does not alter the input data. `typecast` always returns the same number of bytes in the output `Y` as were in the input `X`. For example, casting the 16-bit integer 1000 to `uint8` with `typecast` returns the full 16 bits in two 8-bit segments (3 and 232) thus keeping its original value (3*256 + 232 = 1000). The `cast` function, on the other hand, truncates the input value to 255.

The output of `typecast` can be formatted differently depending on what system you use it on. Some computer systems store data starting with its most significant byte (an ordering called *big-endian*), while others start with the least significant byte (called *little-endian*).

Note MATLAB issues an error if `X` contains fewer values than are needed to make an output value.

Examples

Example 1

This example converts between data types of the same size:

```
typecast(uint8(255), 'int8')
ans =
   -1

typecast(int16(-1), 'uint16')
ans =
  65535
```

Example 2

Set X to a 1-by-3 vector of 32-bit integers, then cast it to an 8-bit integer type:

```
X = uint32([1 255 256])
X =
           1          255          256
```

Running this on a little-endian system produces the following results. Each 32-bit value is divided up into four 8-bit segments:

```
Y = typecast(X, 'uint8')
Y =
   1   0   0   0  255   0   0   0   0   1   0   0
```

The third element of `X`, 256, exceeds the 8 bits that it is being converted to in `Y`(9) and thus overflows to `Y`(10):

```
Y(9:12)
ans =
    0    1    0    0
```

Note that `length(Y)` is equal to `4.*length(X)`. Also note the difference between the output of `typecast` versus that of `cast`:

```
Z = cast(X, 'uint8')
Z =
    1  255  255
```

Example 3

This example casts a smaller data type (`uint8`) into a larger one (`uint16`). Displaying the numbers in hexadecimal format makes it easier to see just how the data is being rearranged:

```
format hex
```

```
X = uint8([44 55 66 77])
X =
   2c   37   42   4d
```

The first `typecast` is done on a big-endian system. The four 8-bit segments of the input data are combined to produce two 16-bit segments:

```
Y = typecast(X, 'uint16')
Y =
   2c37   424d
```

The second is done on a little-endian system. Note the difference in byte ordering:

```
Y = typecast(X, 'uint16')
Y =
   372c   4d42
```

You can format the little-endian output into big-endian (and vice versa) using the `swapbytes` function:

```
Y = swapbytes(typecast(X, 'uint16'))
Y =
   2c37   424d
```

Example 4

This example attempts to make a 32-bit value from a vector of three 8-bit values. MATLAB issues an error because there are an insufficient number of bytes in the input:

```
format hex
typecast(uint8([120 86 52]), 'uint32')

Error using typecast
Too few input values to make output type.
```

Repeat the example, but with a vector of four 8-bit values, and it returns the expected answer:

```
typecast(uint8([120 86 52 18]), 'uint32')
ans =
   12345678
```

See Also

`cast` | `class` | `swapbytes`

Introduced before R2006a

tzoffset

Time zone offset from UTC

Syntax

```
dt = tzoffset(t)
[dt,dst] = tzoffset(t)
```

Description

`dt = tzoffset(t)` returns an array of durations equal to the time zone offset from UTC of each datetime value in `t`. That is, `dt` is the amount of time that each datetime in `t` differs from UTC. For datetimes that occur during Daylight Saving Time (DST), `dt` includes the time shift for DST. The offset for unzoned datetime values is not defined.

`[dt,dst] = tzoffset(t)` additionally returns the time shift for Daylight Saving Time for each datetime in `t`.

Examples

Find Time Zone Offset

Create a `datetime` array in the time zone for New York City.

```
t1 = datetime('today','TimeZone','America/New_York');
t = dateshift(t1,'end','month',[1:3:9])
```

```
t =

   31-Mar-2015   30-Jun-2015   30-Sep-2015
```

Find the time zone offset from UTC and the time shift for Daylight Saving Time for each datetime value.

```
[dt,dst] = tzoffset(t)
```

```
dt =

   -4:00   -4:00   -4:00

dst =

   01:00   01:00   01:00
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

Output Arguments

`dt` — Time zone offset from UTC

scalar | vector | matrix | multidimensional array

Time zone offset from UTC, returned as a scalar, vector, matrix, or multidimensional `duration` array. `dt` is the same size as `t`.

`dst` — Time shift for Daylight Saving Time

scalar | vector | matrix | multidimensional array

Time shift for Daylight Saving Time, returned as a scalar, vector, matrix, or multidimensional `duration` array. `dt` is the same size as `t`. A value of `01:00` indicates that the corresponding datetime in `t` occurs during Daylight Saving Time in a location that observes it. For locations that do not observe Daylight Saving Time, the elements of the `dst` array are all `00:00`.

See Also

`isdst`

Introduced in R2014b

uibuttongroup

Create button group to manage radio buttons and toggle buttons

Syntax

```
b = uibuttongroup
b = uibuttongroup(Name,Value,...)
b = uibuttongroup(parent)
b = uibuttongroup(parent,Name,Value,...)
```

Description

`b = uibuttongroup` creates a uibuttongroup in an existing figure and returns the uibuttongroup object, `b`. If there is no figure available, then MATLAB creates a new figure to serve as the parent.

`b = uibuttongroup(Name,Value,...)` creates a uibuttongroup and specifies one or more uibuttongroup property names and corresponding values. Use this syntax to override the default uibuttongroup properties.

`b = uibuttongroup(parent)` creates a uibuttongroup and designates a specific parent object. The `parent` argument can be a figure, uipanel, uitab, or another uibuttongroup object.

`b = uibuttongroup(parent,Name,Value,...)` creates a uibuttongroup with a specific parent and one or more uibuttongroup properties.

Uibuttongroups are containers that group UI components together and manage exclusive selection of radio buttons and toggle buttons. These are some of the important characteristics of the uibuttongroups and some recommended best practices:

- Uibuttongroups can contain any UI component type (except anActiveX control), but it only manages the selection of radio buttons and toggle buttons.
- Define a `SelectionChangedFcn` callback function for the uibuttongroup to make your program respond when the user selects a radio button or toggle button that is inside the button group. Do not define callbacks for in the individual buttons.

- Query the `SelectedObject` property of the uibuttongroup to determine which radio button or toggle button is currently selected. You can execute this query anywhere in your code that can access the uibuttongroup object.

Examples

This example creates a uibuttongroup with three radio buttons. It manages the radio buttons with the SelectionChangedFcn callback, `bselection`.

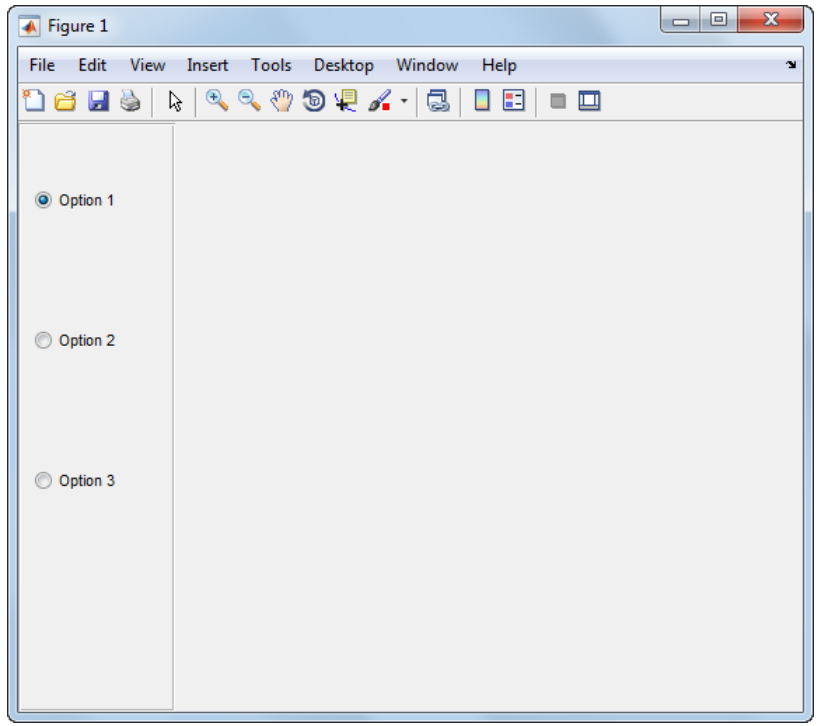
When the user selects a new radio button, the `bselection` function displays the previous and current selection.

Copy and paste this code into the Editor and run it to see how it works.

```
function myui
bg = uibuttongroup('Visible','off',...
                   'Position',[0 0 .2 1],...
                   'SelectionChangedFcn',@bselection);

% Create three radio buttons in the button group.
r1 = uicontrol(bg,'Style',...
                   'radiobutton',...
                   'String','Option 1',...
                   'Position',[10 350 100 30],...
                   'HandleVisibility','off');

r2 = uicontrol(bg,'Style','radiobutton',...
                   'String','Option 2',...
                   'Position',[10 250 100 30],...
                   'HandleVisibility','off');

r3 = uicontrol(bg,'Style','radiobutton',...
                   'String','Option 3',...
                   'Position',[10 150 100 30],...
                   'HandleVisibility','off');

% Make the uibuttongroup visible after creating child objects.
bg.Visible = 'on';

    function bselection(source,callbackdata)
       display(['Previous: ' callbackdata.OldValue.String]);
       display(['Current: ' callbackdata.NewValue.String]);
       display('------------------');
    end
end
```

The `bselection` function displays the `OldValue` and `NewValue` properties of the `callbackdata` object. MATLAB passes the `callbackdata` object to this callback function when you specify the SelectionChangedFcn property as a function handle

(or a cell array containing a function handle). See `SelectionChangedFcn` for more information.



More About

Tips

If you set the Visible property of a `uibuttongroup` object to `'off'`, then any child objects it contains (buttons, other button groups, etc.) become invisible along with the parent uibuttongroup. However, the Visible *property value* of each child object remains unaffected.

- "Access Property Values"

See Also

Uibuttongroup Properties | `uicontrol` | `uipanel`

Introduced before R2006a

Uibuttongroup Properties

Control appearance and behavior of button group

Uibuttongroups are containers for grouping together and managing the exclusive selection of radio buttons and toggle buttons. The `uibuttongroup` function creates a button group and sets any required properties. By changing uibuttongroup property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
b = uibuttongroup;
bgcolor = b.BackgroundColor;
b.BackgroundColor = [.5 .5 .5];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uibuttongroup visibility

`'on'` (default) | `'off'`

Uibuttongroup visibility, specified as `'on'` or `'off'`. The Visible property determines whether the uibuttongroup displays on the screen. If the Visible property of a uibuttongroup is set to `'off'`, the entire uibuttongroup is invisible, but you can still specify and access its properties.

Changing the size of an invisible uibuttongroup triggers the SizeChangedFcn callback when the uibuttongroup becomes visible.

Note Changing the Visible property of a uibuttongroup does *not* change the Visible property of its child components even though hiding the uibuttongroup prevents its children from displaying.

`BackgroundColor` — Uibuttongroup background color

`[.94 .94 .94]` (default) | RGB triplet | short name | long name

Uibuttongroup background color, specified as an RGB triplet, short name, or long name. The color you specify fills the area bounded by the uibuttongroup.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Note: Some operating systems override the `BackgroundColor` value.

Data Types: `double` | `char`

`ForegroundColor` — Uibuttongroup title color

`[0 0 0]` (default) | RGB triplet | short name | long name

Uibuttongroup title color, specified as an RGB triplet, short name, or long name.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`

Long Name	Short Name	RGB Triplet
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0 0 1]

Example: 'b'

Example: 'blue'

BorderType — Border of the uibuttongroup

'etchedin' (default) | 'etchedout' | 'beveledin' | 'beveledout' | 'line' | 'none'

Border of the uibuttongroup area, specified as 'etchedin', 'none', 'etchedout', 'beveledin', 'beveledout', or 'line'.

- For a 3-D look, use etched or beveled borders.

  Use the HighlightColor and ShadowColor properties to specify the color of 3-D borders.
- For a 2-D look, use a line border.

  Use the ForegroundColor property to specify the line border color.

BorderWidth — Width of uibuttongroup border

1 (default) | positive integer value

Width of the uibuttongroup border, specified as a positive integer value. The unit of measurement is pixels. Etched and beveled borders wider than three pixels might not appear correctly at the corners.

HighlightColor — Uibuttongroup 3-D frame highlight color

RGB triplet | short name | long name

Uibuttongroup 3-D frame highlight color, specified as an RGB triplet, short name, or long name.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

[0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

ShadowColor — Uibuttongroup border shadow color

RGB triplet | short name | long name

Uibuttongroup border shadow color, specified as an RGB triplet, short name, or long name.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

`Clipping` — Clipping of child components to uibuttongroup

`'on'` (default) | `'off'`

Note: The behavior of the Clipping property has changed. It no longer has any effect on uibuttongroups. Child objects are now clipped to the uibuttongroup boundary regardless of the value of this property. This property might be removed in a future release.

Location and Size

`Position` — Location and size of uibuttongroup relative to parent

`[left bottom width height]`

Location and size of the uibuttongroup relative to the parent, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the uibuttongroup
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the uibuttongroup
`width`	Distance between the right and left outer edges of the uibuttongroup
`height`	Distance between the top and bottom outer edges of the uibuttongroup

All measurements are in units specified by the Units property.

Note: If the parent of the uibuttongroup is a figure, then the Position values are relative to the figure's *drawable area*. The drawable area of a figure is the area inside the window borders, excluding the menu bar and tool bar.

Example: Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the Position vector. For example, this code sets the width of the Uibuttongroup to `0.5`:

```
b = uibuttongroup;
b.Position(3) = 0.5;
b.Position

ans =

      0   0   0.5000  1.0000
```

Units — Units of measurement

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'pixels'` | `'points'` | `'characters'`

Units of measurement, specified as `'normalized'`, `'inches'`, `'centimeters'`, `'pixels'`, `'points'`, or `'characters'`.

MATLAB uses these units to interpret the location and size values of the Position property:

- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- The size of a uibuttongroup specified in pixel units depends on the system display settings and resolution.
- Characters units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the Units property before the Position property, then MATLAB sets Position using the units you specified.
- If you specify the Units property after the Position property, MATLAB sets the position using the default Units. Then, MATLAB converts the Position values to the equivalent values in the units you specified.

`SizeChangedFcn` — Uibuttongroup resize callback function

`''` (default) | function handle | cell array | string

Uibuttongroup resize callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Define this callback function to control the UI layout when the size of the uibuttongroup changes.

The SizeChangedFcn callback executes under these circumstances:

- The uibuttongroup becomes visible for the first time.
- The uibuttongroup is visible while its drawable area changes. The drawable area is the area inside the outer bounds of the uibuttongroup.
- The uibuttongroup becomes visible for the first time after its drawable area changes. This situation occurs when the drawable area changes while the uibuttongroup is invisible, and then it becomes visible later.

These are some of the important characteristics of the SizeChangedFcn callback and some recommended best practices:

- Consider delaying the display of the figure until after all the variables that the uibuttongroup's SizeChangedFcn uses are defined. This practice can prevent the uibuttongroup's SizeChangedFcn callback from returning an error. To delay the display of the figure, set its Visible property to `'off'`. Then, set the Visible property to `'on'` after you define the variables that your SizeChangedFcn callback uses.
- Use the `gcbo` function in your SizeChangedFcn code to get the uibuttongroup object that the user is resizing. See the description of the figure `SizeChangedFcn` for an example that uses `gcbo`.
- All visible Uibuttongroups that are specified in normalized units resize before the parent figure does. All visible, nested uibuttongroups resize from inside-out.

Tip As an easy alternative to specifying a SizeChangedFcn callback, you can set the Units property of all the objects you put inside the uibuttongroup to `'normalized'`. Doing so makes those components scale proportionally with the uibuttongroup.

See “Managing the Layout in Resizable UIs” for more information on controlling the resize behavior of programmatic UIs.

See Resize Behavior for more information on the resize behavior of GUIDE UIs.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`ResizeFcn` — Uibuttongroup resize callback function

`''` (default) | function handle | cell array | string

Callback function that executes when the uibuttongroup size changes, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Note: Use of the ResizeFcn property is not recommended. It might be removed in a future release. Use SizeChangedFcn instead.

Data Types: `function_handle | cell | char`

Font Style

`FontName` — Font for displaying uibuttongroup title

string

Font for displaying the uibuttongroup title, specified as a string. To display and print properly, the font name must refer to a font that the user’s system supports. The default font is system dependent.

To use a fixed-width font, set the FontName property to the string, `'FixedWidth'`. This setting instructs MATLAB to use the value of the root FixedWidthFontName property, which you can set in the `startup.m` file.

See “Startup Options in MATLAB Startup File” for more information about the `startup.m` file.

Example: `'Arial'`

`FontSize` — Font size for uibuttongroup title

positive number

Font size for the uibuttongroup title, specified as a positive number. The FontUnits property specifies the units. The default size is system-dependent.

Example: `12`

Example: `12.5`

Data Types: `double`

`FontUnits` — Units of font size for uibuttongroup title

`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Units of font size for the uibuttongroup title, specified as `'points'`, `'inches'`, `'centimeters'`, `'normalized'`, or `'pixels'`.

If you set this property to `'normalized'`, then MATLAB interprets the font size as a fraction of the uibuttongroup height. When you resize the uibuttongroup, MATLAB scales the displayed font to maintain that fraction.

The other FontUnits options (`pixels`, `inches`, `centimeters`, and `points`) are absolute units. 1 point = 1/72 inch.

`FontWeight` — Font weight for the uibuttongroup title

`'normal'` (default) | `'bold'`

Font weight of uibuttongroup title, specified as a value from the following table.

FontWeight Value	Description
`'normal'`	Normal font weight
`'bold'`	Heavy font weight

Not all fonts support all font weights. Therefore, if you specify an unsupported value for the `FontWeight` property, the result might appear the same as the default.

Note: The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

`FontAngle` — Character slant of the uibuttongroup title

`'normal'` (default) | `'italic'`

Character slant of uibuttongroup title, specified as `'normal'` or `'italic'`. MATLAB uses this property to select a font from those available on your system. Setting this property to `'italic'` selects a slanted version of the font, if it is available on your system.

Note: The `'oblique'` value has been removed. Use `'italic'` instead.

Text

`Title` — Uibuttongroup title

string

Uibuttongroup title, specified as a string.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the uibuttongroup title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as `Multiples of` π.

Example: `'Options'`

Example: `['Multiples of ' char(960)]`

`TitlePosition` — Location of uibuttongroup title

`'lefttop'` (default) | `'centertop'` | `'righttop'` | `'leftbottom'` | `'centerbottom'` | `'righttbottom'`

Location of the uibuttongroup title, specified as `'lefttop'`, `'centertop'`, `'righttop'`, `'leftbottom'`, `'centerbottom'`, or `'righttbottom'`.

Interactive Control

`SelectionChangedFcn` — Callback that executes when user selects a different button

`''` (default) | function handle | cell array | string

Callback that executes when the user selects a different button, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes when the user selects a different button within the uibuttongroup.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in this table. You can access these properties inside the callback function using dot notation.

Property	Description
`OldValue`	Previously selected button, or `[]` if none was selected
`NewValue`	Currently selected button
`Source`	The parent uibuttongroup object
`EventName`	`'SelectionChanged'`

Define a uibuttongroup SelectionChangedFcn callback to make your program respond when the user selects different buttons within the uibuttongroup. Do not code the response in the individual uicontrol Callback functions.

If you want another component to respond to the selection, then that component's callback function can access the selected radio button or toggle button from the uibuttongroup's SelectedObject property.

Example: `@myfun`

Example: `{@myfun,x}`

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The ButtonDownFcn callback is a function that executes when the user clicks a mouse button on the uibuttongroup.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle` | `cell` | `char`

`UIContextMenu` — Uibuttongroup context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Uibuttongroup context menu, specified as a uicontextmenu object. Use this property to display a context menu when the user right-clicks on the uibuttongroup. Create the context menu using the `uicontextmenu` function.

`Selected` — Selection status of uibuttongroup

`'off'` (default) | `'on'`

Note: The behavior of the Selected property changed in R2014b, and it is not recommended. It no longer has any effect on uibuttongroups. This property might be removed in a future release.

`SelectionHighlight` — Ability to highlight selection handles

`'on'` (default) | `'off'`

Note: The behavior of the SelectionHighlight property changed in R2014b, and it is not recommended. It no longer has any effect on uibuttongroups. This property might be removed in a future release.

Callback Execution Control

`Interruptible` — Callback interruption

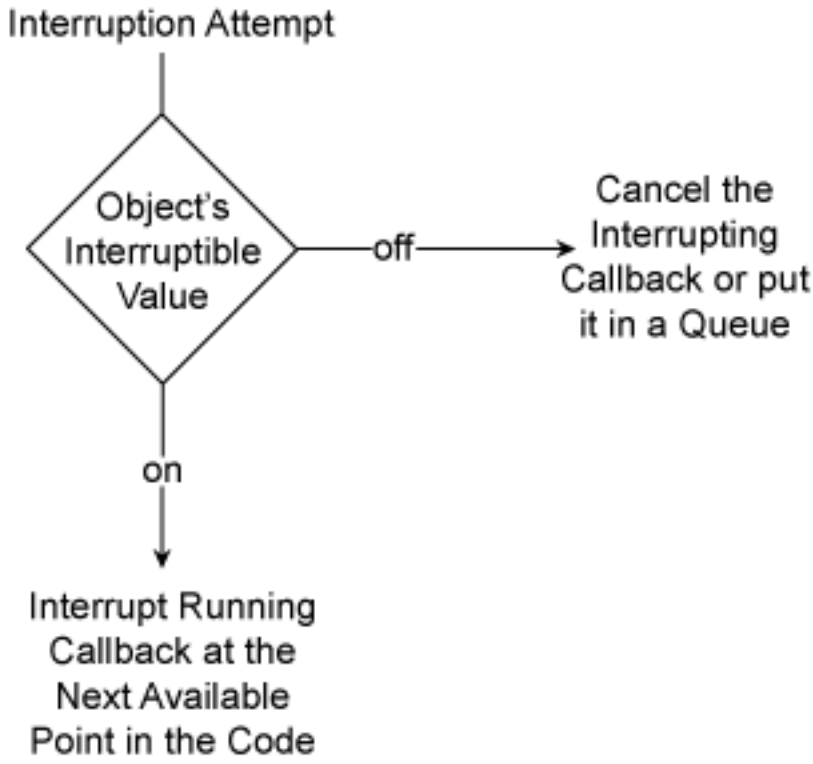
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uibuttongroup callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See "Interrupt Callback Execution" for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See "Interrupt Callback Execution" for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

Ability to become current object, specified as `'on'` or `'off'`:

- Setting the value to `'on'` allows the uibuttongroup to become the current object when the user clicks on it. A value of `'on'` also allows the figure CurrentObject property and the `gco` function to report the uibuttongroup as the current object.
- Setting the value to `'off'` sets the figure CurrentObject property to an empty `GraphicsPlaceholder` array when the user clicks on the uibuttongroup.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uibuttongroup

`'off'` (default) | `'on'`

Deletion status of uibuttongroup, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uibuttongroup begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uibuttongroup no longer exists.

Check the value of the BeingDeleted property to verify that the uibuttongroup is not about to be deleted before querying or modifying it.

`CreateFcn` — Uibuttongroup creation function

function handle | cell array | string

Uibuttongroup creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uibuttongroup. MATLAB initializes all uibuttongroup property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uibuttongroup that is being created.

Setting the CreateFcn property on an existing uibuttongroup has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uibuttongroup object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uibuttongroup deletion function

function handle | cell array | string

Uibuttongroup deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uibuttongroup (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the uibuttongroup. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uibuttongroup that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object

`'uibuttongroup'`

Type of graphics object, returned as `'uibuttongroup'`.

`Tag` — Uibuttongroup identifier

`''` (default) | string

Uibuttongroup identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uibuttongroup. When you need access to the uibuttongroup elsewhere in your code, you can use the `findobj` function to search for the uibuttongroup based on the Tag value.

Example: `'buttongroup1'`

Data Types: `char`

`UserData` — Data to associate with the uibuttongroup object

empty array (default) | array

Data to associate with the uibuttongroup object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

`SelectedObject` — Currently selected radio button or toggle button

uicontrol object | `[]`

Currently selected radio button or toggle button, specified as a uicontrol object.

Use this property to determine the currently selected button within a uibuttongroup. You can also use this property to set a default button selection. If you want no selection, then set this property to `[]`.

The default value of the SelectedObject property is the first uicontrol radio button or toggle button that you add to the uibuttongroup.

Note: The SelectionChangedFcn callback does not execute when you set the SelectedObject property programmatically..

Parent/Child

`Parent` — Uibuttongroup parent

figure | uipanel | uibuttongroup | uitab

Uibuttongroup parent, specified as a figure, uipanel, uibuttongroup, or uitab. You can move a uibuttongroup to a different figure, uipanel, uibuttongroup, or uitab by setting this property to the handle of the target figure, uipanel, uibuttongroup, or uitab.

`Children` — Children of uibuttongroup

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Children of uibuttongroup, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. Although a uibuttongroup manages only the user selection of radio buttons and toggle buttons, its children can be axes, uipanels, uibuttongroups, and any style of uicontrol.

You cannot add or remove children using the `Children` property of the uibuttongroup. Use this property to view the list of children or to reorder the children. The order of the

children in this array reflects the front-to-back order (stacking order) of the components on the screen.

To add a child to this list, set the `Parent` property of the child component to be the uibuttongroup object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of uibuttongroup handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uibuttongroup handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uibuttongroup handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uibuttongroup handle is always visible.
`'callback'`	The uibuttongroup handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uibuttongroup at the command-line, but allows callback functions to access it.
`'off'`	The uibuttongroup handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

Note: Do not try to access radio buttons and toggle buttons that are managed by a `uibuttongroup` outside of the button group. Set the HandleVisibility of those radio buttons and toggle buttons to `'off'` to prevent accidental access.

See Also

`uibuttongroup` | `uicontrol`

More About

- "Access Property Values"
- "Default Property Values"

uicontextmenu

Create context menu

A uicontextmenu is a context menu that appears when the user right-clicks on a graphics object or UI component. The `uicontextmenu` function creates a context menu and sets any required properties.

Syntax

```
c = uicontextmenu
c = uicontextmenu(Name,Value)

c = uicontextmenu(parent)
c = uicontextmenu(parent,Name,Value)
```

Description

`c = uicontextmenu` creates a uicontextmenu in the current figure and returns the uicontextmenu object as c. If no figure is available, then MATLAB creates a new figure to serve as the parent.

`c = uicontextmenu(Name,Value)` specifies one or more uicontextmenu property names and corresponding values. Use this syntax to override the default uicontextmenu property values.

`c = uicontextmenu(parent)` creates a uicontextmenu and specifies the parent figure.

`c = uicontextmenu(parent,Name,Value)` specifies a parent figure and one or more uicontextmenu properties.

Examples

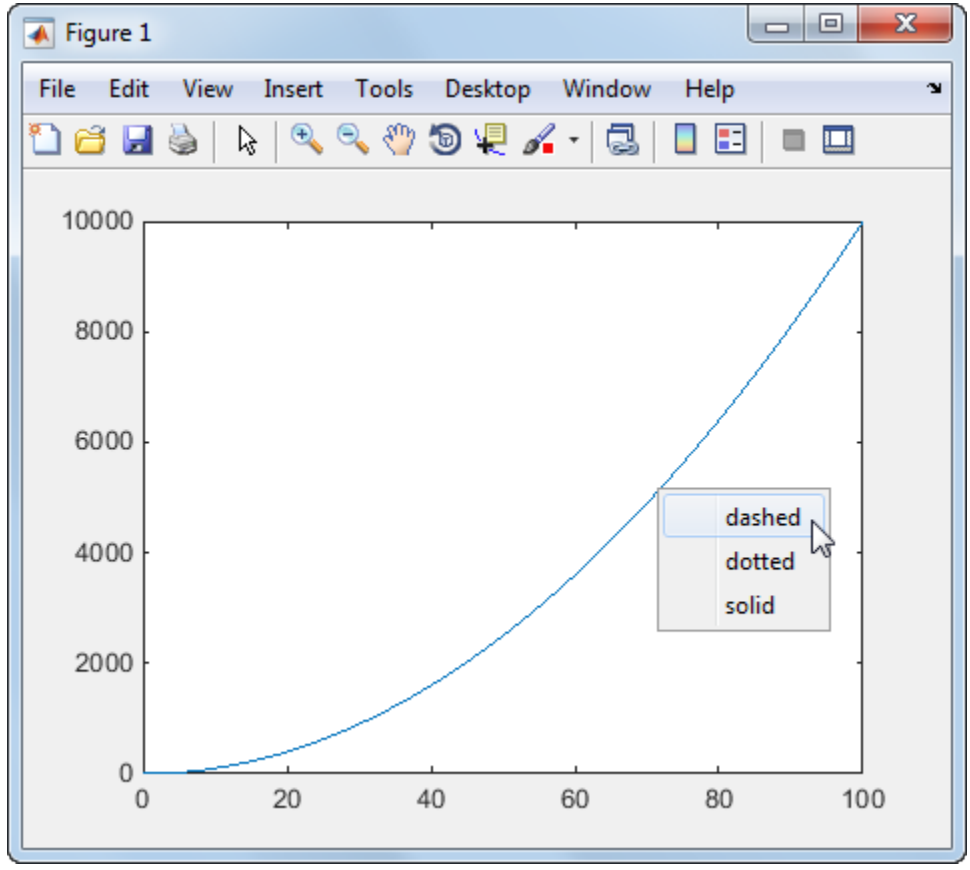
Context Menu Attached to a Plot Line

Specify a value for the line object's UIContextMenu property to attach a uicontextmenu to that line. The context menu becomes visible when the user right-clicks on the line. For

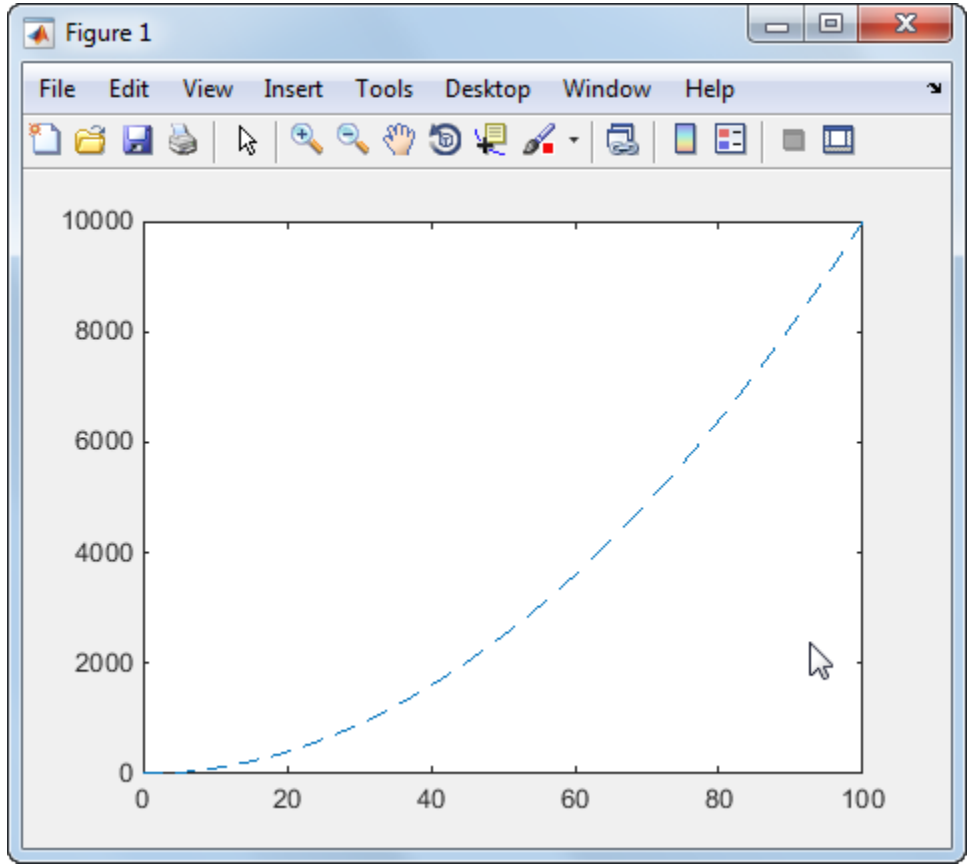
instance, create a program file called `myprogram.m` that creates a plot and attaches a uicontextmenu to the plot line:

```
function myprogram

    f = figure('WindowStyle','normal');
    ax = axes;
    x = 0:100;
    y = x.^2;

    plotline = plot(x,y);
    c = uicontextmenu;

    % Set c to be the plot line's UIContextMenu
    plotline.UIContextMenu = c;

    % Create menu items for the uicontextmenu
    m1 = uimenu(c,'Label','dashed','Callback',@setlinestyle);
    m2 = uimenu(c,'Label','dotted','Callback',@setlinestyle);
    m3 = uimenu(c,'Label','solid','Callback',@setlinestyle);

        function setlinestyle(source,callbackdata)
            switch source.Label
                case 'dashed'
                    plotline.LineStyle = '--';
                case 'dotted'
                    plotline.LineStyle = ':';
                case 'solid'
                    plotline.LineStyle = '-';
            end
        end
end
```

The context menu appears when the user right-clicks the plot line.



Selecting an item from the context menu changes the line style.



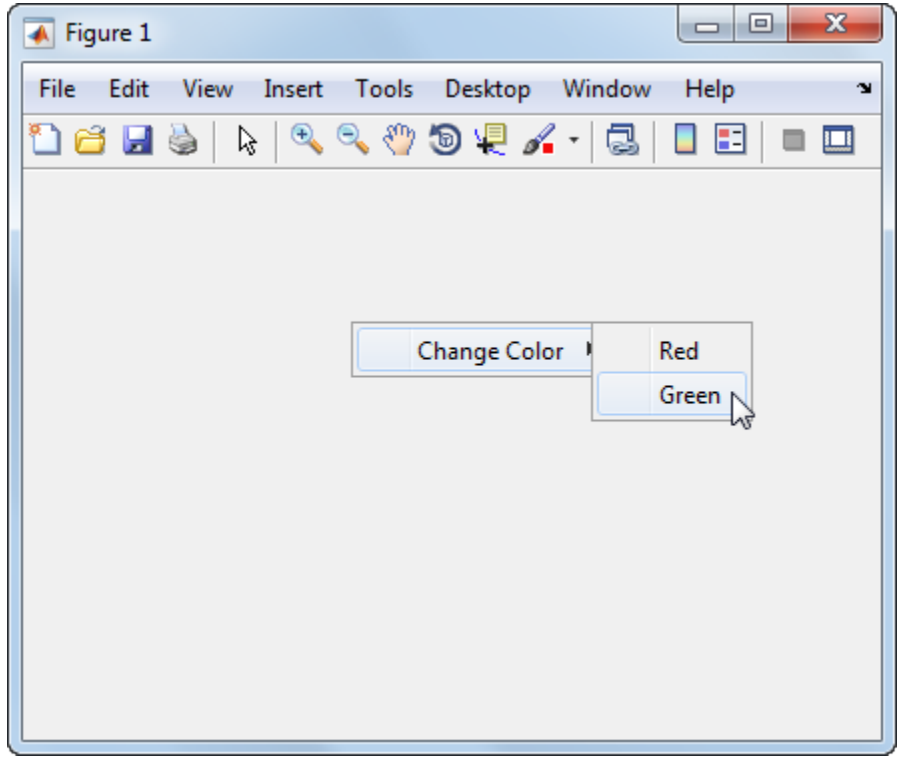
Context Menu with Submenus

Specify the Parent property value of any uimenu to make it into a submenu. For instance, create a program file called `myprogram2` that creates a uicontextmenu containing one top-level menu and two submenu items:

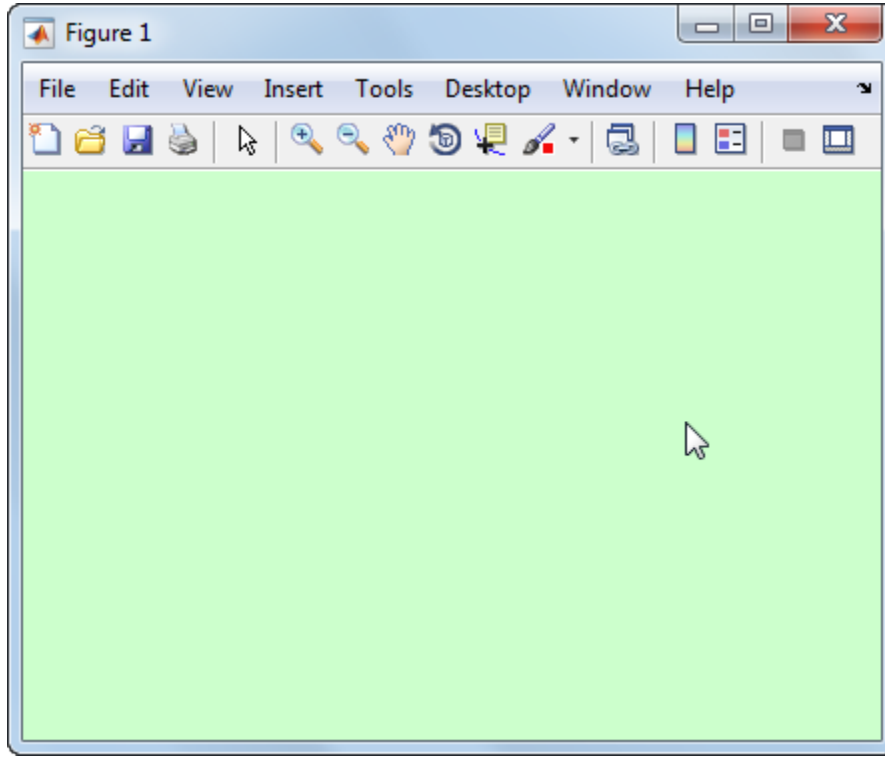
```
function myprogram2
    f = figure('WindowStyle','normal');
    c = uicontextmenu(f);

    % Assign the uicontextmenu to the figure
    f.UIContextMenu = c;
```

```
    % Create top-level menu item
    topmenu = uimenu('Parent',c,'Label','Change Color');

    % Create submenu items
    m1 = uimenu('Parent',topmenu,'Label','Red','Callback',@changecolor);
    m2 = uimenu('Parent',topmenu,'Label','Green','Callback',@changecolor);

    function changecolor(source,callbackdata)
        switch source.Label
            case 'Red'
                f.Color = [1.0 0.80 0.80];
            case 'Green'
                f.Color = [0.80 1.0 0.80];
        end
    end
end
```

The resulting context menu appears when the user right-clicks the mouse inside the figure.



Selecting a color from the context menu changes the color of the window.



Input Arguments

`parent` — Parent figure

figure

Parent figure, specified as a figure object.

Name-Value Pair Arguments

The properties listed here are only a subset, for a complete list see Uicontextmenu Properties.

Example: `'Callback',@myfunction` specifies `myfunction` to be the function that executes when the user interacts with the context menu.

`'Callback'` — Callback function that executes when user selects the uicontextmenu

`''` (default) | function handle | cell array | string

Callback function that executes when user selects the uicontextmenu, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle` | `cell` | `char`

`'Children'` — Children of uicontextmenu

empty `GraphicsPlaceholder` array (default) | 1-D array of uimenu objects

Children of uicontextmenu, returned as an empty `GraphicsPlaceholder` or a 1-D array of uimenu objects. The children of uicontextmenus are other uimenus that function as submenus.

You cannot add or remove children using the `Children` property of the uicontextmenu. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the displayed menu items.

To add a child to this list, set the `Parent` property of the child component to be the uicontextmenu object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

More About

Tips

The context menu becomes accessible in the UI when both of these requirements are met:

- You create at least one uimenu and set its parent to be the uicontextmenu. Here is an example:

  ```
  c = uicontextmenu;
  m = uimenu('Parent',c,'Label','Disable');
  ```

- You assign the uicontextmenu to be another component's UIContextMenu property value. Here is an example:

  ```
  c = uicontextmenu;
  b = uicontrol('UIContextMenu',c);
  ```

  `b` must be a descendant of the same figure as `c`.

See Also

`uibuttongroup` | Uicontextmenu Properties | `uicontrol` | `uimenu` | `uipanel`

Introduced before R2006a

Uicontextmenu Properties

Control appearance and behavior of context menu

A uicontextmenu is a context menu that appears when the user right-clicks on a graphics object or UI component. The `uicontextmenu` function creates a context menu and sets any required properties. By changing uicontextmenu property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
c = uicontextmenu;
pos = c.Position;
c.Position = [20 20];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uicontextmenu visibility

`'off'` (default) | `'on'`

Uicontextmenu visibility, specified as `'on'` or `'off'`. You can use the Visible property in one of two ways:

- Query the value to find out if the context menu is currently posted. The context menu is posted while the Visible property value is set to `'on'`.
- Set the Visible property to `'on'` to make the context menu to appear. Set this property to `'off'` to make the context menu to disappear.

Location and Size

`Position` — Uicontextmenu location

`[left bottom]`

Uicontextmenu location, specified as the vector, `[left bottom]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the figure to the left edge of the uicontextmenu
`bottom`	Distance from the inner bottom edge of the figure to the top edge of the uicontextmenu

The Position property defines the location of a context menu when its Visible property is set to `'on'`.

Data Types: `double`

Interactive Control

`Callback` — Callback function that executes when user selects the uicontextmenu

`''` (default) | function handle | cell array | string

Callback function that executes when user selects the uicontextmenu, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Callback Execution Control

`Interruptible` — Callback interruption

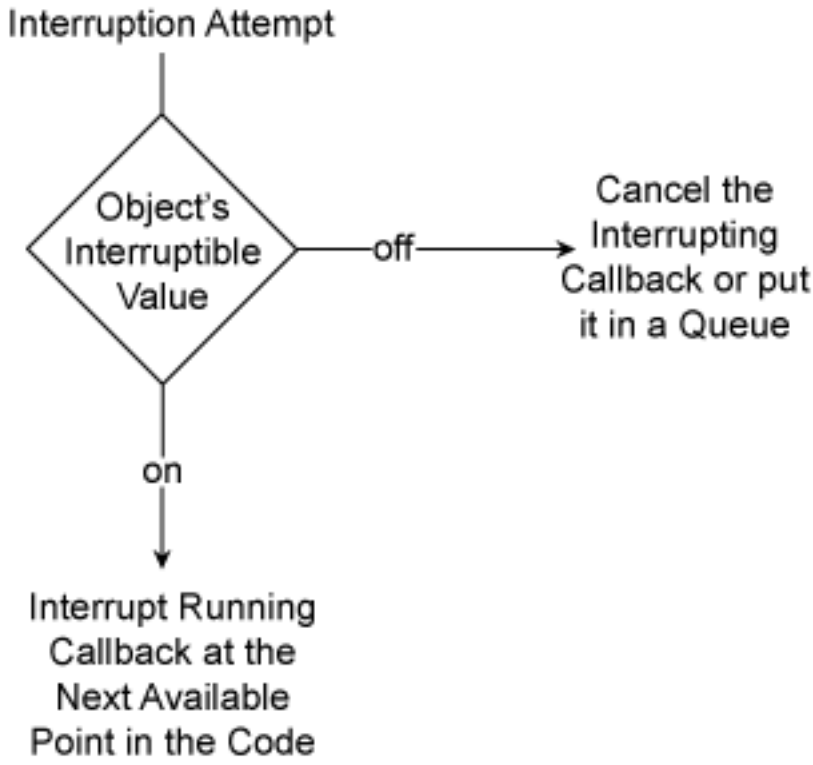
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uicontextmenu callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uicontextmenu

`'off'` (default) | `'on'`

Deletion status of uicontextmenu, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uicontextmenu begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uicontextmenu no longer exists.

Check the value of the BeingDeleted property to verify that the uicontextmenu is not about to be deleted before querying or modifying it.

`CreateFcn` — Uicontextmenu creation function

function handle | cell array | string

Uicontextmenu creation function, specified as one of these values:

- Function handle

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uicontextmenu. MATLAB initializes all uicontextmenu property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uicontextmenu that is being created.

Setting the CreateFcn property on an existing uicontextmenu has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uicontextmenu object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uicontextmenu deletion function

function handle | cell array | string

Uicontextmenu deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uicontextmenu (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the uicontextmenu. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uicontextmenu that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object

`'uicontextmenu'`

Type of graphics object, returned as `'uicontextmenu'`.

`Tag` — Uicontextmenu identifier

`''` (default) | string

Uicontextmenu identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uicontextmenu. When you need access to the uicontextmenu elsewhere in your code, you can use the `findobj` function to search for the uicontextmenu based on the Tag value.

Example: `'contextmenu1'`

Data Types: `char`

`UserData` — Data to associate with the uicontextmenu object

empty array (default) | array

Data to associate with the uicontextmenu object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uicontextmenu parent

figure

Uicontextmenu parent, specified as a figure. You can move a uicontextmenu to a different figure by setting this property to the handle of the target figure.

`Children` — Children of uicontextmenu

empty `GraphicsPlaceholder` array (default) | 1-D array of uimenu objects

Children of uicontextmenu, returned as an empty `GraphicsPlaceholder` or a 1-D array of uimenu objects. The children of uicontextmenus are other uimenus that function as submenus.

You cannot add or remove children using the `Children` property of the uicontextmenu. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the displayed menu items.

To add a child to this list, set the `Parent` property of the child component to be the uicontextmenu object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of Uicontextmenu handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uicontextmenu handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uicontextmenu handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's

handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uicontextmenu handle is always visible.
`'callback'`	The uicontextmenu handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uicontextmenu at the command-line, but allows callback functions to access it.
`'off'`	The uicontextmenu handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`uicontextmenu`

More About

- “Access Property Values”
- “Default Property Values”

uicontrol

Create user interface control object

Syntax

```
c = uicontrol
c = uicontrol(Name,Value,...)
c = uicontrol(parent)
c = uicontrol(parent,Name,Value,...)
uicontrol(c)
```

Description

`c = uicontrol` creates a uicontrol (push button) in the current figure and returns the uicontrol object, `c`. If there is no figure available, then MATLAB creates a new figure to serve as the parent.

`c = uicontrol(Name,Value,...)` creates a uicontrol and specifies one or more uicontrol property names and corresponding values. Use this syntax to override the default uicontrol properties. The default uicontrol style is `'pushbutton'`.

`c = uicontrol(parent)` creates a uicontrol and designates a specific parent object. The `parent` argument can be a figure, uipanel, uibuttongroup, or uitab object.

`c = uicontrol(parent,Name,Value,...)` creates a uicontrol with a specific parent and one or more uicontrol properties.

`uicontrol(c)` gives focus to a specific uicontrol object, `c`.

Specifying the Uicontrol Style

When selected, most uicontrol objects perform a predefined action. To create a specific type of uicontrol, set the `Style` property as one of the strings that follow. You can specify a partial string if it is unique among all the styles. For example, instead of `'radiobutton'`, you can specify `'radio'`.

- `'checkbox'` – A check box generates an action when you select it. Use check boxes to provide a number of independent choices. To activate a check box, click the mouse button on the object. The check box updates its appearance when its state changes.
- `'edit'` – Editable text fields enable you to enter or modify text values. Use editable text when you want free text as input. To enable multiple lines of text, set `Max-Min>1`. Multiline edit boxes provide a vertical scroll bar for scrolling. The arrow keys also provide a way to scroll. Obtain the current text by getting the `String` property. The `String` property does not update as you type in an edit box. To execute the callback routine for an edit text control, type in the desired text and then do one of the following:
  - Click another component, the menu bar, or elsewhere on the window.
  - For a single line editable text box, press **Enter**.
  - For a multiline editable text box, press **Ctl+Enter**.
- `'frame'`

  Note MathWorks recommends you use `uipanel` or `uibuttongroup` instead of frames.

  GUIDE continues to support frames in UIs that contain them, but the frame component does not appear in the GUIDE Layout Editor component palette.
- `'listbox'` – List boxes display a list of items, from which you can select one or more items. Unlike pop-up menus, list boxes do not expand when clicked. The `Min` and `Max` properties control the selection mode:
  - To enable multiple selection of items, set `Max-Min > 1`.
  - To enable selection of only one item at a time, set `Max-Min <= 1`

  The `Value` property stores the row indexes of currently selected list box items, and is a vector value when you select multiple items. After any mouse button up event that changes the `Value` property, MATLAB evaluates the list box's callback routine. To delay action when multiple items can be selected, you can associate a "Done" push button with the list box. Use the callback for that button to evaluate the list box `Value` property.

  List boxes with the `Enable` property set to `on` differentiate between single and double left clicks. MATLAB sets the figure `SelectionType` property to `normal` or `open` accordingly before evaluating the list box `Callback` property. For enabled list boxes,

Ctrl-left click and **Shift**-left click also set the figure `SelectionType` property to `normal` or `open`, respectively indicating a single or double click.

- `'popupmenu'` – Pop-up menus (also known as drop-down menus) display a list of choices when you open them with a button-press. When closed, a pop-up menu indicates the current choice. Pop-up menus are useful when you want to provide a number of mutually exclusive choices, but do not want to take up the amount of space that a group of radio buttons requires.
- `'pushbutton'` – Push buttons generate an action when activated. Left-click a push button to activate it. The button appears to depress until you release the mouse button. The callback activates when you release the mouse button while still pointing within the push button.
- `'radiobutton'` – Radio buttons are similar to check boxes, but are intended to be mutually exclusive within a group of related radio buttons. When used this way, you can only select one radio button at any given time). To activate a radio button, click and release the mouse button over it. The easiest way to implement mutually exclusive behavior for a set of radio buttons is to place them within a `uibuttongroup`.
- `'slider'` – Sliders accept numeric input within a specific range when you move the "thumb" button along a bar. The location of the thumb indicates a numeric value, assigned to the `Value` property when you release the mouse button. You can set the minimum, maximum, and current values, and step sizes of a slider.

  Move the thumb by doing any one of the following:

  - Press the mouse button on the thumb, and drag it along the bar.
  - Click in the bar or on arrow buttons located at both ends of the bar.
  - Click the keyboard arrow keys when the slider is in focus.
- `'text'` – Static text boxes display lines of text. You typically use static text to label other controls, provide information to the user, or indicate values associated with a slider. If you assign the Callback property of a static text object to a function (or a string of commands), the static text will not respond when users try to interact with the text. However, you can code the `ButtonDownFcn` callback to respond to mouse clicks on the static text. See "Tips" on page 1-8439 for more information.
- `'togglebutton'` – Toggle buttons are similar in appearance to push buttons, but they visually indicate their state, either `'on'` (depressed) or `'off'` (up). Clicking a toggle button changes its state, and switches its `Value` property between the toggle button's `Min` and `Max` values.

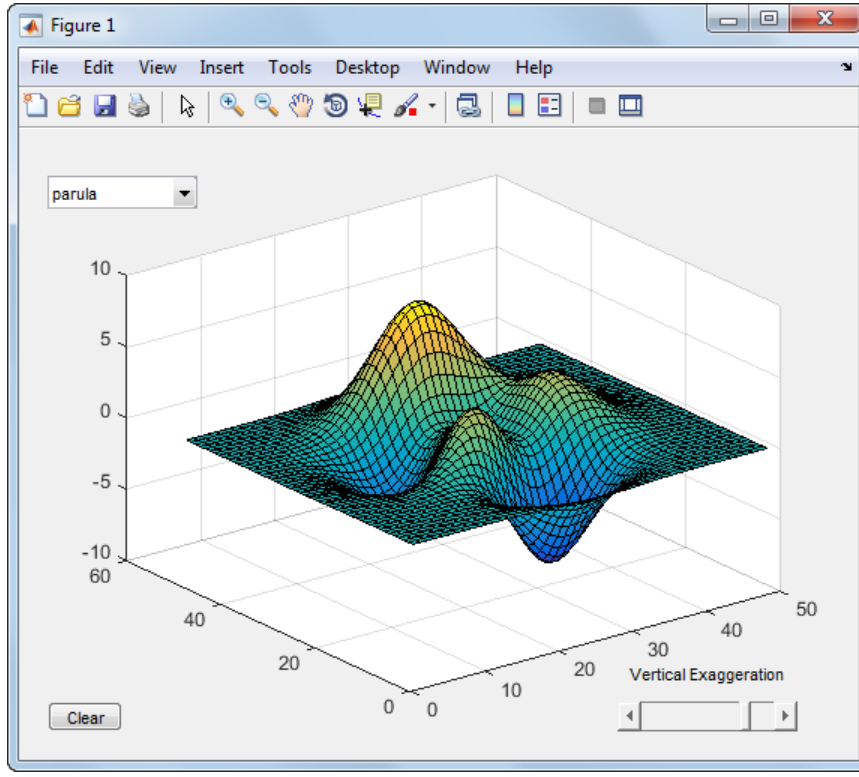
Examples

Create uicontrols to allow users to adjust the appearance of a plot. For instance, create a program file called `myui.m` that contains the following code.

```
function myui
   % Create a figure and axes
   f = figure('Visible','off');
   ax = axes('Units','pixels');
   surf(peaks)

   % Create pop-up menu
   popup = uicontrol('Style', 'popup',...
          'String', {'parula','jet','hsv','hot','cool','gray'},...
          'Position', [20 340 100 50],...
          'Callback', @setmap);

  % Create push button
   btn = uicontrol('Style', 'pushbutton', 'String', 'Clear',...
       'Position', [20 20 50 20],...
       'Callback', 'cla');

  % Create slider
   sld = uicontrol('Style', 'slider',...
       'Min',1,'Max',50,'Value',41,...
       'Position', [400 20 120 20],...
       'Callback', @surfzlim);

   % Add a text uicontrol to label the slider.
   txt = uicontrol('Style','text',...
       'Position',[400 45 120 20],...
       'String','Vertical Exaggeration');

   % Make figure visble after adding all components
   f.Visible = 'on';
   % This code uses dot notation to set properties.
   % Dot notation runs in R2014b and later.
   % For R2014a and earlier: set(f,'Visible','on');

   function setmap(source,callbackdata)
       val = source.Value;
       maps = source.String;
       % For R2014a and earlier:
       % val = get(source,'Value');
```

```
        % maps = get(source,'String');

        newmap = maps{val};
        colormap(newmap);
    end

    function surfzlim(source,callbackdata)
        val = 51 - source.Value;
        % For R2014a and earlier:
        % val = 51 - get(source,'Value');

        zlim(ax,[-val val]);
    end
end
```

The resulting UI displays a plot. Users can adjust the color map, change the vertical scaling, or clear the axes.



More About

Tips

- To make static text respond to mouse clicks, set the text object's `Enable` property to `'Inactive'`. Then, define a `ButtonDownFcn` callback that responds when the user clicks on the text. For example, the following code displays `'Text was clicked'` in the Command Window when the user clicks on the text in the UI.

  ```
  mytxt = uicontrol('Style','text','String','Click Me!');
  mytxt.Enable = 'Inactive';
  mytxt.ButtonDownFcn = 'disp(''Text was clicked'')';
  ```

  Note that this code uses dot notation to set properties. Dot notation runs in R2014b and later. If you are using an earlier release, use the `set` function instead.

- "Access Property Values"

See Also

`figure` | `uibuttongroup` | Uicontrol Properties

Introduced before R2006a

Uicontrol Properties

Control appearance and behavior of user interface control

Uicontrols are interface components, such as buttons and sliders, that users can interact with. The `uicontrol` function creates an interface component and sets any required properties before displaying it. By changing uicontrol property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
b = uicontrol('Style','pushbutton');
pos = b.Position;
b.Position = [100 100 50 20];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uicontrol visibility

`'on'` (default) | `'off'`

Uicontrol visibility, specified as `'on'` or `'off'`. When Visible is `'off'`, the uicontrol is not visible, but you can query and set its properties.

To make your program start faster, set the Visible property of all uicontrols that are not initially displayed to `'off'`.

`BackgroundColor` — Uicontrol background color

`[.94 .94 .94]` (default) | RGB triplet | short name | long name

Uicontrol background color, specified as an RGB triplet, short name, or long name. The color you specify fills the area bounded by the uicontrol.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`

Long Name	Short Name	RGB Triplet
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Note: Some operating systems override the `BackgroundColor` value.

Data Types: `double` | `char`

`ForegroundColor` — Uicontrol text color

`[0 0 0]` (default) | RGB triplet | short name | long name

Uicontrol text color, specified as an RGB triplet, short name, or long name from the table below.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]

Long Name	Short Name	RGB Triplet
'black	'k'	[0 0 0]

Note: If you change the value of `ForegroundColor` for a listbox, then MATLAB uses that color for all listbox items, except for the currently selected listbox item. For selected items, MATLAB uses a color that ensures good contrast between the text of selected items and the selection color.

Example: `[0 0 1]`

Example: `'b'`

Example: `'blue'`

Data Types: `double | char`

`CData` — Optional image to display on uicontrol

3-D array of truecolor RGB values

Optional image to display on the uicontrol, specified as a 3-D array of truecolor RGB values. The values in the array can be:

- Double-precision values between `0.0` and `1.0`
- `uint8` values between `0` and `255`

Push buttons and toggle buttons are the only uicontrols that fully support CData. If you specify the CData property for a radio button or check box, the image might overlap with the text string. Also, specifying an image for radio button or check box disables the ability to show when they are selected or deselected.

Example: `zeros(16,16,3)`

Example: `uint8(zeros(16,16,3))`

Data Types: `double | uint8`

Location and Size

`Position` — Location and size of uicontrol

`[left bottom width height]`

Location and size of the uicontrol relative to the parent, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the uicontrol
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the uicontrol
`width`	Distance between the right and left outer edges of the uicontrol
`height`	Distance between the top and bottom outer edges of the uicontrol

All measurements are in units specified by the Units property.

Note: If the parent of the uicontrol is a figure, then the Position values are relative to the figure's *drawable area*. The drawable area of a figure is the area inside the window borders, excluding the menu bar and tool bar.

Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the Position vector. For example, this code changes the width of the uicontrol to `52`:

```
b = uicontrol;
b.Position(3) = 52;
b.Position
```

```
ans =

    20    20    52    20
```

Data Types: `double`

`Units` — Units of measurement

`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as `'pixels'`, `'normalized'`, `'inches'`, `'centimeters'`, `'points'`, or `'characters'`.

MATLAB uses these units to interpret the location and size values of the Position property:

- The size of a uicontrol specified in pixel units depends on the system display settings and resolution.
- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- Character units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the Units before the Position property, then MATLAB sets Position using the units you specify.
- If you specify the Units property after the Position property, MATLAB sets the position using the default Units. Then, MATLAB converts the Position value to the equivalent value in units you specify.

`Extent` — Size of uicontrol rectangle

four-element row vector

Size of uicontrol rectangle, returned as a four-element row vector. The first two elements of the vector are always zero. The third and fourth elements are the width and height of the rectangle containing the uicontrol, respectively. All measurements are in units specified by the Units property.

MATLAB determines the size of the rectangle based on the size of the String property value and the font characteristics. To adjust the width and height of the uicontrol to accommodate the size of the String value, set the Position width and height values to be slightly larger than the Extent width and height values.

For single line strings, the height element of the Extent property indicates the height of a single line. The width element indicates the width of the longest line, even if the

string wraps when displayed on the control. For multiline strings, the Extent rectangle encompasses all lines of text. Editable text fields are considered multiline if Max – Min > 1.

When the uicontrol Units property is `'normalized'`, the Extent values are measured relative to the figure, regardless of whether the uicontrol is contained in (parented to) a uipanel.

Font Style

`FontName` — Font for displaying uicontrol text

`'Helvetica'` (default) | string

Font for displaying the uicontrol text, specified as a string. This property specifies the name of the font to use to display the text specified in the uicontrol 'String' property.

To use a fixed-width font, set the FontName property to the string, `'FixedWidth'`. This setting instructs MATLAB to use the value of the root FixedWidthFontName property, which you can set in the `startup.m` file.

Ror more information about the `startup.m` file, see"Startup Options in MATLAB Startup File".

Example: `'Arial'`

`FontSize` — Font size for uicontrol text

positive number

Font size for the uicontrol text, specified as a positive number. The FontUnits property specifies the units. The default size is system-dependent.

Example: `12`

Example: `12.5`

Data Types: `double`

`FontUnits` — Units of font size for uicontrol text

`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Units of font size for the uicontrol text, specified as `'points'`, `'normalized'`, `'inches'`, `'centimeters'`, or `'pixels'`.

If you set this property to `'normalized'`, then MATLAB interprets the font size as a fraction of the uicontrol height. When you resize the uicontrol, MATLAB scales the displayed font to maintain that fraction.

The other FontUnits options (`pixels`, `inches`, `centimeters`, and `points`) are absolute units. 1 point = 1/72 inch.

`FontWeight` — Font weight for uicontrol text

`'normal'` (default) | `'bold'`

Font weight for the uicontrol text, specified as a value from the table below.

FontWeight Value	Description
`'normal'`	Normal font weight
`'bold'`	Heavy font weight

Not all fonts support all font weights. Therefore, if you specify an unsupported value for the `FontWeight` property, the result might appear the same as the default.

Note: The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

`FontAngle` — Character slant of uicontrol text

`'normal'` (default) | `'italic'`

Character slant of uicontrol text, specified as `'normal'` or `'italic'`. MATLAB uses this property to select a font from those available on your system. Setting this property to `'italic'` selects a slanted version of the font, if it is available on your system.

Note: The `'oblique'` value has been removed. Use `'italic'` instead.

Text

`String` — Text to display

string | cell array of `char` values | pipe-delimited row vector | padded column matrix

Text to display, specified as a string, cell array, a pipe-delimited row vector, or a padded column matrix. The uicontrol `Style` property value determines the array format you can use.

Style Property Value	Supported Array Formats	Examples
`'checkbox'`	• String • Cell array of char values	`'Item 1'` `{'Item 1'}`
`'edit'`	• String • Cell array of char values	`'Default text'` `{'Default text'}`
`'pushbutton'`	• String • Cell array of char values	`'Evaluate'` `{'Evaluate'}`
`'radiobutton'`	• String • Cell array of char values	`'Round to nearest integer'` `{'Round to nearest integer'}`
`'text'`	• String • Cell array of char values	`'Enter values'` `{'Enter values'}`
`'togglebutton'`	• String • Cell array of char values	`'Activate'` `{'Activate'}`

`'listbox'`	• String
	• Cell array of char values
	• Pipe-delimited row vector
	• Padded column matrix
		`'Red'`
		`{'Red','Blue','Orange'}`
		`'Red|Blue|Orange'`
		`['Red   ';'Blue  ';'Orange']`

`'popupmenu'`	• String	`'Red'`
		`{'Red','Blue','Orange'}`
		`'Red|Blue|Orange'`

Style Property Value	Supported Array Formats	Examples
	• Cell array of char values • Pipe-delimited row vector • Padded column matrix	`['Red   ';'Blue  ';'Orange']`

Note: Some important characteristics of the String property:

- If you specify a cell array for a check box, push button, radio button, or toggle button, then MATLAB displays only the first element in the cell array.
- If you programmatically replace the string of an `'edit'` style of uicontrol, the cursor moves to the beginning of the text.
- If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as Multiples of π.

Data Types: `char`

`HorizontalAlignment` — Alignment of uicontrol text

`'center'` (default) | `'left'` | `'right'`

Alignment of the uicontrol text, specified as `'center'`, `'left'`, or `'right'`. This property determines the justification of the String property text.

Interactive Control

`Callback` — Callback function that executes when user interacts with uicontrol

`''` (default) | function handle | cell array | string

Uicontrol action, specified as one of these values:

- Function handle

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

This is a function that makes the uicontrol respond to user inputs, such as push button clicks, slider movements, or check box selections.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The ButtonDownFcn callback is a function that executes when the user clicks a mouse button on the uicontrol. The callback executes in the following situations:

- The user right-clicks the uicontrol, and the uicontrol Enable property is set to `'on'`.
- The end user right-clicks or left-clicks the Uicontrol, and the uicontrol Enable property is set to `'off'` or `'inactive'`.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

KeyPressFcn — Key press callback function

'' (default) | function handle | cell array | string

Key press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes when the uicontrol object has focus and the user presses a key. If you do not define a function for this property, MATLAB passes key presses to the parent figure. Repeated key presses retain the focus of the uicontrol, and the function executes with each key press. If the user presses multiple keys at approximately the same time, MATLAB detects the key press for the last key pressed.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description	Examples:			
		a	=	Shift	Shift-a
Character	The character that displays as a result of pressing a key or keys. The character can be empty or unprintable.	'a'	'='	''	'A'
Modifier	A cell array containing the names of one or more modifier keys that are being pressed (such as, **Ctrl**, **Alt**, **Shift**).	{1x0 cell}	{1x0 cell}	{'shift'}	{'shift'}
Key	The key being pressed, identified by the (lowercase) label on the key, or a descriptive string.	'a'	'equal'	'shift'	'a'

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Source`	The object that has focus when the user presses the key.	uicontrol object	uicontrol object	uicontrol object	uicontrol object
`EventName`	The action that caused the callback function to execute.	`'KeyPre`	`'KeyPre`	`'KeyPress`	`'KeyPress'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the CurrentCharacter property of the figure to determine which character the user pressed.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`KeyReleaseFcn` — Key-release callback function

`''` (default) | function handle | cell array | string

Key-release callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback function executes when the uicontrol object has focus and the user releases a key.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Character`	Character interpretation of the key that was released.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	Current modifier, such as `'control'`. This value is always an empty cell array for aseFcn callbacks.	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`
`Key`	Name of the key that was released, identified by the lowercase label on the key, or a descriptive string.	`'a'`	`'equal'`	`'shift'`	`'a'`
`Source`	The object that has focus when the user presses the key.	uicontrol object	uicontrol object	uicontrol object	uicontrol object
`EventName`	The action that caused the callback function to execute.	`'ase'`	`'ase'`	`'ase'`	`'ase'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the CurrentCharacter property of the figure to determine which character the user pressed.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`Enable` — Operational state of uicontrol

`'on'` (default) | `'off'` | `'inactive'`

Operational state of the uicontrol, specified as `'on'`, `'off'`, or `'inactive'`. The `Enable` property controls whether the uicontrol responds to button clicks. These are the possible values:

- `'on'` – The uicontrol is operational.
- `'off'` – The uicontrol is not operational and appears grayed-out.
- `'inactive'` – The uicontrol is not operational, but it has the same appearance as when `Enable` is set to `'on'`.

The value of the `Enable` property and the type of button click determine the response.

Enable Value	Response to Left-Click	Response to Right-Click
`'on'`	The uicontrol's `Callback` function executes.	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The uicontrol's `ButtonDownFcn` callback executes.
`'off'` or `'inactive'`	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The uicontrol's `ButtonDownFcn` callback executes.	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The uicontrol's `ButtonDownFcn` callback executes.

`TooltipString` — Tooltip text

string

Tooltip text, specified as a string. When the user hovers the mouse pointer over the uicontrol and leaves it there, the tooltip displays. If you want to create a tooltip that has more than one line of text, use `sprintf` to generate a string containing newline (`\n`) characters and then set the TooltipString to that string.

Specify TooltipString Containing Two Lines

```
b = uicontrol;
s = sprintf('Tooltip line 1\nTooltip line 2');
b.TooltipString = s;
```

`UIContextMenu` — Uicontrol context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Uicontrol context menu, specified as a uicontextmenu object. Use this property to display a context menu when the user right-clicks on the uicontrol. Create the context menu using the `uicontextmenu` function.

`Selected` — Selection status of uicontrol

`'off'` (default) | `'on'`

Note: The behavior of the Selected property changed in R2014b, and it is not recommended. It no longer has any effect on uicontrols. This property might be removed in a future release.

`SelectionHighlight` — Ability to highlight selection handles

`'on'` (default) | `'off'`

Note: The behavior of the SelectionHighlight property changed in R2014b, and it is not recommended. It no longer has any effect on uicontrols. This property might be removed in a future release.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

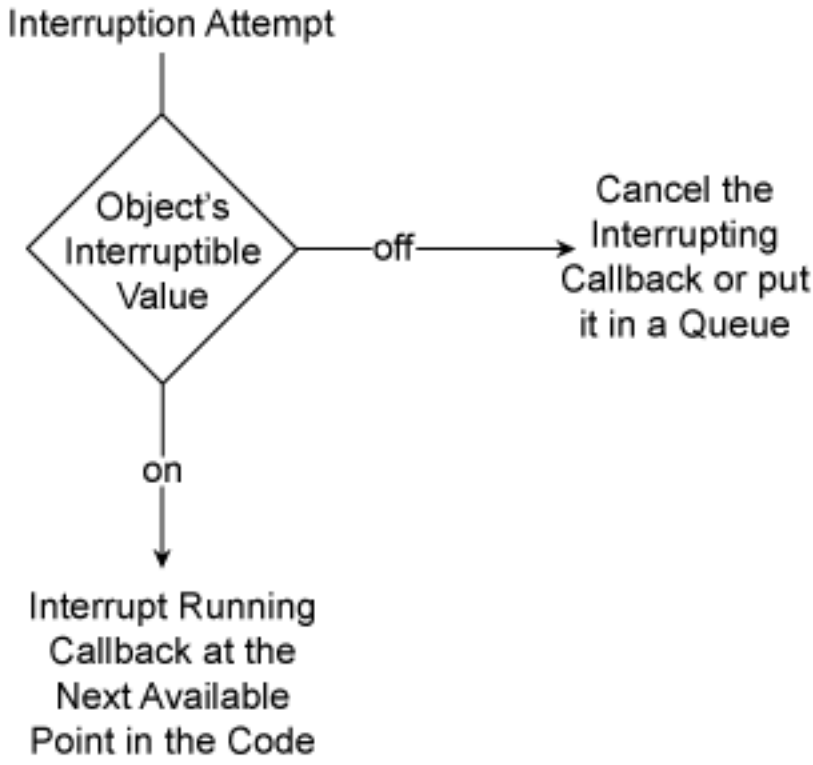
Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction

property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uicontrol callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See "Interrupt Callback Execution" for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

This property has no effect on the uicontrol.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uicontrol

`'off'` (default) | `'on'`

Deletion status of uicontrol, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uicontrol begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uicontrol no longer exists.

Check the value of the BeingDeleted property to verify that the uicontrol is not about to be deleted before querying or modifying it.

`CreateFcn` — Uicontrol creation function

function handle | cell array | string

Uicontrol creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uicontrol. MATLAB initializes all uicontrol property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uicontrol that is being created.

Setting the CreateFcn property on an existing uicontrol has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uicontrol object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uicontrol deletion function

function handle | cell array | string

Uicontrol deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uicontrol (for example, when the end user deletes the figure). MATLAB executes the

DeleteFcn callback before destroying the properties of the uicontrol. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uicontrol that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object
`'uicontrol'`

Type of graphics object, returned as `'uicontrol'`.

`Tag` — Uicontrol identifier
`''` (default) | string

Uicontrol identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uicontrol. When you need access to the uicontrol elsewhere in your code, you can use the `findobj` function to search for the uicontrol based on the Tag value.

Example: `'pushbutton1'`

Data Types: `char`

`UserData` — Data to associate with the uicontrol object
empty array (default) | array

Data to associate with the uicontrol object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uicontrol parent

figure | uipanel | uibuttongroup | uitab

Uicontrol parent, specified as a figure, uipanel, uibuttongroup, or uitab. You can move a uicontrol to a different figure, uipanel, uibuttongroup, or uitab by setting this property to the handle of the target figure, uipanel, uibuttongroup, or uitab.

`Children` — Children of uicontrol

empty array

Children of uicontrol, returned as an empty array. Uicontrol objects have no children. Setting this property has no effect.

`HandleVisibility` — Visibility of uicontrol handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uicontrol handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uicontrol handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uicontrol handle is always visible.
`'callback'`	The uicontrol handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uicontrol at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The uicontrol handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

Note: Do not try to access radio buttons and toggle buttons that are managed by a `uibuttongroup` outside of the button group. Set the HandleVisibility of those radio buttons and toggle buttons to `'off'` to prevent accidental access.

Type of Control

`Style` — Style of uicontrol

`'pushbutton'` (default) | string

Style of uicontrol, specified as a string from the following table. See "Specifying the Uicontrol Style" on page 1-8434 for more details about each style.

Style Value	Description
`'pushbutton'`	Button that appears to depress until you release the mouse button.
`'togglebutton'`	Button that can have two states: up or depressed. The state of the toggle button changes every time you click it.
`'checkbox'`	Check box that can have two states: checked or unchecked. The state changes when the user clicks and releases the mouse button over it.
`'radiobutton'`	Button that can have two states: selected or deselected. Radio buttons are intended to be mutually exclusive within a group of related radio buttons.
`'edit'`	Editable text field.

Style Value	Description
`'text'`	Static text field.
`'slider'`	Button that the user pushes along a horizontal or vertical bar. The position of the button indicates a value within a specified range.
`'listbox'`	List of items from which the user can select one or more items. Unlike pop-up menus, list boxes do not expand when clicked.
`'popupmenu'`	Isolated menu that expands to display a list of choices when you click it. When it is collapsed, the menu shows the current choice.

`Value` — Current value of uicontrol

number

Current value of uicontrol, specified as a number. The Value property is useful in querying or modifying the status of certain uicontrols:

Style of uicontrol	Description of Value Property
`'togglebutton'`	• **Raised:** Value property equals the value of the Min property. • **Depressed:** Value property equals the value of the Max property.
`'checkbox'`	• **Unchecked:** Value property changes to the value of the Min property. • **Checked:** Value property changes to the value of the Max property.
`'radiobutton'`	• **Deselected:** Value property changes to the value of the Min property. • **Selected:** Value property changes to the value of the Max property.
`'slider'`	Value property equals the corresponding slider value.
`'listbox'`	Value property equals an array index corresponding to the selected item in the list box. A value of `1` corresponds to the first item in the list.

Style of uicontrol	Description of Value Property
`'popupmenu'`	Value property equals an array index corresponding to the selected item in the pop-up menu. A value of `1` corresponds to the first item in the pop-up menu.

`Max` — Maximum value of uicontrol

`1` (default) | number

Maximum value of uicontrol, specified as a number. The Max property affects the presentation of certain uicontrols:

Style of uicontrol	Description of Value Property
`'togglebutton'`	When the toggle button is depressed, the Value property changes to the value of the Max property.
`'checkbox'`	When the check box is checked, the Value property changes to the value of the Max property.
`'radiobutton'`	When the radio button is selected, the Value property changes to the value of the Max property.
`'edit'`	The edit text box accepts multiple lines of input when `Max – Min > 1`. Otherwise, the edit text box accepts a single line of input. The absolute values of `Max` and `Min` have no effect on the number of possible lines. As long as the difference is greater than `1`, the edit box can contain any number of lines.
`'slider'`	The Max property value is the maximum slider value, which must be greater than the Min property value.
`'listbox'`	The Max property value helps determine whether the user can select multiple items in the list box simultaneously. If Max – Min > 1, then the user can select multiple items simultaneously. Otherwise, the user cannot select multiple items simultaneously. If you set the Max and Min properties to allow multiple selections, then the Value property value can be a vector of indices.

`Min` — Minimum value of uicontrol

`0` (default) | number

Minimim value of uicontrol, specified as a number. The Min property affects the presentation of certain uicontrols:

Style of uicontrol	Description of Value Property
`'togglebutton'`	When the toggle button is raised, the Value property changes to the value of the Min property.
`'checkbox'`	When the check box is unchecked, the Value property changes to the value of the Min property.
`'radiobutton'`	When the radio button is deselected, the Value property changes to the value of the Min property.
`'edit'`	The edit text box accepts multiple lines of input when `Max – Min > 1`. Otherwise, the edit text box accepts a single line of input. The absolute values of `Max` and `Min` have no effect on the number of possible lines. As long as the difference is greater than `1`, the edit box can contain any number of lines.
`'slider'`	The Min property value is the minimum slider value, which must be less than the Max property value.
`'listbox'`	The Max property value helps determine whether the user can select multiple items in the list box simultaneously. If Max – Min > 1, then the user can select multiple items simultaneously. Otherwise, the user cannot select multiple items simultaneously. If you set the Max and Min properties to allow multiple selections, then the Value property value can be a vector of indices.

`SliderStep` — Slider step size

`[0.01 0.10]` (default) | `[minorstep majorstep]`

Slider step size, specified as the array, `[minorstep majorstep]`. This property controls the magnitude of the slider value change when the user clicks the arrow keys or the slider trough (slider channel):

- The slider Value property increases or decreases by the value of `minorstep` when the user presses an arrow key.

- The slider Value property increases or decreases by the value of `majorstep` when the user clicks the slider trough.

Both `minorstep` and `majorstep` must be greater than `1e-6`, and `minorstep` must be less than or equal to `majorstep`.

The actual step size depends on the SliderStep value and the slider range (Max – Min). For example, a slider having SliderStep value of `[0.01 0.10]`, Max value of `1`, and Min of `0` provides a 1% change when the user presses an arrow key and a 10% change when the user clicks in the trough.

As `majorstep` increases, the slider thumb indicator grows longer. When `majorstep` is equal to `1`, the thumb indicator is half as long as the trough. The size is larger for `majorstep` values greater than `1`.

Example: `[.5 1]`

`ListboxTop` — Index of top item in list box

1 (default) | integer value

Index of top item in list box, specified as an integer value. This property applies only to the listbox style of uicontrol. This property specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. The ListboxTop value is an index into the array of strings you specify as the String property value. The ListboxTop value must be between `1` and the number of strings in the array. Noninteger values are fixed to the next lowest integer.

Note: The String and Value properties might override the value of the ListboxTop property regardless of the ListboxTop value you specify. The ListboxTop value can change depending on the value of other uicontrol properties. For example, explicitly setting the Value property scrolls the list to that value.

For the most reliable results, query or modify the ListboxTop property after MATLAB finishes drawing the uicontrol on the screen.

Example: `3`

Data Types: `double`

See Also

`uicontrol`

More About

- "Access Property Values"
- "Default Property Values"

uigetdir

Open folder selection dialog box

Syntax

```
folder_name = uigetdir
folder_name = uigetdir(start_path)
folder_name = uigetdir(start_path,dialog_title)
```

Description

`folder_name = uigetdir` displays a modal dialog box showing the folders that are inside the current working directory. This dialog allows you to navigate to a folder and select it (or type the name of a folder). If the folder you specify exists, `uigetdir` returns the selected path when you click **OK**. If you click **Cancel** (or the window's close box), `uigetdir` returns `0`.

`folder_name = uigetdir(start_path)` shows the folders that are inside the folder, `start_path`. If `start_path` is an empty string (`''`) or is not a valid path, the dialog box opens in the current working directory.

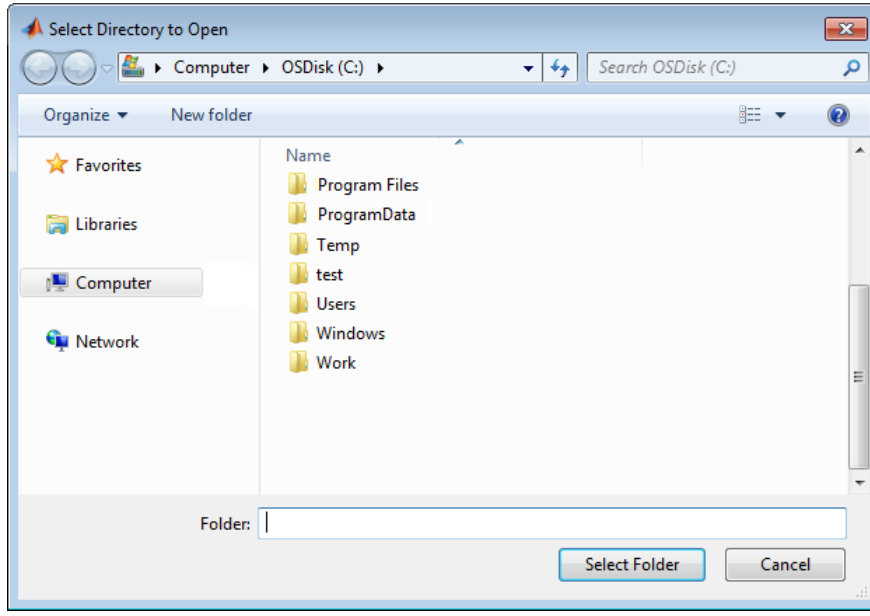
`folder_name = uigetdir(start_path,dialog_title)` opens a dialog box with the title, `dialog_title`. The default `dialog_title` is **Select folder to Open**.

Note: The visual characteristics of the dialog box depend on the operating system that is running MATLAB.

Examples

The following statement displays directories on the `C:` drive.

```
dname = uigetdir('C:\');
```



This statement uses `matlabroot` to display the MATLAB root folder in the dialog box:

```
uigetdir(matlabroot,'MATLAB Root Directory')
```

See Also

`uigetfile` | `uiputfile`

Introduced before R2006a

uigetfile

Open file selection dialog box

Syntax

```
filename = uigetfile
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle)
[FileName,PathName,FilterIndex] =
uigetfile(FilterSpec,DialogTitle,DefaultName)
[FileName,PathName,FilterIndex] =
uigetfile(...,'MultiSelect',selectmode)
```

Description

`filename = uigetfile` displays a modal dialog box that lists files in the current folder and enables you to select or enter the name of a file. If the file name is valid (and the file exists), `uigetfile` returns the file name as a string when you click **Open**. If you click **Cancel** (or the window's close box), `uigetfile` returns `0`.

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)` displays only those files with extensions that match `FilterSpec`. On some platforms `uigetfile` also displays in gray the files that do not match `FilterSpec`. `FilterSpec` can be a string or a cell array of strings, and can include the `*` wildcard.

- If `FilterSpec` is a file name, that file name displays, selected in the **File name** field. The extension of the file is the default filter.
- `FilterSpec` can include a path. That path can contain `'.'`,`'..'`, \, `'/'`, or `'~'`. For example, `'../*.m'` lists all code files in the folder above the current folder.
- If `FilterSpec` is a folder name, `uigetfile` displays the contents of that folder, the **File name** field is empty, and no filter applies. To specify a folder name, make the last character of `FilterSpec` either `'\'` or `'/'`.
- If `FilterSpec` is a cell array of strings, it can include two columns. The first column contains a list of file extensions. The optional second column contains a corresponding

list of descriptions. These descriptions replace standard descriptions in the **Files of type** field. A description cannot be an empty string. The second and third examples illustrate use of a cell array as `FilterSpec`.

If `FilterSpec` is missing or empty, `uigetfile` uses the default list of file types (for example, all MATLAB files).

After you click **Open** and if the file name exists,`uigetfile` returns the name of the file in `FileName` and its path in `PathName`. If you click **Cancel** or the window's close box, the function sets `FileName` and `PathName` to `0`.

`FilterIndex` is the index of the filter selected in the dialog box. Indexing starts at 1. If you click **Cancel** or the window's close box, the function sets `FilterIndex` to `0`.

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle)` displays a dialog box that has the title `DialogTitle`. To use the default file types and specify a dialog title, enter

```
uigetfile('',DialogTitle)
```

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle,DefaultName)` displays a dialog box in which the file name specified by `DefaultName` appears in the **File name** field. `DefaultName` can also be a path or a path/filename. In this case, `uigetfile` opens the dialog box in the folder specified by the `path`. You can use `'.'`,`'..'`, `\`, or `'/'` in the `DefaultName` argument. To specify a folder name, make the last character of `DefaultName` either `'\'` or `'/'`. If the specified path does not exist, `uigetfile` opens the dialog box in the current folder.

`[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',`*`selectmode`*`)` opens the dialog box in *multiselect* mode. Valid values for *`selectmode`* are `'on'` and `'off'` (the default, which allows single selection only). If `'MultiSelect'` is `'on'` and you select more than one file in the dialog box, then `FileName` is a cell array of strings. Each array element contains the name of a selected file. File names in the cell array are sorted in the order your platform uses. If you select multiple files, they must be in the same folder, otherwise MATLAB displays a warning dialog box. Be aware that Microsoft Windows libraries can span multiple folders. `PathName` is a string identifying the folder containing the files.

If you include either of the wildcard characters, `'*'` or `'?'`, in a file name, `uigetfile` does not respond to clicking **Open**. The dialog box remains open until you cancel it or

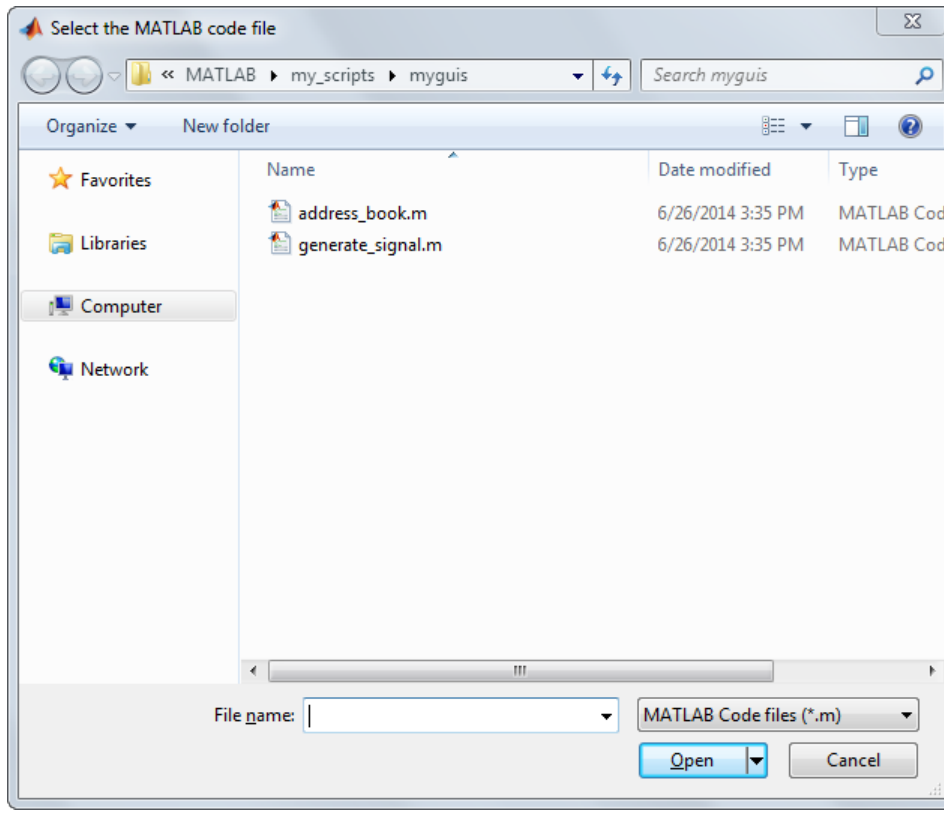
remove the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

Note: The visual characteristics of the dialog box depend on the operating system that is running MATLAB.

Examples

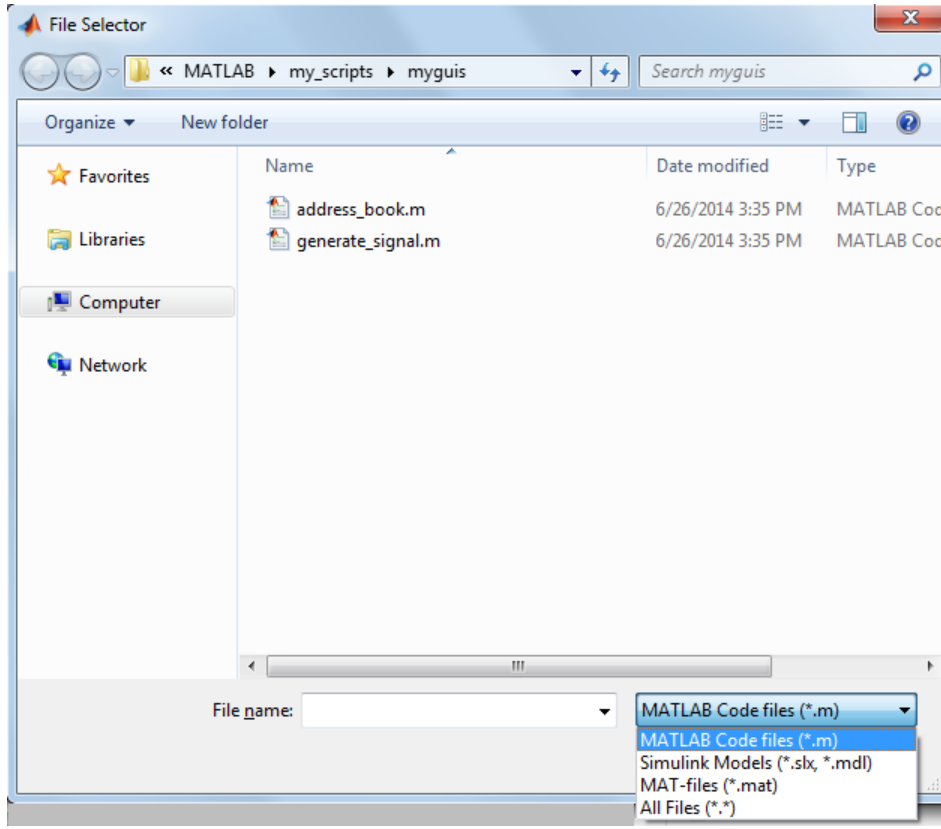
This dialog box lists all MATLAB code files in the current directory. `uigetfile` returns the name and path to the file you select.

```
[FileName,PathName] = uigetfile('*.m','Select the MATLAB code file');
```



To create a list of file types that appears in the file type drop-down list, separate the file extensions with semicolons, as in the following code. `uigetfile` displays a default description for each known file type.

```
[filename, pathname] = ...
     uigetfile({'*.m';'*.slx';'*.mat';'*.*'},'File Selector');
```

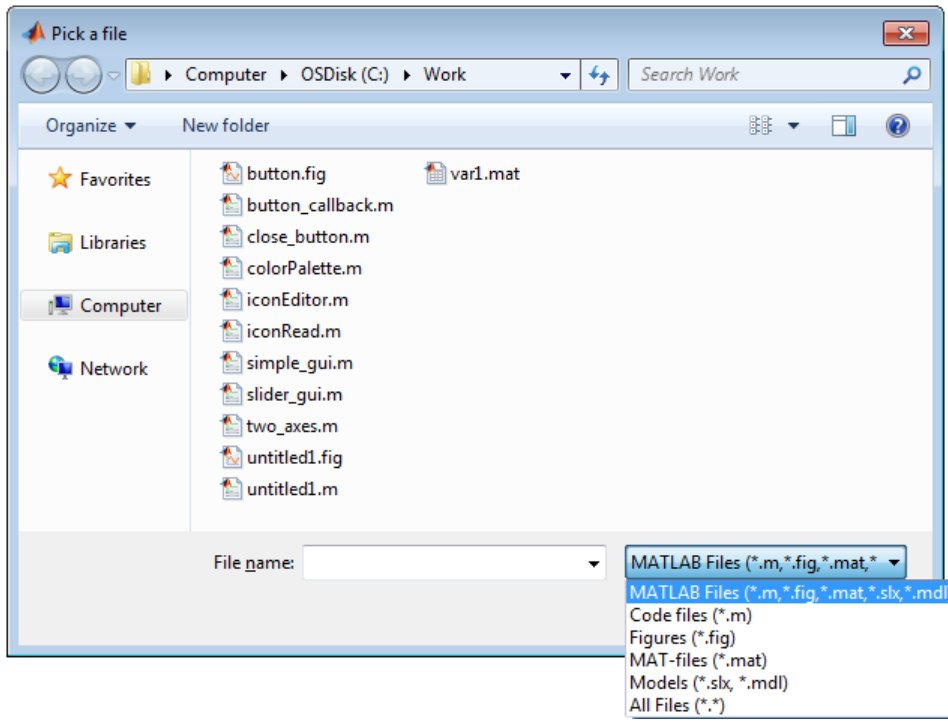


If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the `'MATLAB Files'` and `'Models'` descriptions.

```
[filename, pathname] = uigetfile( ...
{'*.m;*.fig;*.mat;*.slx;*.mdl',...
 'MATLAB Files (*.m,*.fig,*.mat,*.slx,*.mdl)';
   '*.m',  'Code files (*.m)'; ...
   '*.fig','Figures (*.fig)'; ...
```

```
'*.mat','MAT-files (*.mat)'; ...
'*.mdl;*.slx','Models (*.slx, *.mdl)'; ...
'*.*',  'All Files (*.*)'}, ...
'Pick a file');
```

The first column of the cell array contains the file extensions, while the second contains your descriptions of the file types. In this example, the first entry of column one contains several extensions, separated by semicolons, which are all associated with the description `'MATLAB Files (*.m,*.fig,*.mat,*.mdl)'`. The code produces the dialog box shown in the following figure.



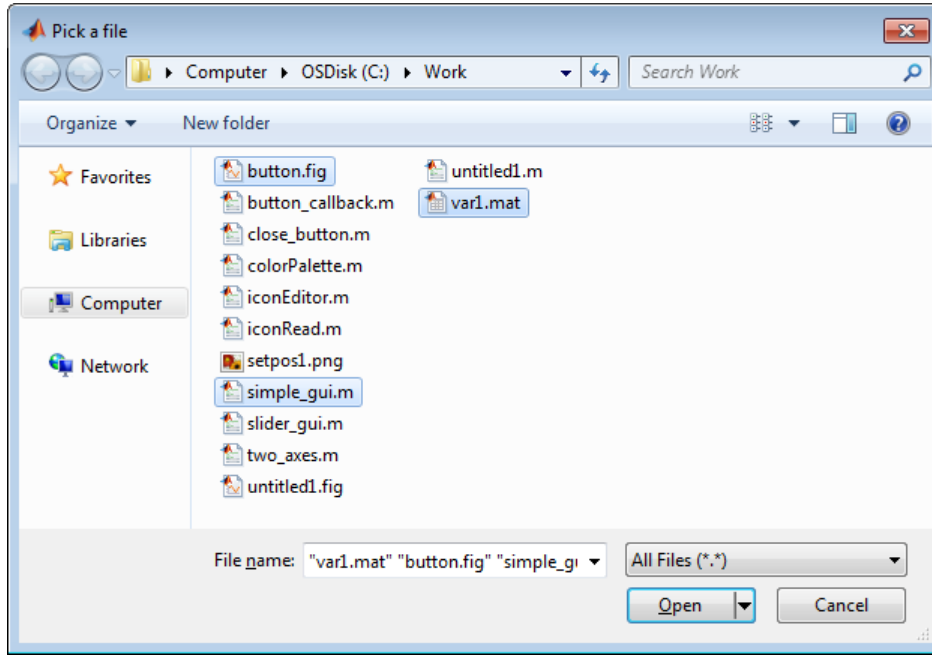
The following code lets you select a file, and then displays a message in the Command Window that summarizes the result.

```
[filename, pathname] = uigetfile('*.m', 'Select a MATLAB code file');
if isequal(filename,0)
   disp('User selected Cancel')
else
   disp(['User selected ', fullfile(pathname, filename)])
```

```
end
```

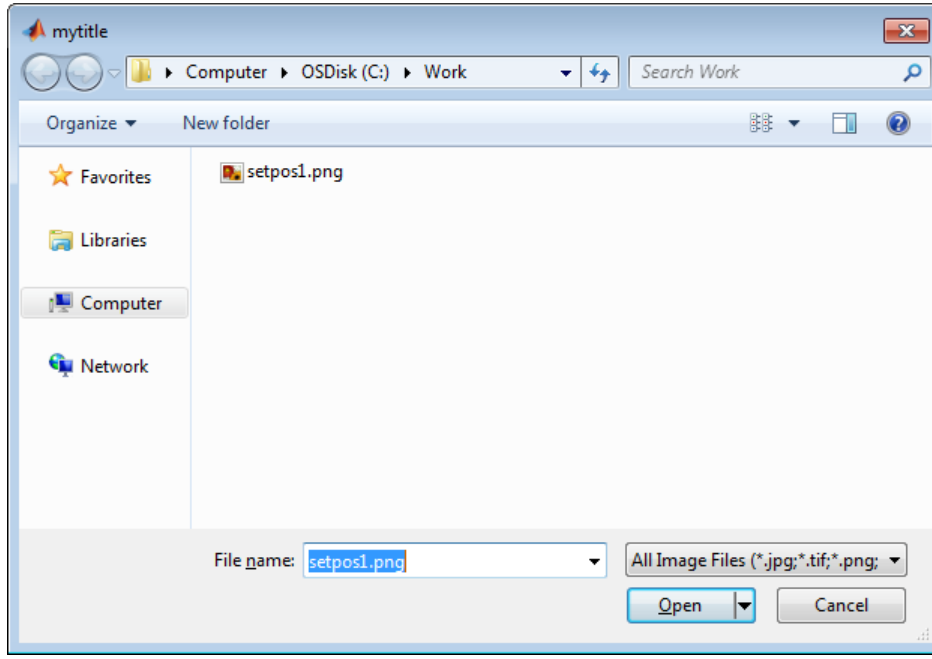
This code creates a list of file types and gives them descriptions that are different from the defaults. It also enables multiple-file selection. Select multiple files by holding down the **Shift** or **Ctrl** key and clicking on additional file names.

```
[filename, pathname, filterindex] = uigetfile( ...
{  '*.mat','MAT-files (*.mat)'; ...
   '*.slx;*.mdl','Models (*.slx, *.mdl)'; ...
   '*.*',  'All Files (*.*)'}, ...
   'Pick a file', ...
   'MultiSelect', 'on');
```



You can use the `DefaultName` argument to specify a start path and a default file name for the dialog box.

```
uigetfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'mytitle',...
          'C:\myfiles\my_examples\gbtools\setpos1.png')
```



Alternatives

Use the `dir` function to return a filtered or unfiltered list of files in your current folder or a folder you specify. `dir` also can return file attributes.

See Also

`uigetdir` | `uiopen` | `uiputfile`

Introduced before R2006a

uigetpref

Conditionally open dialog box according to user preference

Syntax

```
pref_value = uigetpref(group,pref,title,question,pref_choices)
[pref_value,dlgshown] = uigetpref(...)
[...] = uigetpref(... 'Name',value)
```

Description

`pref_value = uigetpref(group,pref,title,question,pref_choices)` returns one of the strings in `pref_choices` in either of two ways:

- Displays a multiple-choice dialog box that prompts you to answer a question. The dialog box includes a check box with the label **Do not show this dialog again**.
- Retrieves a previous answer from the preferences data base and returns it without displaying a dialog. No dialog is displayed if you previously selected the check box **Do not show this dialog again**.

`[pref_value,dlgshown] = uigetpref(...)` also returns in `dlgshown` whether or not the dialog displayed.

`[...] = uigetpref(... 'Name',value)` specify optional name-value pairs to control how dialogs display.

Input Arguments

`group`

String specifying the name of the preference group that preference `pref` belongs to. If the group does not already exist, `uigetpref` creates it.

Default: None

`pref`

String specifying the name of the preference that controls whether the dialog displays. If the preference does not already exist, `uigetpref` creates it.

Default: None

`title`

String to display in the dialog box title bar

Default:

`question`

String or cell array of strings specifying a descriptive paragraph for the dialog to display. Use `question` to define what you are asking the user to decide. Clearly state the alternatives and the consequences of choosing among them. The dialog box that `uigetpref` generates inserts line breaks between rows of the string array, between elements of the cell array of strings, and between `'|'` or newline characters within a string vector.

Default: None

`pref_choices`

String, cell array of strings, or `'|'`-separated strings specifying the labels for the dialog box push buttons. The string on the selected push button is returned.

If the internal preference values are different from the strings displayed on the push buttons, provide a 2-by-n cell array of strings. The first row contains the preference strings, and the second row contains the associated push button strings. When `pref_choices` is not a 2-by-n cell array, MATLAB constructs lowercase versions of the button labels and returns them in `pref_value`. If your code tests returned values, make sure it compares them against the appropriate strings.

Default: None

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'CheckboxState'`

State of check box when dialog box opens:

- `false` or `0`: Check box is not initially selected.
- `true` or `1`: Check box is initially selected.

Default: `0`

`'CheckboxString'`

String specifying the label for the check box.

Default: `'Do not show this dialog again'`

`'HelpString'`

String specifying the label for **Help** button.

Default: No **Help** button displays.

`'HelpFcn'`

String or function handle specifying the callback that executes when you click the **Help** button. Also, to have a button to trigger the callback, you must specify the `HelpString` option.

Default: `doc('uigetpref')`

`'ExtraOptions'`

String or cell array of strings specifying extra buttons. The additional buttons are not mapped to any preference settings. If you click any of these buttons, the dialog closes and `uigetpref` returns the string in `value`.

Default: `{}`

`'DefaultButton'`

String specifying the value that `uigetpref` returns if you close the dialog without clicking a button. This string does not have to correspond to any preference or `ExtraOption`.

Default: The first button specified in pref_choices

Output Arguments

`pref_value`

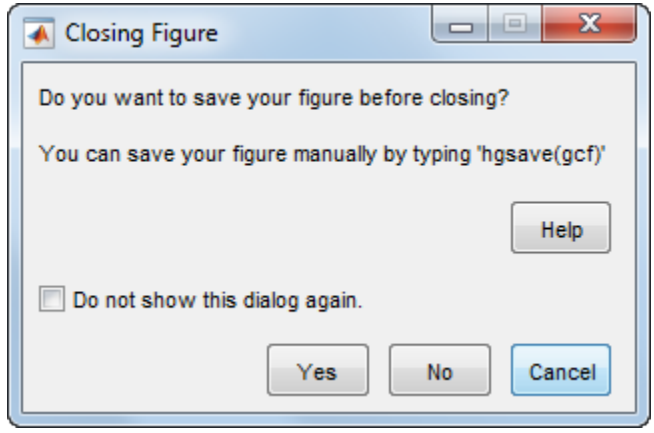
String containing the preference corresponding to the button you press. If you cancel the dialog by clicking its close box, `uigetpref` returns the label of the first button specified in pref_choices or the value for `DefaultButton`, if specified. After you select the **Do not show this dialog again** check box, `uigetpref` does not display a dialog box when you call it with the same group and pref arguments. Instead, it returns the last choice you made (your registered preference) in pref_value immediately.

`dlgshown`

Logical value that indicates whether or not the dialog displayed. The value is `1` when the dialog displays. The value is `0` when the dialog does not display. If the user selects the **Do not show this dialog again** check box, then the dialog does not display the next time your code calls the `uigetpref` function with the same group and pref arguments.

Examples

Create a preference dialog for the `'savefigurebeforeclosing'` preference in the `'mygraphics'` group of preferences.



This function, `save_figure_perhaps`, displays a dialog that asks the user to save current figure. The function responds appropriately, depending on the value returned by `uigetpref`.

```
function save_figure_perhaps
% Closes figure conditionally, asking whether to save it first.
% User can suppress the dialog from UIGETPREF permanently by selecting
% "Do not show this dialog again".

fig = gcf;
[selectedButton dlgshown] = uigetpref(...
    'mygraphics',...                        % Group
    'savefigurebeforeclosing',...           % Preference
    'Closing Figure',...                    % Window title
    {'Do you want to save your figure before closing?'
     ''
     'You can save your figure manually by typing ''hgsave(gcf)'''},...
    {'always','never';'Yes','No'},...        % Values and button strings
     'ExtraOptions','Cancel',...             % Additional button
     'DefaultButton','Cancel',...            % Default choice
     'HelpString','Help',...                 % String for Help button
     'HelpFcn','doc(''closereq'');');        % Callback for Help button
switch selectedButton
    case 'always'  % Open a Save dialog (without testing if saved before)
    [fileName,pathName,filterIndex] = uiputfile('fig', ...
                                    'Save current figure', ...
                                    'untitled.fig');
       if filterIndex == 0     % User cancelled save or named no file
           delete(fig);
       else                    % Use filename returned from UIPUTFILE
           saveas(fig,[pathName fileName])
           delete(fig);
       end
   case 'never'                % Close the figure without saving it
       delete(fig);
   case 'cancel'               % Do not close the figure
       return
 end
```

To run this example, copy and paste the code into a new program file. Name the file `save_figure_perhaps.m` and save it on your search path. To use the function as a `CloseRequestFcn` callback, create a figure as follows:

```
figure('CloseRequestFcn','save_figure_perhaps');
```

Clicking the figure's close box displays the dialog box until you select **Do not show this dialog again**.

More About

Preferences

Preferences provide the ability to specify how applications behave and how users interact with them. For example, you can set a preference for which products display their

documentation in the Help browser, or which messages the Code Analyzer displays. In MathWorks software products, preferences persist across sessions and are stored in a preference data base, the location of which is system-dependent. Use the **Preferences** option on the **File** menu to access all built-in preferences.

`uigetpref` uses the same preference data base as `addpref`, `getpref`, `ispref`, `rmpref`, and `setpref`. However, `uigetpref` registers the preferences it sets as a separate list, so that it and `uisetpref` can manage those preferences.

To modify preferences registered with `uigetpref`, you can use `setpref` and `uisetpref` to explicitly change preference values to `'ask'`. If you specify a preference that does not already exist in the preference data base, `uigetpref` creates it.

Tips

- You must supply all input arguments up to and including pref_choices.
- `uigetpref` creates specified groups and preferences, if they do not currently exist. To delete a preference group you no longer need, use `rmpref`.
- The string returned in pref_value is a preference name (as specified in pref), not its button label (as specified in pref_choices).
- After you select the check box **Do not show this dialog again** and close the dialog box, the dialog box does not display again for the same preference. To reenable dialogs that are being suppressed by preferences, use the command:

  ```
  uisetpref('clearall')
  ```

  Executing `uisetpref` with this command re-enables *all* preference dialogs you have defined with `uigetpref`, not just the most recent one.

See Also

`ispref` | `addpref` | `getpref` | `setpref` | `prefdir` | `uisetpref` | `rmpref`

Introduced before R2006a

uiimport

Import data interactively

Syntax

```
uiimport
uiimport(filename)
uiimport('-file')
uiimport('-pastespecial')
S = uiimport( ___ )
```

Description

`uiimport` opens a dialog to interactively load data from a file or the clipboard. MATLAB displays a preview of the data in the file.

`uiimport(filename)` opens the file specified in `filename`.

`uiimport('-file')` presents the file selection dialog first.

`uiimport('-pastespecial')` presents the clipboard contents first.

`S = uiimport( ___ )` stores the resulting variables as fields in the struct `S`.

More About

- "Supported File Formats for Import and Export"

See Also

`load` | `importdata` | `clipboard`

Introduced before R2006a

uimenu

Create menus and menu items on figure windows

Syntax

```
m = uimenu
m = uimenu(Name,Value,...)
m = uimenu(parent)
m = uimenu(parent,Name,Value,...)
```

Description

`m = uimenu` creates a uimenu in an existing figure's menu bar and returns the uimenu object, `m`. If there is no figure available, then MATLAB creates a new figure to serve as the parent.

`m = uimenu(Name,Value,...)` creates a uimenu and specifies one or more uimenu property names and corresponding values. Use this syntax to override the default uimenu properties.

`m = uimenu(parent)` creates a uimenu and designates a specific parent object. The `parent` argument can be a figure, uicontextmenu, or another uimenu object. Setting the parent to a uicontextmenu or another uimenu makes `m` a submenu of the parent menu.

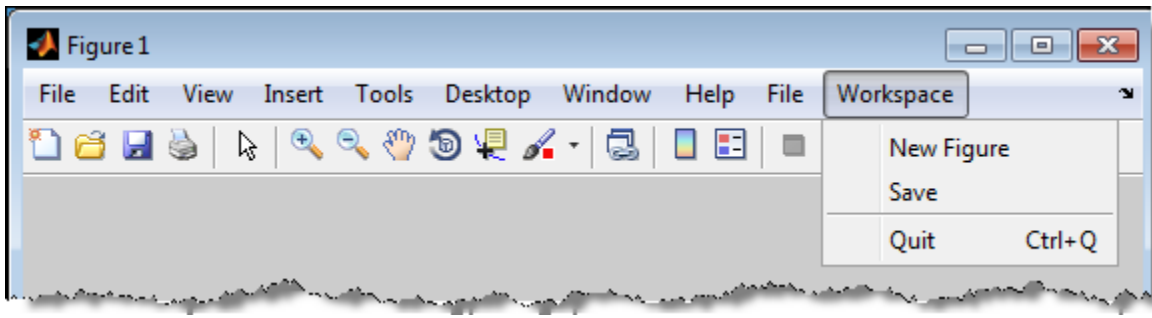
`m = uimenu(parent,Name,Value,...)` creates a uimenu with a specific parent and one or more uimenu properties.

Examples

This example creates a menu labeled **Workspace** with menu options for creating a new figure window, saving workspace variables, and exiting MATLAB. In addition, it defines an accelerator key for the Quit option.

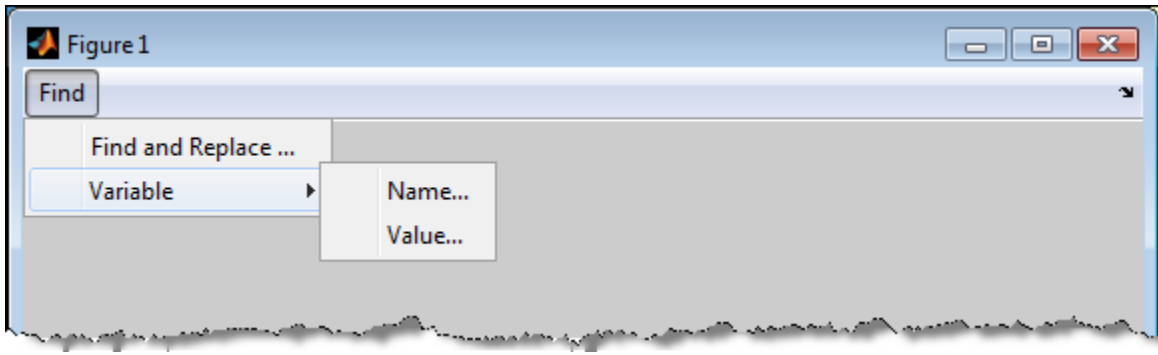
```
f = uimenu('Label','Workspace');
    uimenu(f,'Label','New Figure','Callback','disp(''figure'')');
```

```
uimenu(f,'Label','Save','Callback','disp(''save'')');
uimenu(f,'Label','Quit','Callback','disp(''exit'')',...
       'Separator','on','Accelerator','Q');
```



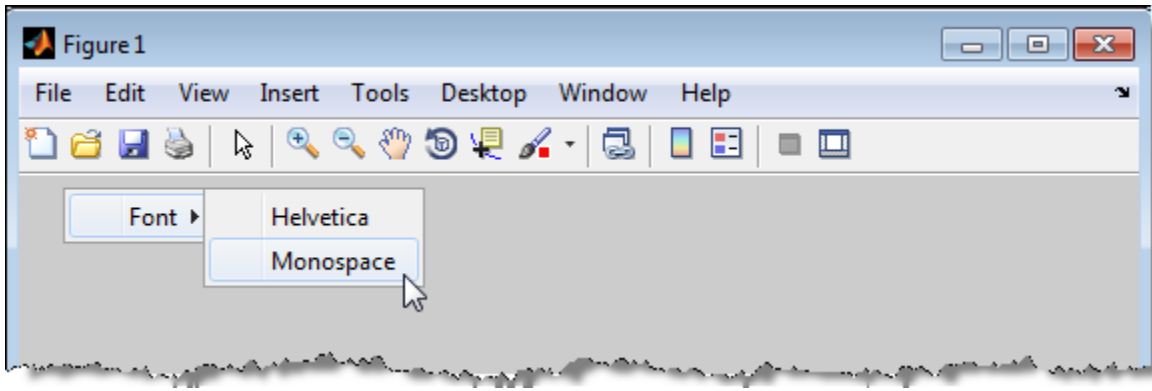
This example creates a new figure with a menu bar that excludes the built-in menus. It creates a **Find** menu with options **Find & Replace**, and **Variable**. For the Variable option, it creates a submenu with options of **Name** and **Value**.

```
f = figure('MenuBar','None');
mh = uimenu(f,'Label','Find');
frh = uimenu(mh,'Label','Find and Replace ...',...
            'Callback','disp(''goto'')');
frh = uimenu(mh,'Label','Variable');
uimenu(frh,'Label','Name...', ...
          'Callback','disp(''variable'')');

uimenu(frh,'Label','Value...', ...
          'Callback','disp(''value'')');
```



This example creates a context menu, **Font**, on a figure with menu options **Helvetica** and **Monospace**. When you run the code and then right-click anywhere within the figure window, the context menu displays.

```
f = figure;
% Create the UICONTEXTMENU
cmenu = uicontextmenu;

% Create the parent menu
fontmenu = uimenu(cmenu,'label','Font');

% Create the submenus
font1 = uimenu(fontmenu,'label','Helvetica',...
                'Callback','disp(''HelvFont'')');
font2 = uimenu(fontmenu,'label',...
                'Monospace','Callback','disp(''MonoFont'')');
f.UIContextMenu = cmenu;
```



More About

Tips

If you want the uimenu to respond when the user activates the uimenu, set its `Callback` property to be a function handle (or a string of MATLAB commands). Uimenus respond to different actions, depending on their location:

- Top level (in the menu bar) — Responds to a single mouse click

- Submenu that contains other submenus — Responds to hovering the mouse
- Terminating submenu — Responds to a single mouse click

The value of the figure's `MenuBar` property determines the content in the menu bar. When the MenuBar property is set to `'figure'`, a default set of menus appear alongside the uimenus defined in your code. When the MenuBar property is set to `'none'`, the default menus do not appear, but the uimenus you create appear in the menu bar.

Uimenus do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing uimenu children is changed to `'Modal'`, the uimenu children still exist in the `Children` property of the figure. However, the uimenus do not display while WindowStyle is set to `'Modal'`.

- "Create Menus for Programmatic UIs"
- "Create Menus for GUIDE UIs"
- "Access Property Values"

See Also

`figure` | `uicontextmenu` | Uimenu Properties

Introduced before R2006a

Uimenu Properties

Control appearance and behavior of menu

A uimenu is a menu in a figure's menu bar. The `uimenu` function adds a menu to a figure or adds a submenu to an existing menu and sets any required properties. By changing uimenu property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
m = uimenu;
color = m.ForegroundColor;
m.ForegroundColor = [0 0 1];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uimenu visibility

`'on'` (default) | `'off'`

Uimenu visibility, specified as `'on'` or `'off'`. When the Visible property is set to `'off'`, the uimenu is not visible, but you can query and set its properties.

`Label` — Menu label

string

Menu label, specified as a string. This property specifies the text that appears on the menu (or menu item).

You can specify a mnemonic for the label using the ampersand (&) character. The character that follows the ampersand appears underlined in the menu. The user can select the menu item by pressing the **Alt** key followed by the underlined character.

In order for the mnemonic system to work, you must specify a mnemonic for all menu items you define in the UI.

If you want to display an ampersand in the menu label, specify two consecutive ampersands in the string.

Here are some examples:

String	Resulting Menu Label
`'&Open Selection'`	**Open Selection** (pressing **Alt** underlines O)
`'O&pen Selection'`	**Open Selection** (pressing **Alt** underlines p)
`'&Save && Go'`	**Save & Go** (pressing **Alt** underlines S)
`'Save& Go'`	**Save& Go** (space cannot be a mnemonic character)

Note: Avoid using these case-sensitive reserved words: “default”, “remove”, and “factory”. If you must use a reserved word, then escape that word by preceding it with a back-slash character. For instance, `'\default'` escapes “default”.

Example: `'Open Selection'`

Example: `'&Open Selection'`

Data Types: `char`

`Checked` — Menu check indicator

`'off'` (default) | `'on'`

Menu check indicator, specified as `'off'` or `'on'`. Setting this property to `'on'` places a check mark next to the corresponding menu item. Setting it to `'off'` removes the check mark. You can use this feature to show the state of menu items that enable or disable functionality in your application.

`Separator` — Separator line mode

`'off'` (default) | `'on'`

Separator line mode, specified as `'off'` or `'on'`. Setting this property to `'on'` draws a dividing line above the uimenu.

`ForegroundColor` — Uimenu text color

`[0 0 0]` (default) | RGB triplet | short name | long name

Uimenu text color, specified as an RGB triplet, short name, or long name from the table below.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0 0 1]`

Example: `'b'`

Example: `'blue'`

Data Types: `double` | `char`

Location and Size

`Position` — Relative menu position

scalar integer value

Relative menu position, specified as a scalar integer value. The value of Position property indicates placement on the menu bar or within a menu. Top-level menus appear from left to right on the menu bar according to the value of their Position property, with 1 representing the left-most position. The individual items within a given menu appear from top to bottom according to the value of their Position property, with 1 representing the top-most position.

Data Types: `double`

Interactive Control

`Callback` — Callback function that executes when user selects the uimenu

`''` (default) | function handle | cell array | string

Callback function that executes when user selects the uimenu, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Note: Do not use a uimenu callback to dynamically change menu items. Deleting, adding, and replacing menu items in a callback can result in a blank menu on some platforms. You can hide, show, and disable menu items in a callback to achieve the same effect. To fully repopulate menu items, delete and create them outside the callback.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle` | `cell`

`Enable` — Operational state of uimenu

`'on'` (default) | `'off'`

Operational state of uimenu, specified as `'on'` or `'off'`. This property controls whether the user can select a menu item. When the value is `'off'`, the menu label appears dimmed, indicating that the user cannot select it.

`Accelerator` — Keyboard equivalent

character

Keyboard equivalent, specified as a character. This property defines a keyboard equivalent character for selecting the menu item. Specifying an Accelerator value

allows users to select the menu item by pressing a character and another key, instead of selecting the menu item with the mouse. The key sequence is platform specific:

- For Windows, the sequence is **Ctrl**+Accelerator. Windows reserves these keys for default menu items: `a`, `c`, `v`, and `x`
- For Macintosh systems, the sequence is **Command**+Accelerator. Apple reserves these keys for default menu items: `a`, `c`, `v`, and `x`.
- For Linux systems, the sequence is **Ctrl**+Accelerator. Linux systems reserve these keys for default menu items: `o`, `p`, `s`, and `w`. Like the other platforms, many Linux applications also use `a`, `c`, `v`, and `x`.

Accelerated menu items do not have to be displayed for the accelerator key to work. However, some restrictions apply:

- You can define the Accelerator property for only menu items that do not have submenus menus.
- Accelerators work only for menu items that execute a callback function.
- The figure must be in focus when entering the accelerator key sequence.

To remove an Accelerator value, set the Accelerator property to an empty string, `''`.

Example: `'f'`

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

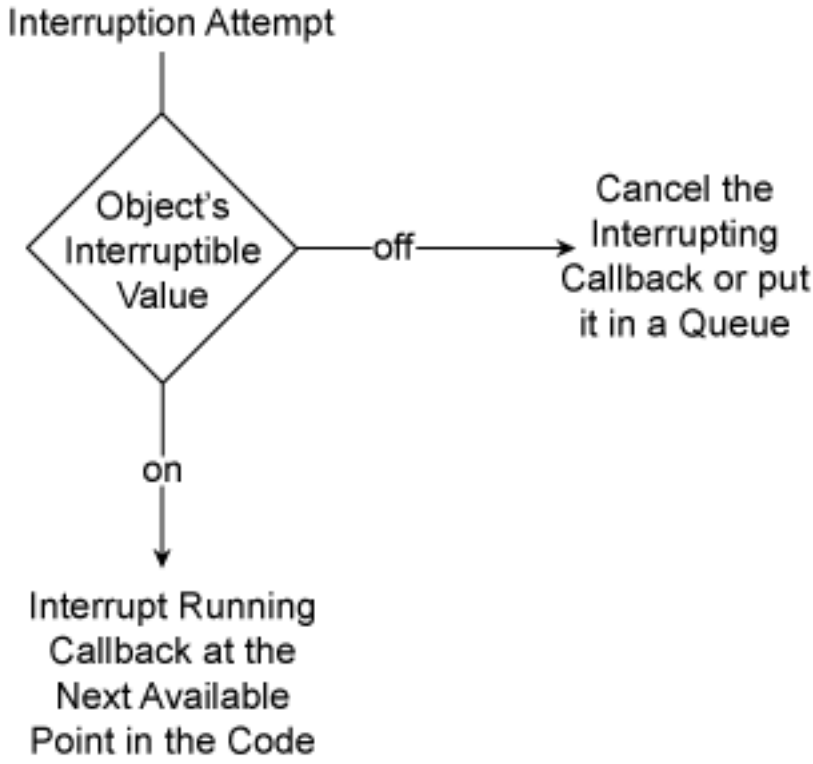
Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback

determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uimenu callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

This property has no effect on the uimenu.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uimenu

`'off'` (default) | `'on'`

Deletion status of uimenu, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uimenu begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uimenu no longer exists.

Check the value of the BeingDeleted property to verify that the uimenu is not about to be deleted before querying or modifying it.

`CreateFcn` — Uimenu creation function

function handle | cell array | string

Uimenu creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uimenu. MATLAB initializes all uimenu property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uimenu that is being created.

Setting the CreateFcn property on an existing uimenu has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uimenu object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uimenu deletion function

function handle | cell array | string

Uimenu deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uimenu (for example, when the end user deletes the figure). MATLAB executes the

DeleteFcn callback before destroying the properties of the uimenu. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uimenu that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object

`'uimenu'`

Type of graphics object, returned as `'uimenu'`.

`Tag` — Uimenu identifier

`''` (default) | string

Uimenu identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uimenu. When you need access to the uimenu elsewhere in your code, you can use the `findobj` function to search for the uimenu based on the Tag value.

Example: `'menu1'`

Data Types: `char`

`UserData` — Data to associate with the uimenu object

empty array (default) | array

Data to associate with the uimenu object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uimenu parent

figure | uicontextmenu | uimenu

Uimenu parent, specified as a figure, uicontextmenu, or another uimenu object. You can move a uimenu to a different figure, or move it under a different uimenu by setting this property to the target figure, uicontextmenu, or uimenu object.

`Children` — Children of uimenu

empty `GraphicsPlaceholder` array (default) | 1-D array of uimenu objects

Children of uimenu, returned as an empty `GraphicsPlaceholder` or a 1-D array of uimenu objects. The children of uimenus are other uimenus that function as submenus.

You cannot add or remove children using the `Children` property of the uimenu. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the displayed menu items.

To add a child to this list, set the `Parent` property of the child component to be the uimenu object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of Uimenu handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uimenu handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uimenu handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uimenu handle is always visible.

HandleVisibility Value	Description
`'callback'`	The uimenu handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uimenu at the command-line, but allows callback functions to access it.
`'off'`	The uimenu handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`uimenu`

More About

- "Default Property Values"

uint8

Convert to 8-bit unsigned integer

Syntax

```
intArray = uint8(array)
```

Description

`intArray = uint8(array)` converts the elements of an array into unsigned 8-bit (1-byte) integers of class `uint8`.

Input Arguments

`array`

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint8`, the `uint8` function has no effect.

Output Arguments

`intArray`

Array of class `uint8`. Values range from 0 to $2^8 - 1$.

The `uint8` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint8(2^8)   %  2^8 = 256
```

returns

```
ans =
    255
```

Examples

Convert a `double` array to `uint8`:

```
mydata = uint8(magic(10));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint8(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint8');   % Preferred
```

See Also

`double` | `single` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Introduced before R2006a

uint16

Convert to 16-bit unsigned integer

Syntax

```
intArray = uint16(array)
```

Description

`intArray = uint16(array)` converts the elements of an array into unsigned 16-bit (2-byte) integers of class `uint16`.

Input Arguments

`array`

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint16`, the `uint16` function has no effect.

Output Arguments

`intArray`

Array of class `uint16`. Values range from 0 to $2^{16} - 1$.

The `uint16` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint16(2^16)   %  2^16 = 65536
```

returns

```
ans =
    65535
```

Examples

Convert a `double` array to `uint16`:

```
mydata = uint16(magic(100));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint16(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint16');   % Preferred
```

See Also

`double` | `single` | `uint8` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Introduced before R2006a

uint32

Convert to 32-bit unsigned integer

Syntax

```
intArray = uint32(array)
```

Description

`intArray = uint32(array)` converts the elements of an array into unsigned 32-bit (4-byte) integers of class `uint32`.

Input Arguments

`array`

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint32`, the `uint32` function has no effect.

Output Arguments

`intArray`

Array of class `uint32`. Values range from 0 to $2^{32} - 1$.

The `uint32` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint32(2^32)    % 2^32 = 4294967296
```

returns

```
ans =
    4294967295
```

Examples

Convert a `double` array to `uint32`:

```
mydata = uint32(magic(1000));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint32(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint32');   % Preferred
```

See Also

`double` | `single` | `uint8` | `uint16` | `uint64` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Introduced before R2006a

uint64

Convert to 64-bit unsigned integer

Syntax

```
intArray = uint64(array)
```

Description

`intArray = uint64(array)` converts the elements of an array into unsigned 64-bit (8-byte) integers of class `uint64`.

Input Arguments

`array`

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint64`, the `uint64` function has no effect.

Output Arguments

`intArray`

Array of class `uint64`. Values range from 0 to $2^{64} - 1$.

The `uint64` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint64(2^64)    % 2^64 = 18446744073709551616
```

returns

```
ans =
    18446744073709551615
```

Examples

Convert a literal value to `uint64`:

```
x = uint64(9007199254740993);
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint64(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint64');   % Preferred
```

More About

Tips

Double-precision floating-point numbers have only 52 bits in the mantissa. Therefore, `double` values cannot represent all integers greater than 2^{53} correctly. Before performing arithmetic operations on values larger than 2^{53} in magnitude, convert the values to 64-bit integers. For example,

```
x = uint64(2^53+1);    % Floating-point arithmetic, loses precision
```

is not as accurate as the 64-bit integer arithmetic operation:

```
x = uint64(2^53) + 1;  % Preferred
```

See Also

`double` | `single` | `uint8` | `uint16` | `uint32` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Introduced before R2006a

uiopen

Open dialog box for selecting file to load into workspace

Syntax

```
uiopen
uiopen(type)
uiopen(filename)
uiopen(filename, TF)
```

Description

`uiopen` displays the dialog box with the file filter set to all MATLAB files (with file extensions `*.m`, `*.mat`, `*.fig`, `*.mdl`, and `*.slx`).

`uiopen(type)` sets the file filter according to the `type`.

`uiopen(filename)` displays `filename` as the default value for **File name** in the dialog box and lists only files having the same extension.

`uiopen(filename, TF)` directly opens file `filename` without displaying a dialog box if `TF` is true, and displays the dialog box if `TF` is false.

Input Arguments

`type`

String that specifies the kind of file to show in the dialog box (the file filter). Acceptable values for type are the following.

Type string	Files Displayed
`matlab`	All MATLAB files (with file extensions `*.m`, `*.mat`, `*.fig`, `*.mdl`, and `*.slx`.)

Type string	Files Displayed
`load`	All MAT-files (`*.mat`)
`figure`	All figure files (`*.fig`)
`simulink`	All Simulink model files (`*.mdl` and `*.slx`)
`editor`	All MATLAB files except for `.mat`, `.fig`, and `.slx` files.

Default:

`filename`

A string, including the file extension, naming a file to open. The filename can be a wildcard character plus extension. For example, `*.txt` displays a list of all files with the file extension `.txt`.

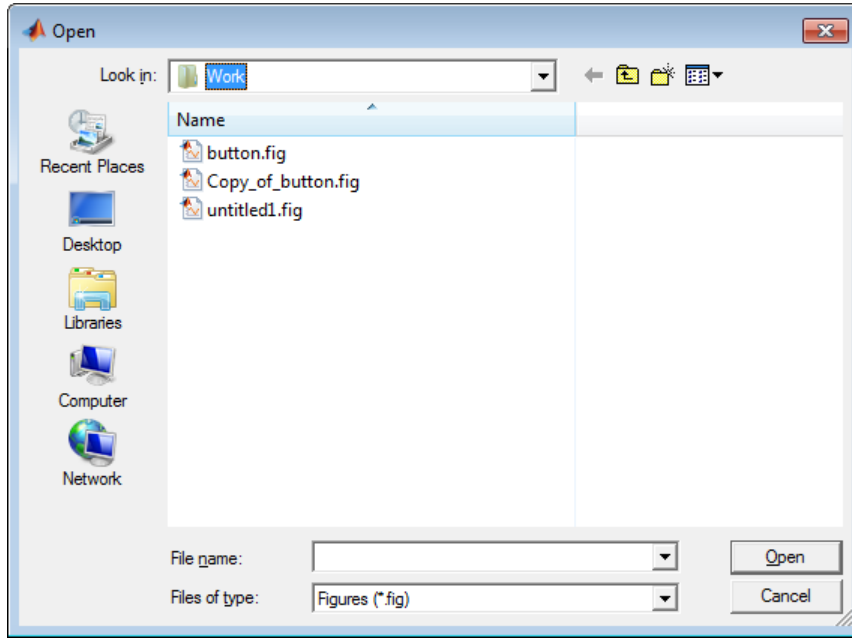
`TF`

A MATLAB expression that evaluates to `true` or `false`. If `true`, filename opens directly, without displaying the dialog.

Examples

Filter to display only FIG-files by setting the **Files of type** field to `Figures (*.fig)`:

```
uiopen('figure')
```



Alternatives

In MATLAB code or in a command:

- To open a file appropriately based on its file extension, use the `open` function.
- To open a file in the Editor, use the `edit` function.
- To open a MAT-file and load its contents into the workspace, use the `load` function.
- To open a FIG-file, use the `openfig` function.
- To open a file in an application in Microsoft Windows, use the `winopen` function.

More About

Modal Dialog

A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

Tips

When you select a file and click **open**, `uiopen` does the following:

- Gets the file using `uigetfile`.
- Opens the file using the `open` command.
  - Files with a file extension of `.m` open in the Editor.
  - Variables stored in files with a file extension of `.mat` appear in the caller's workspace.
  - Files with a file extension of `.fig` open as figure windows.
  - Files with a file extension of `.mdl` or `.slx` open as models in Simulink.
- `uiopen('load')` is the only the form of `uiopen` that you can compile into a standalone application. You can create a file selection dialog box that you can compile using `uigetfile`.
- The `uiopen` dialog box is modal. A modal dialog box prevents you from interacting with other windows until you respond to the modal one.
- `uiopen` displays the same dialog box that opens when you use the MATLAB desktop toolstrip to open a file. (On the **Home** tab, in the **File** section, click **Open**  .)

See Also

`uigetfile` | `uiputfile` | `uisave`

Introduced before R2006a

uipanel

Create panel container object

Syntax

```
p = uipanel
p = uipanel(Name,Value,...)
p = uipanel(parent)
p = uipanel(parent,Name,Value,...)
```

Description

`p = uipanel` creates a uipanel in an existing figure and returns the uipanel object, `p`. If there is no figure available, then MATLAB creates a new figure to serve as the parent.

`p = uipanel(Name,Value,...)` creates a uipanel and specifies one or more uipanel property names and corresponding values. Use this syntax to override the default uipanel properties.

`p = uipanel(parent)` creates a uipanel and designates a specific parent object. The `parent` argument can be a figure, uitab, uibuttongroup or another uipanel object.

`p = uipanel(parent,Name,Value,...)` creates a uipanel with a specific parent and one or more uipanel properties.

Uipanels are containers that group UI components together. Uipanels can contain uicontrols, axes, uitables, uitabs, uibuttongroups, and other uipanels. Uipanels cannot contain ActiveX controls.

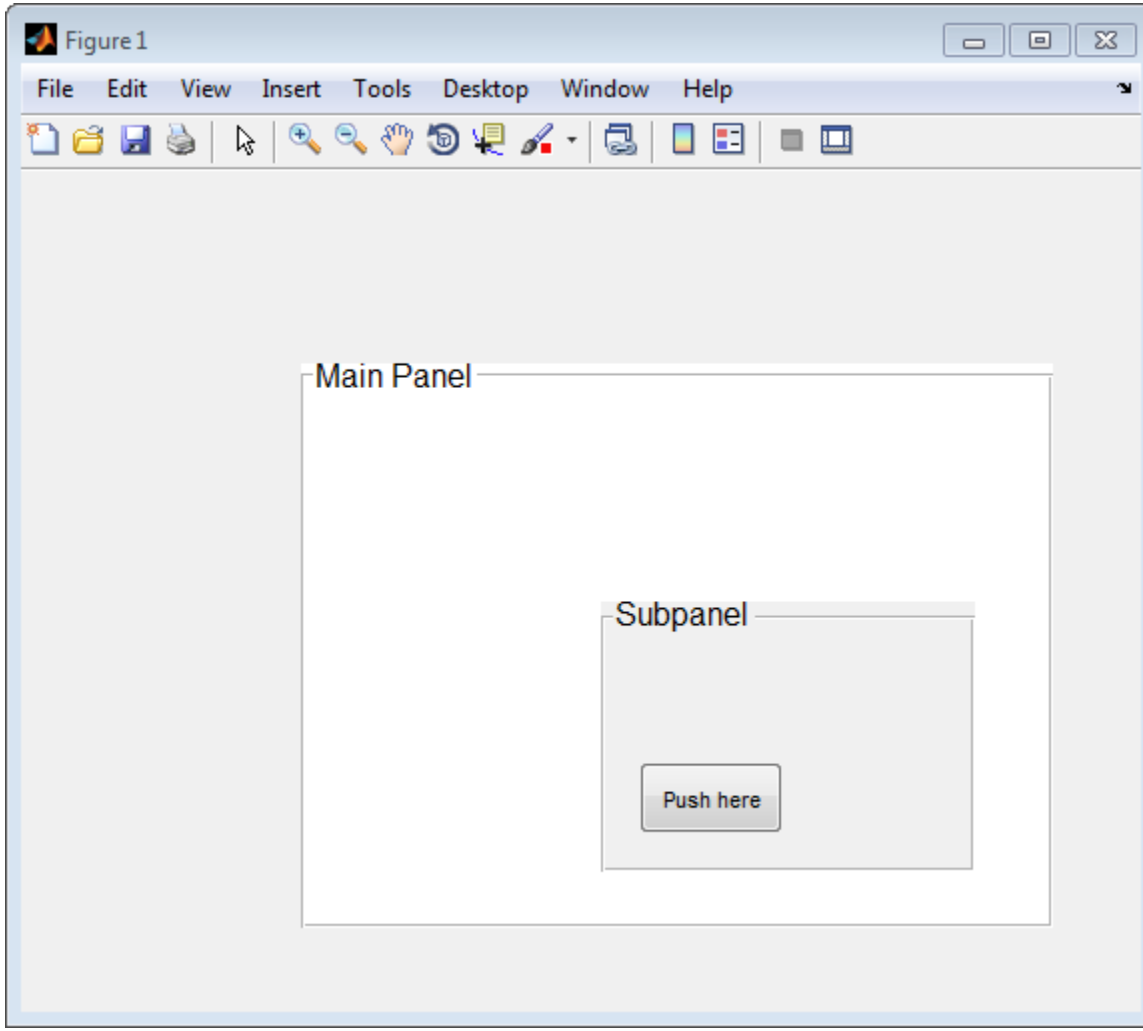
Examples

Create uipanel With Children

Create a figure containing two panels and a push button. The panels use the default Units property value, `'normalized'`. The default units for the `uicontrol` is `'pixels'`.

```
h = figure;
hp = uipanel('Title','Main Panel','FontSize',12,...
```

```
                'BackgroundColor','white',...
                'Position',[.25 .1 .67 .67]);
hsp = uipanel('Parent',hp,'Title','Subpanel','FontSize',12,...
                'Position',[.4 .1 .5 .5]);
hbsp = uicontrol('Parent',hsp,'String','Push here',...
                'Position',[18 18 72 36]);
```



Uipanel That Always Matches Figure Window Width

Create a uipanel that always matches the width of the figure, but maintains a constant height. Use the SizeChangedFcn callback to keep the panel's width equal to the width of the figure's drawable area.

To see how this code works, copy and paste this code into the Editor and run it.

```
function myui
f = figure('Visible','off','SizeChangedFcn',@match_width);
u = uipanel('Units','Pixels', 'Tag','StatusBar',...
            'BackgroundColor','blue');
f.Visible= 'on';
    function match_width(hObject,eventdata)
       fig = gcbo;
       u = findobj(fig,'Tag','StatusBar');

       % change panel units to pixels and adjust position
       panelunits = u.Units;
       u.Units = 'pixels';
       figpos = fig.Position;
       upos = [1, figpos(4) - 20, figpos(3), 20];
       u.Position = upos;

       % restore original panel units
       u.Units = panelunits;

    end
end
```

More About

Tips

If you set the Visible property of a `uipanel` object to `'off'`, then any child objects it contains (buttons, button groups, axes, etc.) become invisible along with the parent uipanel. However, the Visible *property value* of each child object remains unaffected.

- "Access Property Values"

See Also

`figure` | `uibuttongroup` | `uicontrol` | Uipanel Properties

Introduced before R2006a

Uipanel Properties

Control appearance and behavior of panel

Uipanels are containers for grouping together graphics objects or UI components. The `uipanel` function creates a panel in a figure and sets any required properties before displaying it. By changing the uipanel property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
p = uipanel;
pos = p.Position;
p.Position = [.1 .1 .7 .8];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uipanel visibility

`'on'` (default) | `'off'`

Uipanel visibility, specified as `'on'` or `'off'`. The Visible property determines whether the uipanel displays on the screen. If the Visible property of a uipanel is set to `'off'`, the entire uipanel is invisible, but you can still specify and access its properties.

Changing the size of an invisible uipanel triggers the SizeChangedFcn callback when the uipanel becomes visible.

Note Changing the Visible property of a uipanel does *not* change the Visible property of its child components even though hiding the uipanel prevents its children from displaying.

`BackgroundColor` — Uipanel background color

`[.94 .94 .94]` (default) | RGB triplet | short name | long name

Uipanel background color, specified as an RGB triplet, short name, or long name. The color you specify fills the area bounded by the uipanel.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Note: Some operating systems override the `BackgroundColor` value.

Data Types: `double` | `char`

`ForegroundColor` — Uipanel title color

`[0 0 0]` (default) | RGB triplet | short name | long name

Uipanel title color, specified as an RGB triplet, short name, or long name.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`

Long Name	Short Name	RGB Triplet
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0 0 1]

Example: 'b'

Example: 'blue'

BorderType — Border of the uipanel

'etchedin' (default) | 'etchedout' | 'beveledin' | 'beveledout' | 'line' | 'none'

Border of the uipanel area, specified as 'etchedin', 'none', 'etchedout', 'beveledin', 'beveledout', or 'line'.

- For a 3-D look, use etched or beveled borders.

  Use the HighlightColor and ShadowColor properties to specify the color of 3-D borders.
- For a 2-D look, use a line border.

  Use the ForegroundColor property to specify the line border color.

BorderWidth — Width of uipanel border

1 (default) | positive integer value

Width of the uipanel border, specified as a positive integer value. The unit of measurement is pixels. Etched and beveled borders wider than three pixels might not appear correctly at the corners.

HighlightColor — Uipanel 3-D frame highlight color

RGB triplet | short name | long name

Uipanel 3-D frame highlight color, specified as an RGB triplet, short name, or long name.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

[0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

ShadowColor — Uipanel border shadow color

RGB triplet | short name | long name

Uipanel border shadow color, specified as an RGB triplet, short name, or long name.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Clipping — Clipping of child components to uipanel

'on' (default) | 'off'

Note: The behavior of the Clipping property has changed. It no longer has any effect on uipanels. Child objects are now clipped to the uipanel boundary regardless of the value of this property. This property might be removed in a future release.

Location and Size

`Position` — Location and size of uipanel relative to parent

`[left bottom width height]`

Location and size of the uipanel relative to the parent, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the uipanel
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the uipanel
`width`	Distance between the right and left outer edges of the uipanel
`height`	Distance between the top and bottom outer edges of the uipanel

All measurements are in units specified by the Units property.

Note: If the parent of the uipanel is a figure, then the Position values are relative to the figure's *drawable area*. The drawable area of a figure is the area inside the window borders, excluding the menu bar and tool bar.

Example: Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the Position vector. For example, this code changes the width of the uipanel to `0.5`:

```
p = uipanel;
p.Position(3) = 0.5;
p.Position
```

```
ans =
```

```
      0    0    0.5000   1.0000
```

Data Types: `double`

`Units` — Units of measurement

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'pixels'` | `'points'` | `'characters'`

Units of measurement, specified as `'normalized'`, `'inches'`, `'centimeters'`, `'pixels'`, `'points'`, or `'characters'`.

MATLAB uses these units to interpret the location and size values of the Position property:

- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- The size of a uipanel specified in pixel units depends on the system display settings and resolution.
- Characters units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the Units property before the Position property, then MATLAB sets Position using the units you specified.
- If you specify the Units property after the Position property, MATLAB sets the position using the default Units. Then, MATLAB converts the Position values to the equivalent values in the units you specified.

`SizeChangedFcn` — Uipanel resize callback function

`''` (default) | function handle | cell array | string

Uipanel resize callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Define this callback function to control the UI layout when the size of the uipanel changes.

The SizeChangedFcn callback executes under these circumstances:

- The uipanel becomes visible for the first time.
- The uipanel is visible while its drawable area changes. The drawable area is the area inside the outer bounds of the uipanel.
- The uipanel becomes visible for the first time after its drawable area changes. This situation occurs when the drawable area changes while the uipanel is invisible, and then it becomes visible later.

These are some of the important characteristics of the SizeChangedFcn callback and some recommended best practices:

- Consider delaying the display of the figure until after all the variables that the uipanel's SizeChangedFcn uses are defined. This practice can prevent the uipanel's SizeChangedFcn callback from returning an error. To delay the display of the figure, set its Visible property to `'off'`. Then, set the Visible property to `'on'` after you define the variables that your SizeChangedFcn callback uses.
- Use the `gcbo` function in your SizeChangedFcn code to get the uipanel object that the user is resizing. See the description of the figure `SizeChangedFcn` for an example that uses `gcbo`.
- All visible Uipanels that are specified in normalized units resize before the parent figure does. All visible, nested uipanels resize from inside-out.

Tip As an easy alternative to specifying a SizeChangedFcn callback, you can set the Units property of all the objects you put inside the uipanel to `'normalized'`. Doing so makes those components scale proportionally with the uipanel.

See "Managing the Layout in Resizable UIs" for more information on controlling the resize behavior of programmatic UIs.

See Resize Behavior for more information on the resize behavior of GUIDE UIs.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`ResizeFcn` — Uipanel resize callback function

`''` (default) | function handle | cell array | string

Callback function that executes when the uipanel size changes, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Note: Use of the ResizeFcn property is not recommended. It might be removed in a future release. Use SizeChangedFcn instead.

Data Types: `function_handle | cell | char`

Font Style

`FontName` — Font for displaying uipanel title

string

Font for displaying the uipanel title, specified as a string. To display and print properly, the font name must refer to a font that the user's system supports. The default font is system dependent.

To use a fixed-width font, set the FontName property to the string, `'FixedWidth'`. This setting instructs MATLAB to use the value of the root FixedWidthFontName property, which you can set in the `startup.m` file.

See "Startup Options in MATLAB Startup File" for more information about the `startup.m` file.

Example: `'Arial'`

`FontSize` — Font size for uipanel title

positive number

Font size for the uipanel title, specified as a positive number. The FontUnits property specifies the units. The default size is system-dependent.

Example: `12`

Example: `12.5`

Data Types: `double`

`FontUnits` — Units of font size for uipanel title

`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Units of font size for the uipanel title, specified as `'points'`, `'inches'`, `'centimeters'`, `'normalized'`, or `'pixels'`.

If you set this property to `'normalized'`, then MATLAB interprets the font size as a fraction of the uipanel height. When you resize the uipanel, MATLAB scales the displayed font to maintain that fraction.

The other FontUnits options (`pixels`, `inches`, `centimeters`, and `points`) are absolute units. 1 point = 1/72 inch.

`FontWeight` — Font weight for the uipanel title

`'normal'` (default) | `'bold'`

Font weight of uipanel title, specified as a value from the following table.

FontWeight Value	Description
`'normal'`	Normal font weight
`'bold'`	Heavy font weight

Not all fonts support all font weights. Therefore, if you specify an unsupported value for the `FontWeight` property, the result might appear the same as the default.

Note: The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

FontAngle — Character slant of the uipanel title

'normal' (default) | 'italic'

Character slant of uipanel title, specified as 'normal' or 'italic'. MATLAB uses this property to select a font from those available on your system. Setting this property to 'italic' selects a slanted version of the font, if it is available on your system.

Note: The 'oblique' value has been removed. Use 'italic' instead.

Text

Title — Uipanel title

string

Uipanel title, specified as a string.

MATLAB does not interpret a vertical slash ('|') character as a line break, it displays as a vertical slash in the uipanel title.

If you want to specify a Unicode character, pass the Unicode decimal code to the char function. For example, ['Multiples of ' char(960)] displays as Multiples of π.

Example: 'Options'

Example: ['Multiples of ' char(960)]

TitlePosition — Location of uipanel title

'lefttop' (default) | 'centertop' | 'righttop' | 'leftbottom' | 'centerbottom' | 'righttbottom'

Location of the uipanel title, specified as 'lefttop', 'centertop', 'righttop', 'leftbottom', 'centerbottom', or 'righttbottom'.

Interactive Control

ButtonDownFcn — Button-press callback function

'' (default) | function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The ButtonDownFcn callback is a function that executes when the user clicks a mouse button on the uipanel.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`UIContextMenu` — Uipanel context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Uipanel context menu, specified as a uicontextmenu object. Use this property to display a context menu when the user right-clicks on the uipanel. Create the context menu using the `uicontextmenu` function.

`Selected` — Selection status of uipanel

`'off'` (default) | `'on'`

Note: The behavior of the Selected property changed in R2014b, and it is not recommended. It no longer has any effect on uipanels. This property might be removed in a future release.

`SelectionHighlight` — Ability to highlight selection handles

`'on'` (default) | `'off'`

Note: The behavior of the SelectionHighlight property changed in R2014b, and it is not recommended. It no longer has any effect on uipanels. This property might be removed in a future release.

Callback Execution Control

`Interruptible` — Callback interruption

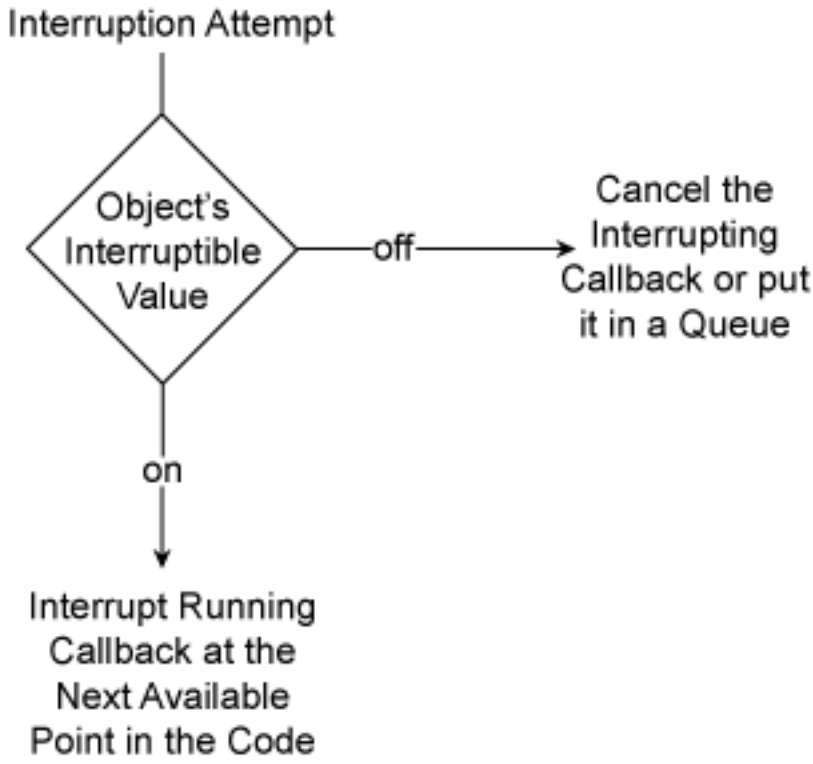
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uipanel callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See "Interrupt Callback Execution" for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See "Interrupt Callback Execution" for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

Ability to become current object, specified as `'on'` or `'off'`:

- Setting the value to `'on'` allows the uipanel to become the current object when the user clicks on it. A value of `'on'` also allows the figure CurrentObject property and the `gco` function to report the uipanel as the current object.
- Setting the value to `'off'` sets the figure CurrentObject property to an empty `GraphicsPlaceholder` array when the user clicks on the uipanel.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uipanel

`'off'` (default) | `'on'`

Deletion status of uipanel, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uipanel begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uipanel no longer exists.

Check the value of the BeingDeleted property to verify that the uipanel is not about to be deleted before querying or modifying it.

`CreateFcn` — Uipanel creation function

function handle | cell array | string

Uipanel creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uipanel. MATLAB initializes all uipanel property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uipanel that is being created.

Setting the CreateFcn property on an existing uipanel has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uipanel object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uipanel deletion function

function handle | cell array | string

Uipanel deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uipanel (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the uipanel. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uipanel that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object
`'uipanel'`

Type of graphics object, returned as `'uipanel'`.

`Tag` — Uipanel identifier
`''` (default) | string

Uipanel identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uipanel. When you need access to the uipanel elsewhere in your code, you can use the `findobj` function to search for the uipanel based on the Tag value.

Example: `'panel1'`

Data Types: `char`

`UserData` — Data to associate with the uipanel object
empty array (default) | array

Data to associate with the uipanel object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uipanel parent

figure | uipanel | uibuttongroup | uitab

Uipanel parent, specified as a figure, uipanel, uibuttongroup, or uitab. You can move a uipanel to a different figure, uipanel, uibuttongroup, or uitab by setting this property to the handle of the target figure, uipanel, uibuttongroup, or uitab.

`Children` — Children of uipanel

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Children of uipanel, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of uipanel can be axes, uipanels, uibuttongroups, and uicontrols.

You cannot add or remove children using the `Children` property of the uipanel. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the front-to-back order (stacking order) of the components on the screen.

To add a child to this list, set the `Parent` property of the child component to be the uipanel object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of Uipanel handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uipanel handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uipanel handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not

visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uipanel handle is always visible.
`'callback'`	The uipanel handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uipanel at the command-line, but allows callback functions to access it.
`'off'`	The uipanel handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`figure` | `uipanel`

More About

- “Access Property Values”
- “Default Property Values”

uipushtool

Create push button on toolbar

Syntax

```
p = uipushtool
p = uipushtool(Name,Value,...)
p = uipushtool(parent)
p = uipushtool(parent,Name,Value,...)
```

Description

`p = uipushtool` creates a uipushtool in the current figure's uitoolbar and returns the uipushtool object, `p`. If there is no uitoolbar available, then MATLAB creates a new uitoolbar in the current figure to serve as the parent. Similarly, if there no figure is available, then MATLAB creates a new figure with a uitoolbar.

`p = uipushtool(Name,Value,...)` creates a uipushtool and specifies one or more uipushtool property names and corresponding values. Use this syntax to override the default uipushtool properties.

`p = uipushtool(parent)` creates a uipushtool and designates a specific parent object. The `parent` argument must be a uitoolbar object.

`p = uipushtool(parent,Name,Value,...)` creates a uipushtool with a specific parent and one or more uipushtool properties.

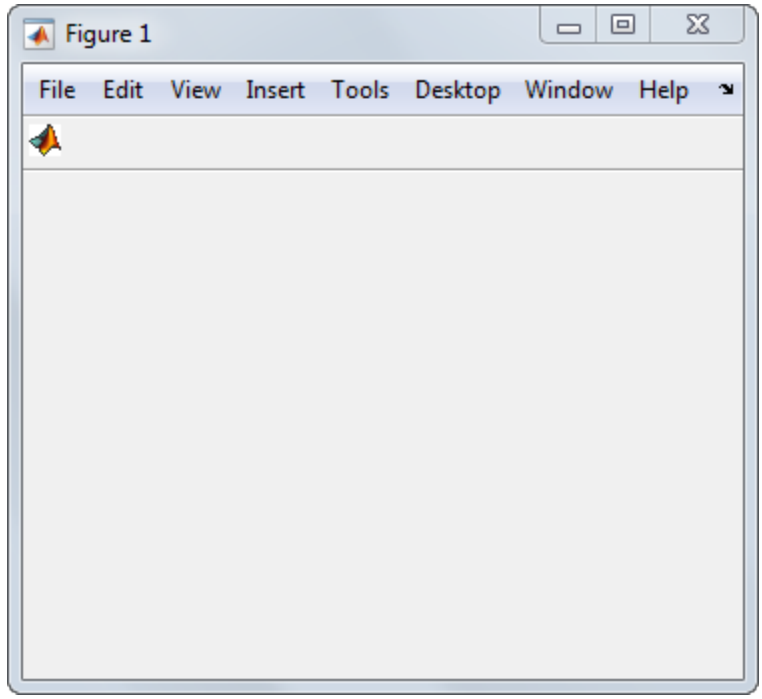
A uipushtool is a push button that appears in the figure's tool bar. The button has no icon, but its borders highlight when the user hovers over it with the mouse. You can create a button icon by setting the uipushtool's `CData` property.

Examples

Create uipushtool With Icon and Tool Tip

```
f  = figure('ToolBar','none');
```

```
t = uitoolbar(f);

% Read an image
[img,map] = imread(fullfile(matlabroot,...
            'toolbox','matlab','icons','matlabicon.gif'));

% Convert image from indexed to truecolor
icon = ind2rgb(img,map);

% Create a uipushtool in the toolbar
p = uipushtool(t,'TooltipString','Toolbar push button',...
                 'ClickedCallback',...
                 'disp(''Clicked uipushtool.'')');

% Set the button icon
p.CData = icon;
```



More About

Tips

- Uitoolbars (and their child uipushtools) do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a uitoolbar is changed to `'Modal'`, the uitoolbar still exists in the `Children` property of the figure. However, the uitoolbar does not display while WindowStyle is set to `'Modal'`.
- Unlike uicontrol push buttons, uipushtools do not set the figure's `SelectionType` property to `'open'` on the second click.

- "Access Property Values"
- "Create Toolbars for Programmatic UIs"

See Also

Uipushtool Properties | `uitoggletool` | `uitoolbar`

Introduced before R2006a

Uipushtool Properties

Control appearance and behavior of uipushtool

Uipushtools are push buttons that appear on the tool bar at the top of the a figure. The `uipushtool` function creates a push button on a tool bar and sets any required properties before displaying it. By changing uipushtool property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
p = uipushtool;
sep = p.Separator;
p.Separator = 'on';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uipushtool visibility

`'on'` (default) | `'off'`

Uipushtool visibility, specified as `'on'` or `'off'`. When the Visible property is set to `'off'`, the uipushtool is not visible, but you can query and set its properties.

`Separator` — Separator line mode

`'off'` (default) | `'on'`

Separator line mode, specified as `'off'` or `'on'`. Setting this property to `'on'` draws a dividing line to left of the uipushtool.

`CData` — Optional image to display on uipushtool

3-D array of truecolor RGB values

Optional image to display on the uipushtool, specified as a 3-D array of truecolor RGB values. The values in the array can be:

- Double-precision values between `0.0` and `1.0`
- `uint8` values between `0` and `255`

The length of the array's first and second dimensions must be less than or equal to 16. Otherwise, it might be clipped or distorted when it displays.

Data Types: `double` | `uint8`

Interactive Control

`ClickedCallback` — Callback function that executes when user clicks uipushtool

`''` (default) | function handle | cell array | string

Callback function that executes when a user clicks the uipushtool, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying callback functions as function handles, cell arrays, or strings, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle` | `cell` | `char`

`Enable` — Operational state of uipushtool

`'on'` (default) | `'off'`

Operational state of uipushtool, specified as `'on'` or `'off'`. The `Enable` property controls whether the uipushtool responds to button clicks. The are two possible values:

- `'on'` – The uipushtool is operational.
- `'off'` – The uipushtool is not operational and appears grayed out.

The value of the `Enable` property and the type of button click determine the response.

Enable Value	Response to Left-Click	Response to Right-Click
`'on'`	The uipushtool's `ClickedCallback` function executes.	The uipushtool is not operational. No callback executes.

Enable Value	Response to Left-Click	Response to Right-Click
`'off'`	The uipushtool is not operational. No callback executes.	The uipushtool is not operational. No callback executes.

`TooltipString` — Tooltip text

string

Tooltip text, specified as a string. When the user hovers the mouse pointer over the uipushtool and leaves it there, the tooltip appears.

Example: `'Some string'`

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu handle

This property has no effect on uipushtools.

Callback Execution Control

`Interruptible` — Callback interruption

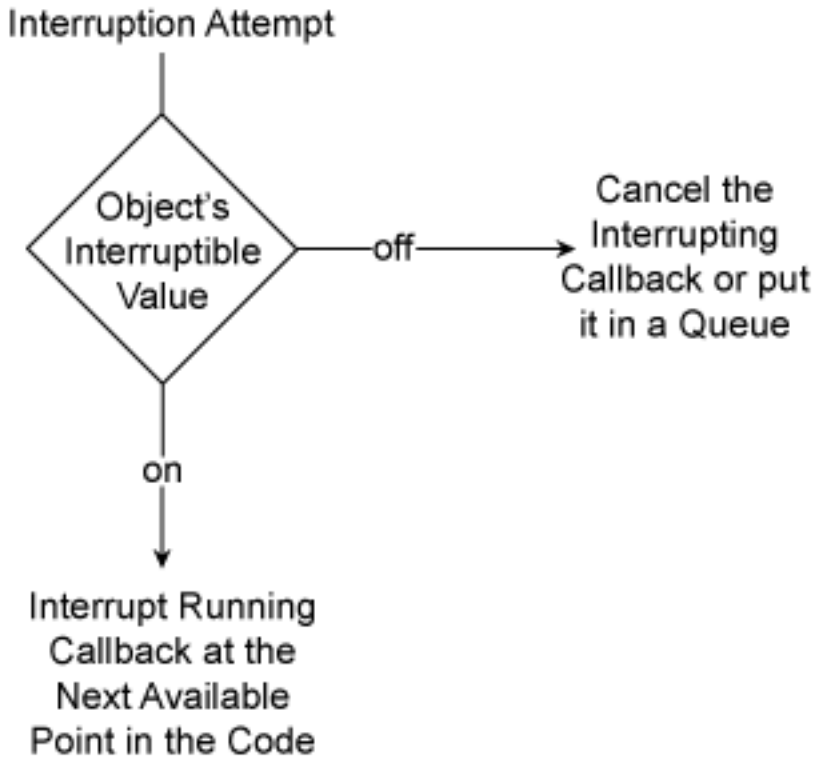
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uipushtool callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

This property has no effect on the uipushtool.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uipushtool

`'off'` (default) | `'on'`

Deletion status of uipushtool, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uipushtool begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uipushtool no longer exists.

Check the value of the BeingDeleted property to verify that the uipushtool is not about to be deleted before querying or modifying it.

`CreateFcn` — Uipushtool creation function

function handle | cell array | string

Uipushtool creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uipushtool. MATLAB initializes all uipushtool property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uipushtool that is being created.

Setting the CreateFcn property on an existing uipushtool has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uipushtool object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uipushtool deletion function

function handle | cell array | string

Uipushtool deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uipushtool (for example, when the end user deletes the figure). MATLAB executes

the DeleteFcn callback before destroying the properties of the uipushtool. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uipushtool that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object

`'uipushtool'`

Type of graphics object, returned as `'uipushtool'`.

`Tag` — Uipushtool identifier

`''` (default) | string

Uipushtool identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uipushtool. When you need access to the uipushtool elsewhere in your code, you can use the `findobj` function to search for the uipushtool based on the Tag value.

Example: `'pushtool1'`

Data Types: `char`

`UserData` — Data to associate with the uipushtool object

empty array (default) | array

Data to associate with the uipushtool object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uipushtool parent

uitoolbar

Uipushtool parent, specified as a uitoolbar. You can move a uipushtool to a different uitoolbar by setting this property to the handle of the target uitoolbar.

`HandleVisibility` — Visibility of Uipushtool handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uipushtool handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uipushtool handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uipushtool handle is always visible.
`'callback'`	The uipushtool handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uipushtool at the command-line, but allows callback functions to access it.
`'off'`	The uipushtool handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`uipushtool` | `uitoolbar`

More About

- "Access Property Values"
- "Default Property Values"

uiputfile

Open dialog box for saving files

Syntax

```
FileName = uiputfile
[FileName,PathName] = uiputfile
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec)
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle)
[FileName,PathName,FilterIndex] =
uiputfile(FilterSpec,DialogTitle,DefaultName)
```

Description

Note: Successful execution of `uiputfile` does not create a file; it only returns the name of a new or existing file that you designate.

`FileName = uiputfile` displays a modal dialog box for selecting or specifying a file you want to create or save. The dialog box lists the files and folders in the current folder. If the selected or specified file name is valid, `uiputfile` returns that name in `FileName`.

`[FileName,PathName] = uiputfile` also returns the path to `FileName` in `PathName`, or if you cancel the dialog, returns `0` for both arguments. If you do not provide any output arguments, the file name alone is returned in `ans`.

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec)` displays only those files with extensions that match `FilterSpec`. On some platforms `uiputfile` also displays in gray any files that do not match `FilterSpec`. The `uiputfile` function appends `'All Files'` to the list of file types. `FilterSpec` can be a string or a cell array of strings, and can include the `*` and `?` wildcard characters. For example, `'*.m'` lists all MATLAB program files in a folder.

`FilterSpec` can be a string that contains a file name. `uiputfile` displays the file name selected in the **File name** field and uses the file extension as the default filter.

The `FilterSpec` string can include a path, or consist of a path only. To specify a folder only, make the last character in `DefaultName` `'\'` or `'/'`. A path can contain special path characters, such as `'.'`, `'..'`, `'/'`, `'\'`, or `'~'`. For example, `'../*.m'` lists all program files in the folder above the current folder. If `FilterSpec` is a cell array of strings, the first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace the default descriptions in the **Save as type** pop-up menu. A description cannot be an empty string. See the "Examples" on page 1-8549 for illustration of using cell arrays as `FilterSpec`. If you do not specify `FilterSpec`, `uiputfile` uses the default list of file types (all MATLAB files). `FilterIndex` is the index of the filter selected in the dialog box. Indexing starts at 1. If you click the **Cancel** button, the window's close box, or if the file does not exist, `uiputfile` returns `FilterIndex` as `0`.

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle)` displays a dialog box that has the title `DialogTitle`. To use the default file types and to specify a dialog title, enter `uiputfile('','DialogTitle')`

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle,DefaultName)` displays a dialog box in which the file name specified by `DefaultName` appears in the **File name** field. `DefaultName` can also be a path or a path+filename. To specify a folder only, make the last character in `DefaultName` `'\'` or `'/'`. In this case, `uiputfile` opens the dialog box in the folder specified by the path. If you specify a path in `DefaultName` that does not exist, `uiputfile` opens the dialog box in the current folder. You can use `'.'`,`'..'`, `'\'`, `'/'`, or ~ in the `DefaultName` argument.

When typing into the dialog box, if you include either of the wildcard characters `'*'` or `'?'` in a file name, `uiputfile` does not respond to clicking **Save**. The dialog box remains open until you cancel it or remove the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

If you select or specify an existing file name, a warning dialog box opens stating that the file already exists and asks if you want to replace it.

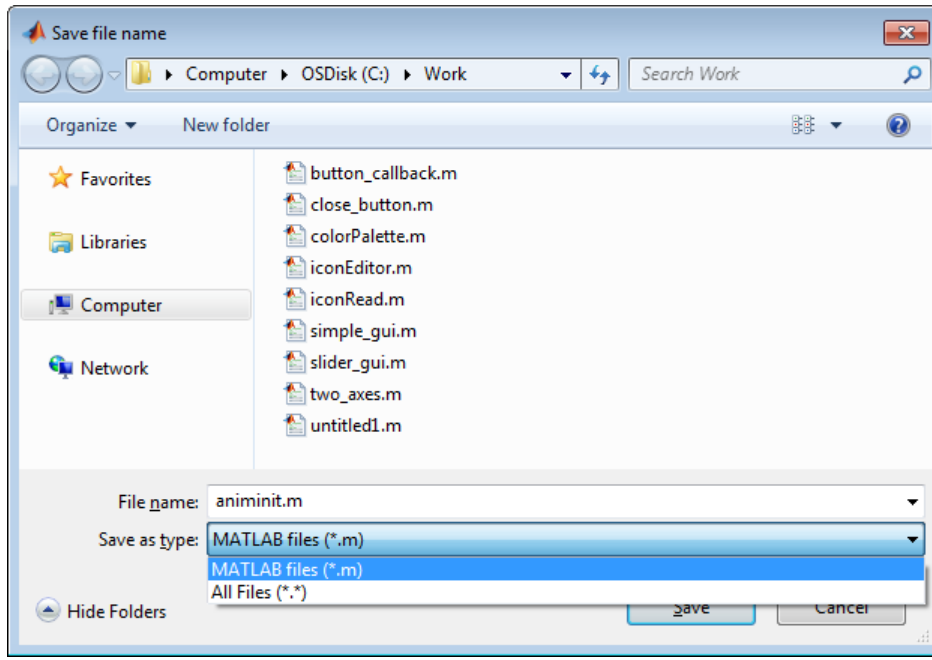
Select **Yes** to replace the existing file or **No** to return to the dialog to select another file name. Selecting **Yes** returns the name of the file. Selecting **No** returns `0`.

The visual characteristics of the dialog box depend on the operating system that is running MATLAB.

Examples

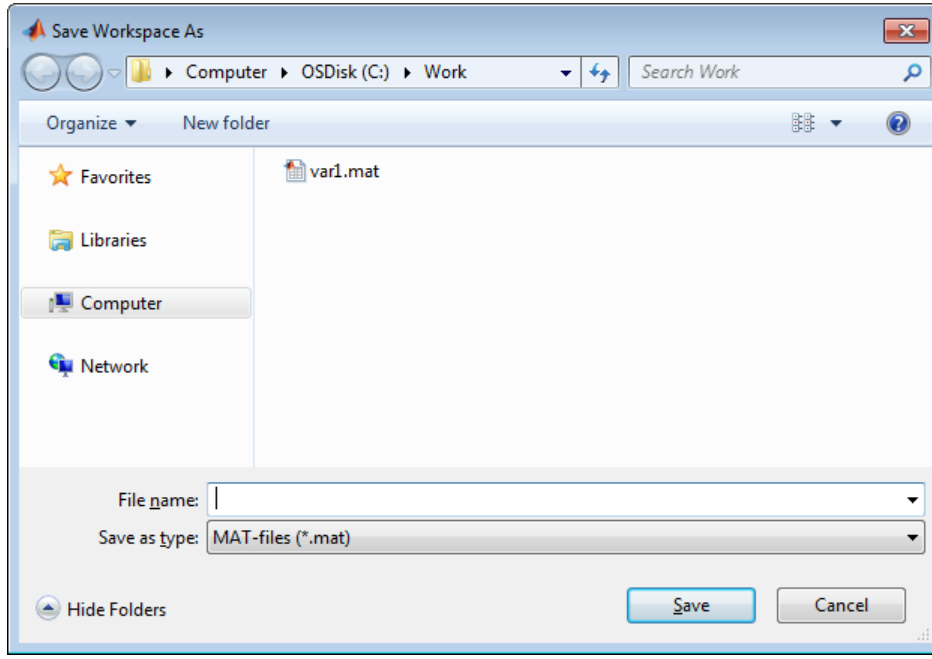
The following statement displays a dialog box entitled `'Save file name'`. This command also sets the **File name** field to `aminimit.m`, and it sets the filter to `program files (*.m)`.

```
[file,path] = uiputfile('aminimit.m','Save file name');
```



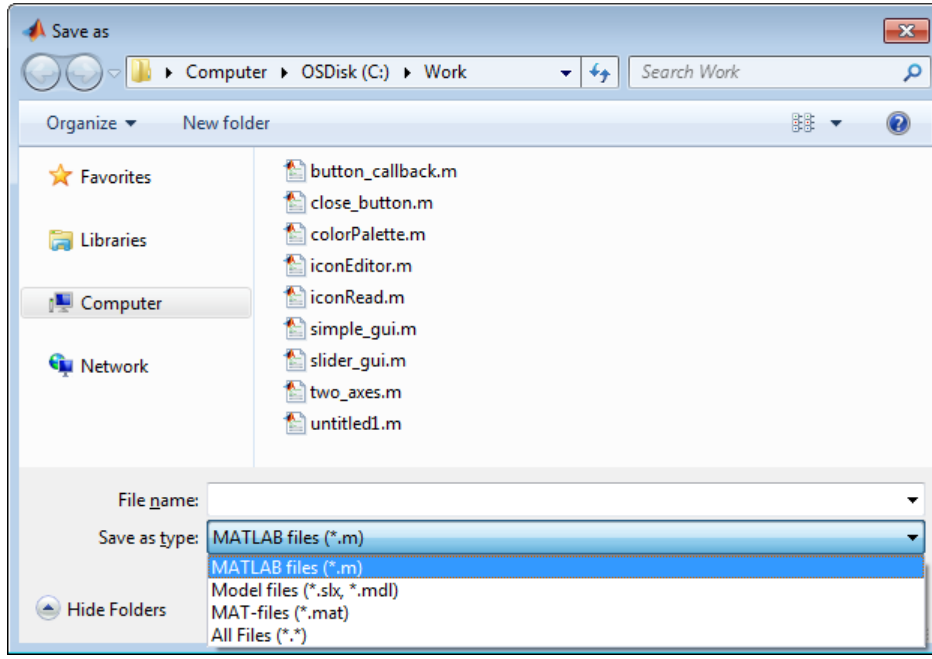
The following statement displays a dialog box entitled `'Save Workspace As'` with the filter set to MAT-files.

```
[file,path] = uiputfile('*.mat','Save Workspace As');
```



To display several file types in the **Save as type** list box, separate each file extension with a semicolon, as in the following code. `uiputfile` displays a default description for each known file type, such as "Model files" for Simulink `.mdl` and `.slx` files.

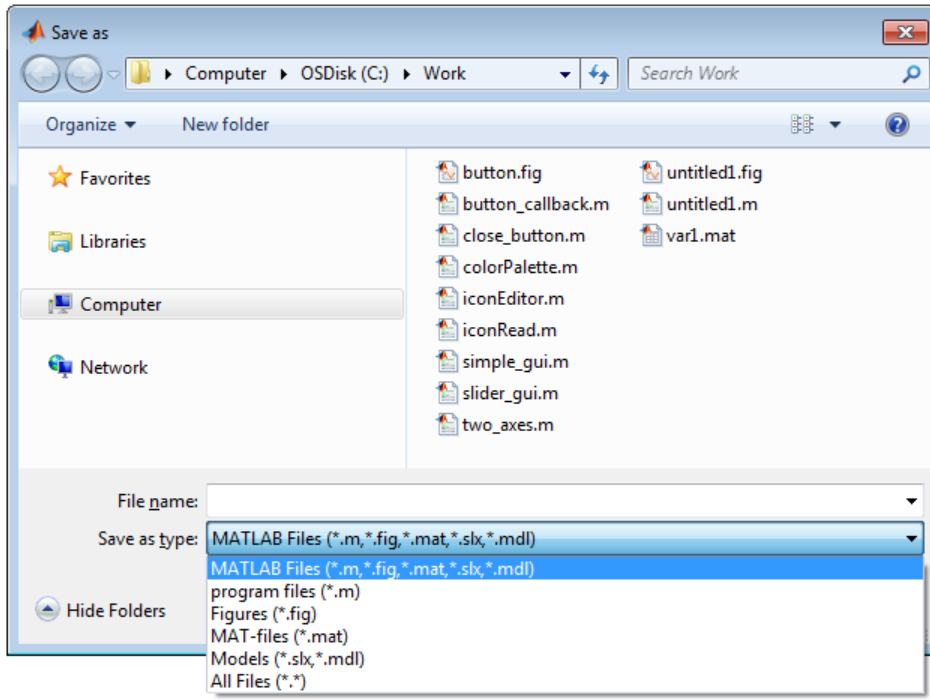
```
[filename, pathname] = uiputfile(...
 {'*.m';'*.slx';'*.mat';'*.*'},...
 'Save as');
```



To create a list of file types and give them descriptions that are different from the defaults, use a cell array. This example also associates multiple file types with the `'MATLAB Files'` and `'Models'` descriptions.

```
[filename, pathname, filterindex] = uiputfile( ...
{'*.m;*.fig;*.mat;*.slx;*.mdl',...
 'MATLAB Files (*.m,*.fig,*.mat,*.slx,*.mdl)';
 '*.m', 'program files (*.m)';...
 '*.fig','Figures (*.fig)';...
 '*.mat','MAT-files (*.mat)';...
 '*.slx;*.mdl','Models (*.slx,*.mdl)';...
 '*.*',  'All Files (*.*)'},...
 'Save as');
```

The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. The first entry of column one contains several extensions separated by semicolons. These file types all associate with the description `'MATLAB Files (*.m,*.fig,*.mat,*.slx,*.mdl)'`. The code produces the dialog box shown in the following figure.

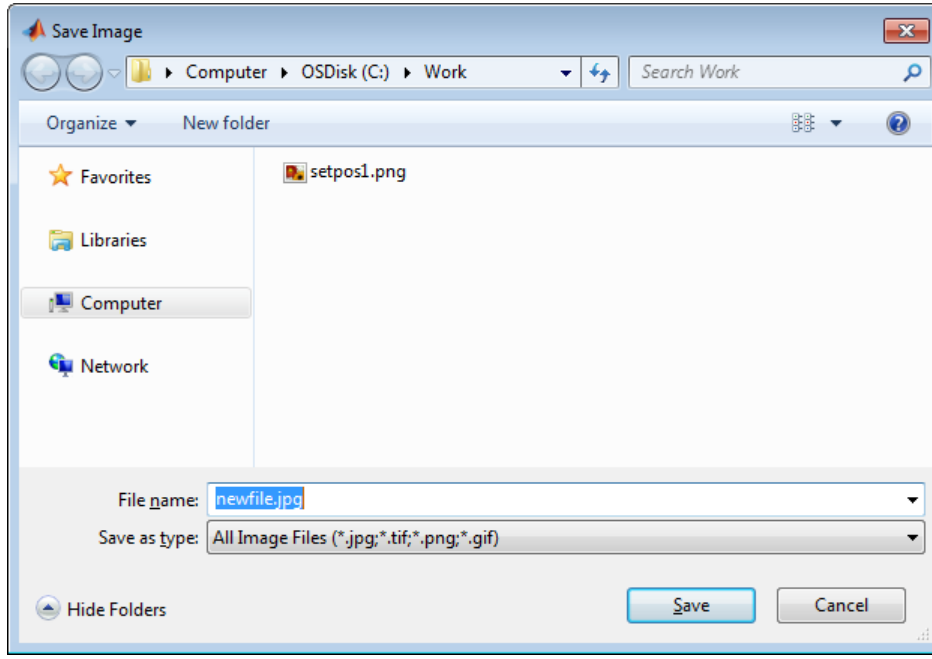


The following code checks for the existence of the file and displays a message about the result of the file selection operation.

```
[filename, pathname] = uiputfile('*.m',...
                       'Pick a MATLAB program file');
if isequal(filename,0) || isequal(pathname,0)
   disp('User selected Cancel')
else
   disp(['User selected ',fullfile(pathname,filename)])
end
```

Select or enter a file name for saving a figure as an image in one of four formats, described in a cell array.

```
uiputfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'Save Image',...
          'C:\Work\newfile.jpg')
```



See Also

`save` | `uigetdir` | `uisave` | `uigetfile`

Introduced before R2006a

uiresume

Resume execution of blocked program

Syntax

```
uiresume(h)
```

Description

`uiresume(h)` resumes the program execution that `uiwait` suspended.

Examples

This example code creates a window containing a push button. The `uiwait` function blocks MATLAB execution until the user clicks the push button.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

The `gcbf` function returns the current figure.

More About

Tips

The `uiwait` and `uiresume` functions block and resume MATLAB program execution. When creating a dialog box, you should have a uicontrol component with a callback that calls `uiresume` or a callback that destroys the dialog box. These are the only methods that resume program execution after the `uiwait` function blocks execution.

When used in conjunction with a modal dialog box, `uiresume` can resume the execution of the program that `uiwait` suspended while presenting a dialog box.

See Also

`dialog` | `figure` | `uicontrol` | `uimenu` | `uiwait` | `waitfor`

Introduced before R2006a

uisave

Open dialog box for saving variables to MAT-file

Syntax

```
uisave
uisave(variables)
uisave(variables,filename)
```

Description

`uisave`, with no arguments, prompts you for a file name, and then saves all variables from your workspace to that file.

`uisave(variables)` saves one or more workspace variables listed in variables. The variables argument is a string or a cell array of strings.

`uisave(variables,filename)` uses the specified filename as the default **File name** in the Save Workspace Variables dialog box, instead of the default `matlab.mat`.

Input Arguments

`variable`

String containing the name of a variable in the current workspace or cell array of strings when specifying more than one variable.

Default: All variables in the current workspace are saved to a MAT-file.

`filename`

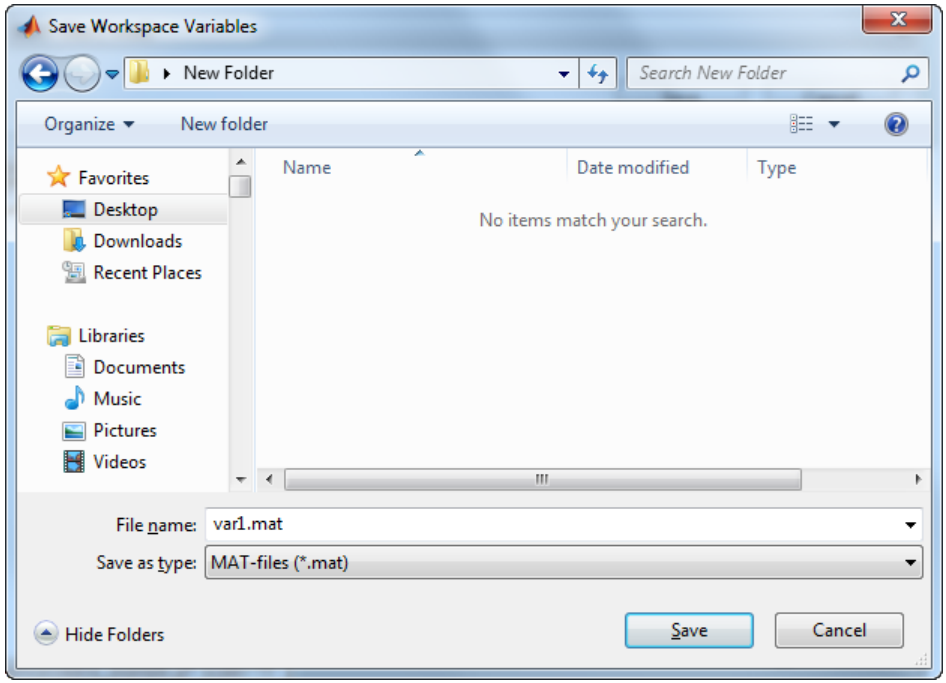
String naming a file to appear in the dialog **File name** field when the dialog opens. You can omit a file extension, or specify the file extension as `.mat`.

Default: `matlab.mat`

Examples

Create workspace variables `h` and `g`, and then display the Save Workspace Variables dialog box in the current folder with the default **File name** set to `var1`.

```
h = 365;
g = 52;
uisave({'h','g'},'var1');
```



Clicking **Save** stores the workspace variables `h` and `g` in the file `var1.mat` in the displayed folder.

More About

Modal Dialog

A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in Figure Properties.

Tips

- If you type a name in the **File name** field, such as `my_vars`, and click **Save**, the dialog saves all workspace variables to the file `my_vars.mat`. The default filename is `matlab.mat`.
- `uisave` calls `uiputfile` to choose a filename.
- If the filename you specify exists in that folder, `uisave` informs you and gives you a chance to cancel the operation.
- The `uisave` dialog box is modal.
- The `uisave` dialog is modal. A modal dialog box prevents you from interacting with other windows until you respond to it.
- "Save, Load, and Delete Workspace Variables"

See Also

`save` | `uigetfile` | `uiopen` | `uiputfile`

Introduced before R2006a

uisetcolor

Open color selection dialog box

Syntax

```
c = uisetcolor
c = uisetcolor([r g b])
c = uisetcolor(h)
c = uisetcolor(...,'dialogTitle')
```

Description

`c = uisetcolor` displays a modal color selection dialog appropriate to the platform, and returns the color selected by the user. The dialog box is initialized to `white`.

`c = uisetcolor([r g b])` displays a dialog box initialized to the specified color, and returns the color selected by the user. `r`, `g`, and `b` must be values between 0 and 1.

`c = uisetcolor(h)` displays a dialog box initialized to the color of the object specified by handle `h`, returns the color selected by the user, and applies it to the object. `h` must be the handle to an object containing a color property.

`c = uisetcolor(...,'dialogTitle')` displays a dialog box with the specified title.

If the user presses **Cancel** from the dialog box, or if any error occurs, the output value is set to the input RGB triple, if provided; otherwise, it is set to 0.

More About

Modal Dialog

A window that blocks interaction with other windows until the user closes the blocking window. For more information, see the figure `WindowStyle` property description.

See Also

`dialog` | `questdlg` | `uigetdir` | `uigetfile` | `uiputfile`

Introduced before R2006a

uisetfont

Open font selection dialog box

Syntax

```
uisetfont
uisetfont(h)
uisetfont(S)
uisetfont(...,'DialogTitle')
S = uisetfont(...)
```

Description

`uisetfont` enables you to change font properties (`FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle`) for a `text`, `axes`, or `uicontrol` object. The function returns a structure consisting of font properties and values. You can specify an alternate title for the dialog box.

`uisetfont` displays a modal dialog box and returns the selected font properties.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in Figure Properties.

`uisetfont(h)` displays a modal dialog box, initializing the font property values with the values of those properties for the object whose handle is `h`. Selected font property values are applied to the current object. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(S)` displays a modal dialog box, initializing the font property values with the values defined for the specified structure (`S`). `S` must define legal values for one or more of these properties: `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` and the field names must match the property names exactly. If other properties are defined, they are ignored. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(...,'DialogTitle')` displays a modal dialog box with the title `DialogTitle` and returns the values of the font properties selected in the dialog box.

`S = uisetfont(...)` returns the properties `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` as fields in a structure. If the user presses **Cancel** from the dialog box or if an error occurs, the output value is set to 0.

Examples

These statements create a text object, then display a dialog box (labeled Update Font) that enables you to change the font characteristics:

```
h = text(.5,.5,'Figure Annotation');
uisetfont(h,'Update Font')
```

These statements create two push buttons, then set the font properties of one based on the values set for the other:

```
% Create push button with string ABC
c1 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 10 100 20], 'String', 'ABC');
% Create push button with string XYZ
c2 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 50 100 20], 'String', 'XYZ');
% Display set font dialog box for c1, make selections,
& and save to d
d = uisetfont(c1);
% Apply those settings to c2
set(c2, d)
```

See Also

`axes` | `text` | `uicontrol`

Introduced before R2006a

uisetpref

Manage preferences used in `uigetpref`

Syntax

```
uisetpref('clearall')
```

Description

`uisetpref('clearall')` resets the value of all preferences registered through `uigetpref` to `'ask'`. This causes the dialog box to display when you call `uigetpref`.

Note Use `setpref` to set the value of a particular preference to `'ask'`.

See Also

`setpref` | `uigetpref`

Introduced before R2006a

uistack

Reorder visual stacking order of objects

Note: The `uistack` function might return different results, starting in R2014b. For more information about how the stacking behavior of UI components has changed, see "Why Are Some Components Missing or Partially Obscured?".

Syntax

```
uistack(h)
uistack(h,stackopt)
uistack(h,stackopt,step)
```

Description

`uistack(h)` shifts object `h` up one level within the visual stacking order of UI objects. You can also specify `h` to be a vector of UI objects. When `h` is a vector, the objects stack according to their order in the vector. All items in the stack must share the same parent.

`uistack(h,stackopt)` moves the object `h` to another place in the stack according to `stackopt`:

- `'up'` – moves `h` up one position in the stacking order
- `'down'` – moves `h` down one position in the stacking order
- `'top'` – moves `h` to the top of the current stack
- `'bottom'` – moves `h` to the bottom of the current stack

`uistack(h,stackopt,step)` moves object `h` up or down the number of levels specified by `step`. For example, `uistack(h,'up',2)` moves `h` up two levels in the stack.

Examples

Change Stacking Order of Two Plots in a Figure

Create a figure containing a uipanel and a child axes. Then, plot a sine function in the axes.

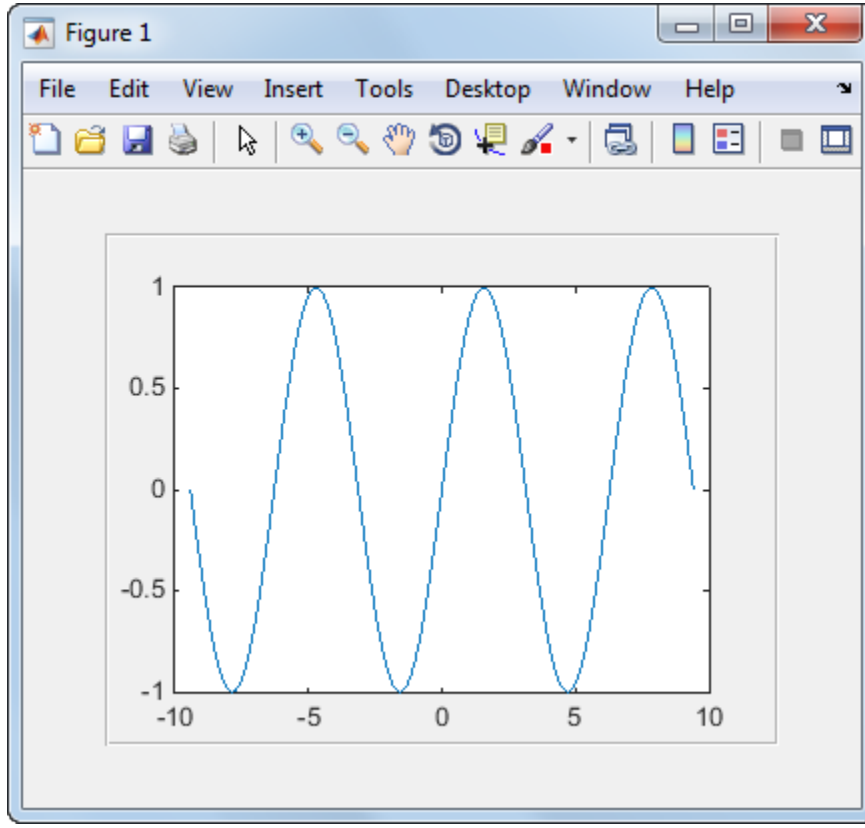
```
f = figure;
p1 = uipanel(f,'Position',[.10 .10 .80 .80],...
             'Tag','Panel with Sine Plot');
ax1 = axes('Parent',p1,'Position',[.1 .1 .8 .8]);

theta = -3*pi:pi/64:3*pi;
y1 = sin(theta);
plot(ax1,theta,y1);
```

In the same figure, create a second uipanel with a child axes. Then, create a surface plot in this axes.

```
p2 = uipanel(f,'Position',[.10 .10 .80 .80],...
             'Tag','Panel with Surface Plot');
ax2 = axes('Parent',p2,'Position',[.1 .1 .8 .8]);
Z  = peaks(25);
surf(ax2,Z);
```

Next, move the uipanel containing the sine function to the top.

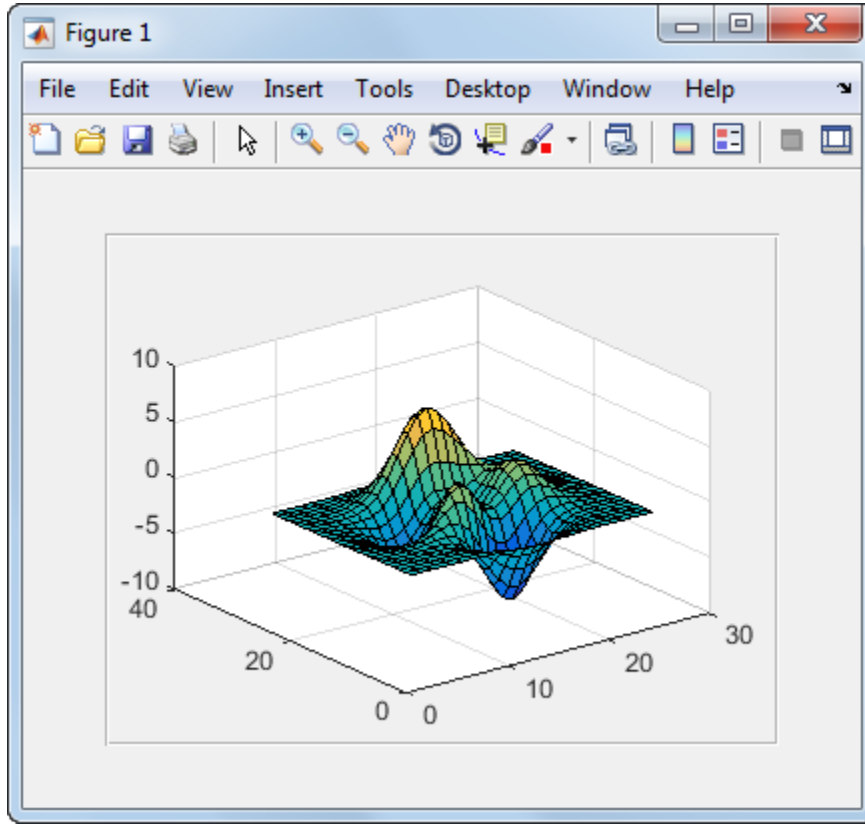
```
uistack(p1,'top');
f.Children

ans =

  2x1 Panel array:

  Panel    (Panel with Sine Plot)
  Panel    (Panel with Surface Plot)
```



Now, move the other uipanel to the top.

```
uistack(p2,'top');
f.Children
```

```
ans =

  2x1 Panel array:

  Panel    (Panel with Surface Plot)
  Panel    (Panel with Sine Plot)
```



More About

Visual Stacking Order

The order of overlapping components in a UI window. Components on top of the stack appear in front of components that are lower in the stack.

- "Manage the Stacking Order of Grouped Components"

See Also

`figure` | `uibuttongroup` | `uipanel` | `uitab`

Introduced before R2006a

uitab

Create tabbed panel

Uitabs are tabbed panels for grouping UI components together. The `uitab` function creates a uitab in a figure and sets any required properties before displaying it.

Syntax

```
t = uitab
t = uitab(Name,Value)

t = uitab(parent)
t = uitab(parent,Name,Value)
```

Description

`t = uitab` creates a uitab in an existing uitabgroup and returns the uitab object in the variable, t. If there is no uitabgroup available, then MATLAB creates a new uitabgroup in a figure to serve as the parent.

`t = uitab(Name,Value)` creates a uitab and specifies one or more uitab property names and corresponding values. Use this syntax to override the default uitab property values.

`t = uitab(parent)` creates a uitab and specifies the parent uitabgroup object for the uitab.

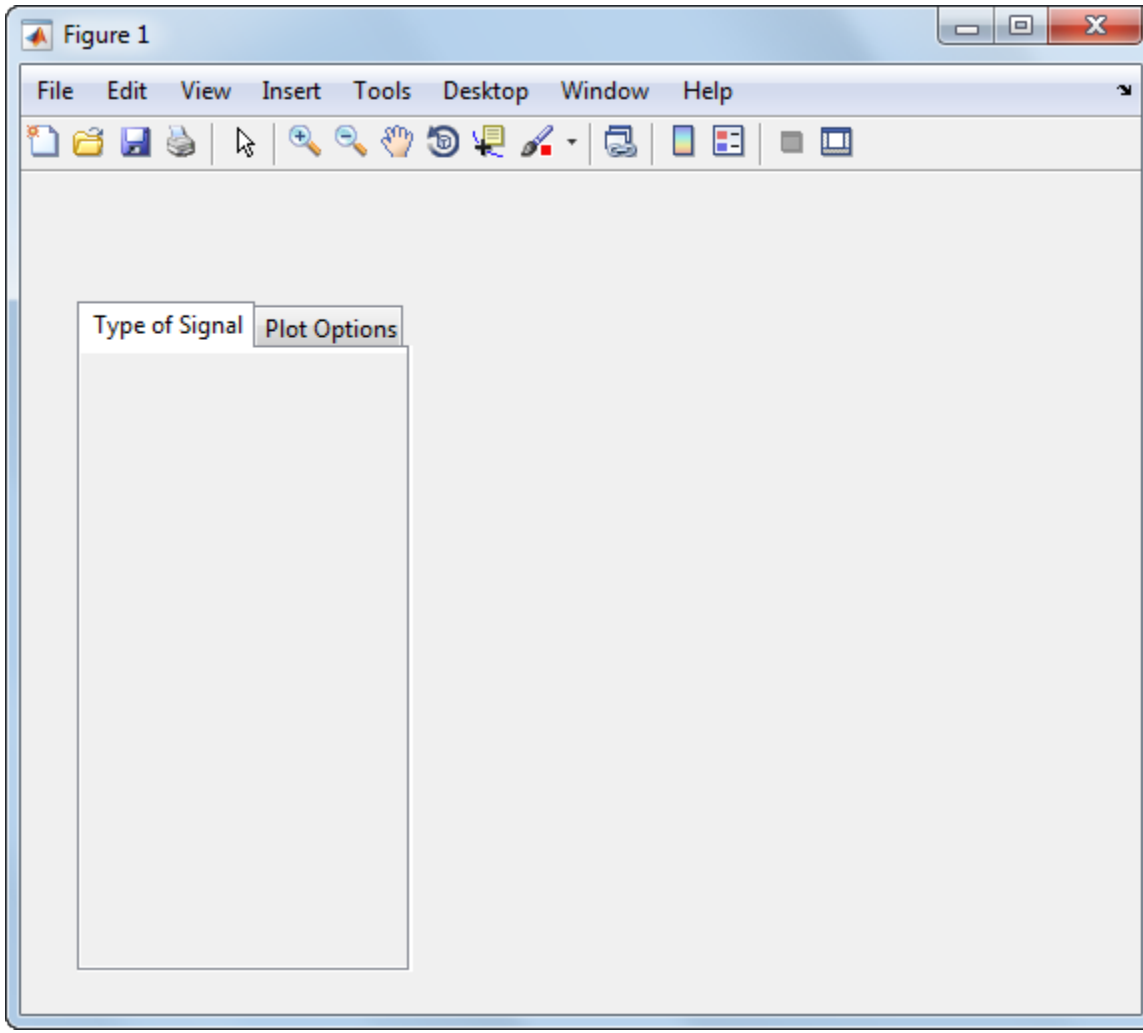
`t = uitab(parent,Name,Value)` creates a uitab with a specific parent and one or more uitab properties.

Examples

Create Two Tabs in a Figure

Create a figure containing a uitabgroup and two uitabs.

```
f = figure;
tabgp = uitabgroup(f,'Position',[.05 .05 .3 .8]);
tab1 = uitab(tabgp,'Title','Type of Signal');
tab2 = uitab(tabgp,'Title','Plot Options');
```



Get the title of the first uitab.

```
tab1.Title
```

```
ans =

Type of Signal
```

Input Arguments

`parent` — Uitab parent

uitabgroup

Uitab parent, specified as a uitabgroup object. You can move a uitab to a different uitabgroup by setting this value to the target uitabgroup object.

Name-Value Pair Arguments

The properties listed here are only a subset, for a complete list see Uitab Properties.

Example: `'Title','Plot Options'`

`'Title'` — uitab title

string

uitab title, specified as a string.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the uitab title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as Multiples of π.

Example: `'Options'`

Example: `['Multiples of ' char(960)]`

`'BackgroundColor'` — uitab background color

`[.94 .94 .94]` (default) | RGB triplet | short name | long name

uitab background color, specified as an RGB triplet, short name, or long name. The color you specify fills the area bounded by the uitab.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range

[0,1], for example, [0.4 0.6 0.7]. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Note: Some operating systems override the BackgroundColor value.

Data Types: double | char

'Position' — Location and size of uitab relative to parent

[left bottom width height]

Location and size of uitab relative to parent, returned as the vector, [left bottom width height]. The left value is the distance from the inner left edge of the parent container to the outer left edge of the uitab. The bottom value is the distance from the inner bottom edge of the parent container to the outer bottom edge of the uitab. The width value specifies the distance between the right and left outer edges of the uitab. The height value specifies the distance between the top and bottom outer edges of the uitab. All measurements are in units specified by the Units property.

'Units' — Units of measurement

'normalized' (default) | 'inches' | 'centimeters' | 'pixels' | 'points' | 'characters'

Units of measurement, specified as 'normalized', 'inches', 'centimeters', 'pixels', 'points', or 'characters'.

MATLAB uses these units to display the location and size values in the Position property:

- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- The size of a uitab specified in pixel units depends on the system display settings and resolution.
- Characters units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

See Also

Uitab Properties | Uitabgroup Properties | `uitabgroup`

Introduced in R2014b

Uitab Properties

Control appearance and behavior of tabbed panel

Uitabs are tabbed panels for grouping UI components together. The `uitab` function creates a uitab in a figure and sets any required properties before displaying it. By changing uitab property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
t = uitab;
bgcolor = t.BackgroundColor;
t.BackgroundColor = [.5 .5 .5];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`BackgroundColor` — Uitab background color

`[.94 .94 .94]` (default) | RGB triplet | short name | long name

Uitab background color, specified as an RGB triplet, short name, or long name. The color you specify fills the area bounded by the uitab.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`

Long Name	Short Name	RGB Triplet
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Note: Some operating systems override the `BackgroundColor` value.

Data Types: `double` | `char`

`ForegroundColor` — Uitab title color

`[0 0 0]` (default) | RGB triplet | short name | long name

Uitab title color, specified as an RGB triplet, short name, or long name.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: `[0 0 1]`

Example: `'b'`

Example: `'blue'`

Location and Size

`Position` — Location and size of uitab relative to parent
`[left bottom width height]`

Location and size of uitab relative to parent, returned as the vector, `[left bottom width height]`. The `left` value is the distance from the inner left edge of the parent container to the outer left edge of the uitab. The `bottom` value is the distance from the inner bottom edge of the parent container to the outer bottom edge of the uitab. The `width` value specifies the distance between the right and left outer edges of the uitab. The `height` value specifies the distance between the top and bottom outer edges of the uitab. All measurements are in units specified by the Units property.

`Units` — Units of measurement
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'pixels'` | `'points'` | `'characters'`

Units of measurement, specified as `'normalized'`, `'inches'`, `'centimeters'`, `'pixels'`, `'points'`, or `'characters'`.

MATLAB uses these units to display the location and size values in the Position property:

- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- The size of a uitab specified in pixel units depends on the system display settings and resolution.
- Characters units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

`SizeChangedFcn` — Uitab resize callback function
`''` (default) | function handle | cell array | string

Uitab resize callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Define this callback function to control the UI layout when the size of the uitab changes.

The SizeChangedFcn callback executes in these circumstances:

- The uitab becomes visible for the first time.
- The uitab is visible while its drawable area changes. The drawable area is the area inside the outer bounds of the uitab.
- The uitab becomes visible for the first time after its drawable area changes. This situation occurs when the drawable area changes while the uitab is invisible, and then it becomes visible later.

These are some of the important characteristics of the SizeChangedFcn callback and some recommended best practices:

- Consider delaying the display of the figure until after all the variables that the uitab's SizeChangedFcn uses are defined. This practice can prevent the uitab's SizeChangedFcn callback from returning an error. To delay the display of the figure, set its Visible property to `'off'`. Then, set the Visible property to `'on'` after you define the variables that your SizeChangedFcn callback uses.
- Use the `gcbo` function in your SizeChangedFcn code to get the uitab object that the user is resizing. See the description of the figure `SizeChangedFcn` for an example that uses `gcbo`.
- All visible Uitabs that are specified in normalized units resize before the parent figure does. All visible, nested uitabs resize from inside-out.

Tip As an easy alternative to specifying a SizeChangedFcn callback, you can set the Units property of all the objects you put inside the uitab to `'normalized'`. Doing so makes those components scale proportionally with the uitab.

See"Managing the Layout in Resizable UIs" for more information on controlling the resize behavior of a programmatic UIs.

Example: `@myfun`

Example: `{@myfun,x}`

Text

`Title` — Uitab title

string

Uitab title, specified as a string.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the uitab title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as Multiples of π.

Example: `'Options'`

Example: `['Multiples of ' char(960)]`

Interactive Control

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

The ButtonDownFcn callback is a function that executes when the user clicks a mouse button on the uitab.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle` | `cell` | `char`

`UIContextMenu` — Uitab context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Uitab context menu, specified as a uicontextmenu object. Use this property to display a context menu when the user right-clicks on the uitab. Create the context menu using the `uicontextmenu` function.

`TooltipString` — Tooltip text

string

Tooltip text, specified as a string. When the user hovers the mouse pointer over the uitab and leaves it there, the tooltip displays.

Example: `'Some string'`

Callback Execution Control

`Interruptible` — Callback interruption

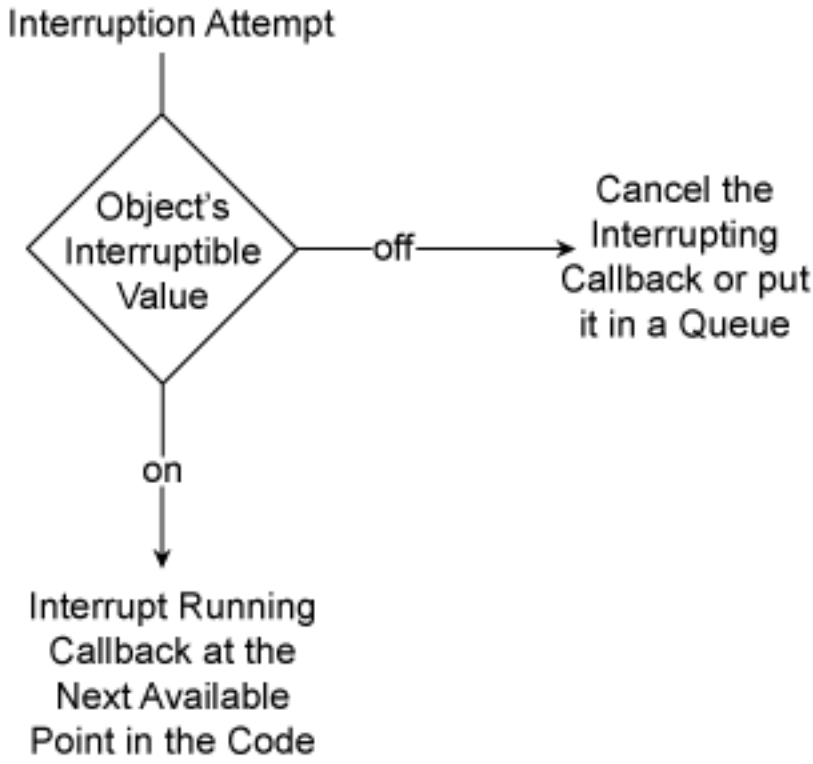
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uitab callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See "Interrupt Callback Execution" for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uitab

`'off'` (default) | `'on'`

Deletion status of uitab, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uitab begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uitab no longer exists.

Check the value of the BeingDeleted property to verify that the uitab is not about to be deleted before querying or modifying it.

`CreateFcn` — Uitab creation function

function handle | cell array | string

Uitab creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the uitab. MATLAB initializes all uitab property values before executing the CreateFcn callback.

If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uitab that is being created.

Setting the CreateFcn property on an existing uitab has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uitab object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uitab deletion function

function handle | cell array | string

Uitab deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uitab (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the uitab. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uitab that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle` | `cell` | `char`

Identifiers

`Type` — Type of graphics object
`'uitab'`

Type of graphics object, returned as `'uitab'`.

`Tag` — Uitab identifier
`''` (default) | string

Uitab identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uitab. When you need access to the uitab elsewhere in your code, you can use the `findobj` function to search for the uitab based on the Tag value.

Example: `'tab1'`

Data Types: `char`

`UserData` — Data to associate with the uitab object
empty array (default) | array

Data to associate with the uitab object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uitab parent
uitabgroup

Uitab parent, specified as a uitabgroup. You can move a uitab to a different uitabgroup by setting this property to the handle of the target uitabgroup.

`Children` — Children of uitab

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Children of uitab, returned as an empty `GraphicsPlaceholder` or a 1-D array of UI component objects. The children of uitab can be axes, uipanels, uibuttongroups, and uicontrols.

You cannot add or remove children using the `Children` property of the uitab. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the front-to-back order (stacking order) of the components on the screen.

To add a child to this list, set the `Parent` property of the child component to be the uitab object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of Uitab handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uitab handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uitab handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uitab handle is always visible.
`'callback'`	The uitab handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uitab at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The uitab handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`uitab` | `uitabgroup`

More About

- “Access Property Values”
- “Default Property Values”

uitabgroup

Create container for tabbed panels

Uitabgroups are uitab containers. They allow you to group uitabs together, identify the currently selected uitab, and detect when the user selects a different uitab.

Syntax

```
g = uitabgroup
g = uitabgroup(Name,Value)

g = uitabgroup(parent)
g = uitabgroup(parent,Name,Value)
```

Description

`g = uitabgroup` creates a uitabgroup in the current figure and returns the uitabgroup object in the variable, g. If no figure exists, then MATLAB creates a new figure and places the uitabgroup in it.

`g = uitabgroup(Name,Value)` creates a uitabgroup and specifies one or more uitabgroup property names and corresponding values. Use this syntax to override the default uitabgroup property values.

`g = uitabgroup(parent)` creates a uitabgroup and specifies the parent object for the uitabgroup (a figure, uipanel, or uibuttongroup).

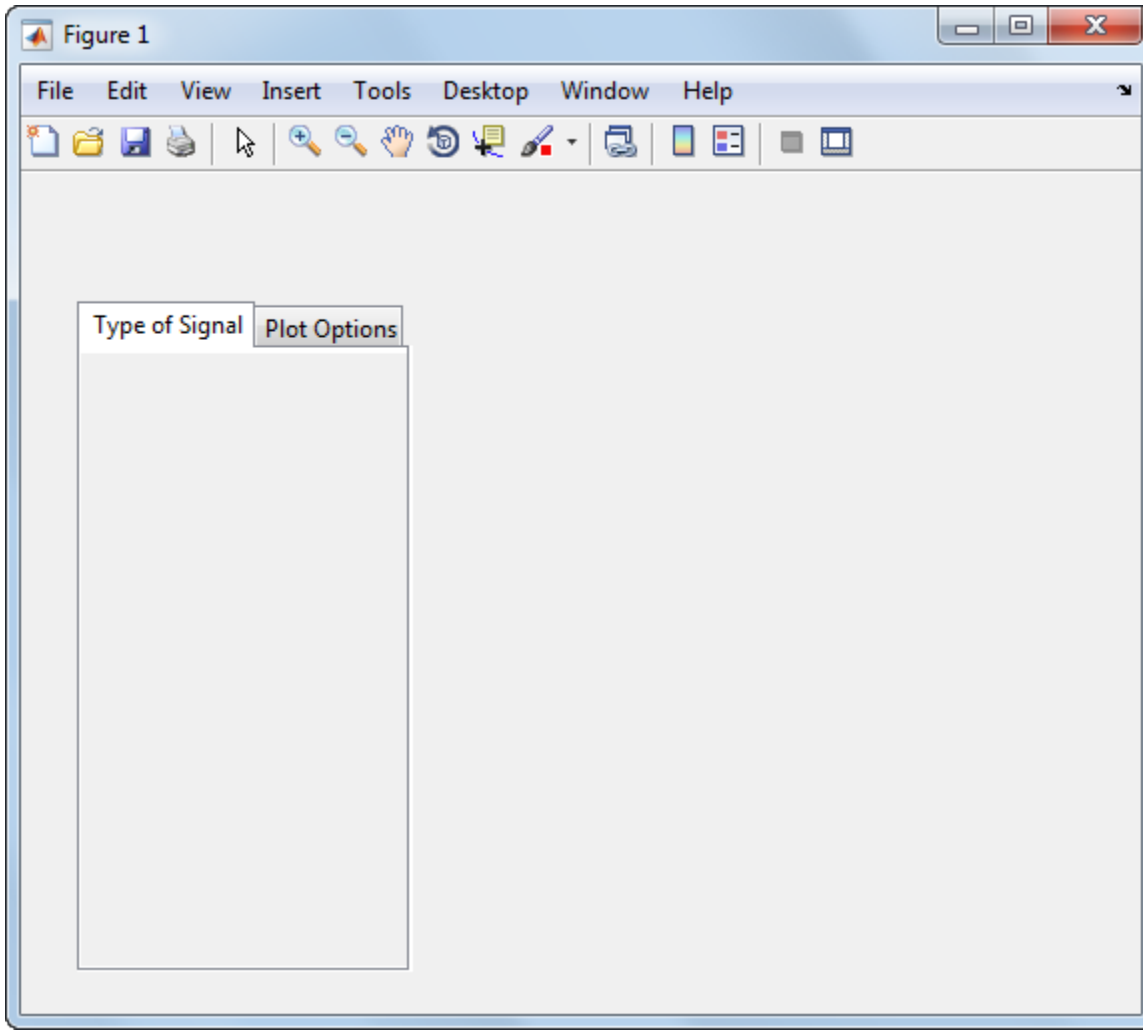
`g = uitabgroup(parent,Name,Value)` creates a uitabgroup with a specific parent and one or more uitabgroup properties.

Examples

Create Two Tabs in a Figure

Create a figure containing a uitabgroup and two uitabs.

```
f = figure;
tabgp = uitabgroup(f,'Position',[.05 .05 .3 .8]);
tab1 = uitab(tabgp,'Title','Type of Signal');
tab2 = uitab(tabgp,'Title','Plot Options');
```



Get the currently selected uitab.

```
currenttab = tabgp.SelectedTab
```

```
currenttab =

  Tab (Type of Signal) with properties:

              Title: 'Type of Signal'
    BackgroundColor: [0.9400 0.9400 0.9400]
           Position: [0.0119 0.0089 0.9702 0.9107]
              Units: 'normalized'
```

Input Arguments

`parent` — Uitabgroup parent

figure | uipanel | uitab

Uitabgroup parent, specified as a figure, uipanel, or uitab object. You can move a uitabgroup to a different figure, uipanel, or uitab by setting this value to the target parent object.

Name-Value Pair Arguments

The properties listed here are only a subset, for a complete list see Uitabgroup Properties.

Example: `'TabLocation','left'`

`'SelectedTab'` — Currently selected tab

uitab object

Currently selected tab, specified as a uitab object.

Use this property to determine the currently selected uitab within a uitabgroup. You can also use this property to set the default uitab selection.

The default value of the SelectedTab property is the first uitab that you add to the uitabgroup.

`'SelectionChangedFcn'` — Uitab selection callback

`''` (default) | function handle | cell array | string

Uitab selection callback, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different tab within the uitabgroup.

If you set the SelectionChangedFcn property to a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Callbackdata Field	Description
`OldValue`	Previously selected uitab, or `[]` if none was selected
`NewValue`	Currently selected uitab
`Source`	The parent uitabgroup object
`EventName`	`'SelectionChange'`

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

`'TabLocation'` — Tab label location

`'top'` (default) | `'bottom'` | `'left'` | `'right'`

Tab label location, specified as `'top'`, `'bottom'`, `'left'`, or `'right'`. This property specifies the location of the tab labels with respect to the uitabgroup.

`'Position'` — Location and size of uitabgroup relative to parent

`[left bottom width height]`

Location and size of the uitabgroup relative to the parent, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the uitabgroup
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the uitabgroup
`width`	Distance between the right and left outer edges of the uitabgroup
`height`	Distance between the top and bottom outer edges of the uitabgroup

All measurements are in units specified by the Units property.

Note: If the parent of the uitabgroup is a figure, then the Position values are relative to the figure's *drawable area*. The drawable area of a figure is the area inside the window borders, excluding the menu bar and tool bar.

Example: Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the Position vector. For example, this code sets the width of the uitabgroup to `.5` without changing any of the other Position values:

```
tg = uitabgroup;
tg.Position(3) = .5;
```

`'Units'` — Units of measurement
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'pixels'` | `'points'` | `'characters'`

Units of measurement, specified as `'normalized'`, `'inches'`, `'centimeters'`, `'pixels'`, `'points'`, or `'characters'`.

MATLAB uses these units to interpret the location and size values of the Position property:

- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.

- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- The size of a uitabgroup specified in pixel units depends on the system display settings and resolution.
- Characters units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the Units property before the Position property, then MATLAB sets Position using the units you specified.
- If you specify the Units property after the Position property, MATLAB sets the position using the default Units. Then, MATLAB converts the Position values to the equivalent values in the units you specified.

See Also

Uitab Properties | `uitab` | Uitabgroup Properties

Introduced in R2014b

Uitabgroup Properties

Control appearance and behavior of tabbed panel container

Uitabgroups are uitab containers that allow to you identify the currently selected uitab and detect when the user selects a different uitab. By changing uitabgroup property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
g = uitabgroup;
pos = g.Position;
g.Position = [.1 .1 .8 .8];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uitabgroup visibility

`'on'` (default) | `'off'`

Uitabgroup handle visibility, specified as `'on'` or `'off'`. The Visible property determines whether the uitabgroup displays on the screen. If the Visible property of a uitabgroup is set to `'off'`, the entire uitabgroup is invisible, but you can still specify and access its properties.

Changing the size of an invisible uitabgroup triggers the SizeChangedFcn callback if and when the uitabgroup becomes visible.

Note Changing the Visible property of a uitabgroup does *not* change the settings of the Visible property of its child components even though hiding the uitabgroup causes them to be hidden.

Location and Size

`TabLocation` — Tab label location

`'top'` (default) | `'bottom'` | `'left'` | `'right'`

Tab label location, specified as `'top'`, `'bottom'`, `'left'`, or `'right'`. This property specifies the location of the tab labels with respect to the uitabgroup.

`Position` — Location and size of uitabgroup relative to parent

`[left bottom width height]`

Location and size of the uitabgroup relative to the parent, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the uitabgroup
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the uitabgroup
`width`	Distance between the right and left outer edges of the uitabgroup
`height`	Distance between the top and bottom outer edges of the uitabgroup

All measurements are in units specified by the Units property.

Note: If the parent of the uitabgroup is a figure, then the Position values are relative to the figure's *drawable area*. The drawable area of a figure is the area inside the window borders, excluding the menu bar and tool bar.

Example: Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the Position vector. For example, this code sets the width of the Uitabgroup to `.5` without changing any of the other Position values:

```
tg = uitabgroup;
tg.Position(3) = .5;
```

`Units` — Units of measurement

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'pixels'` | `'points'` | `'characters'`

Units of measurement, specified as `'normalized'`, `'inches'`, `'centimeters'`, `'pixels'`, `'points'`, or `'characters'`.

MATLAB uses these units to interpret the location and size values of the Position property:

- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- The size of a uitabgroup specified in pixel units depends on the system display settings and resolution.
- Characters units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the Units property before the Position property, then MATLAB sets Position using the units you specified.
- If you specify the Units property after the Position property, MATLAB sets the position using the default Units. Then, MATLAB converts the Position values to the equivalent values in the units you specified.

`SizeChangedFcn` — Uitabgroup resize callback function

`''` (default) | function handle | cell array | string

Uitabgroup resize callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Define this callback function to control the UI layout when the size of the uitabgroup changes.

The SizeChangedFcn callback executes in these circumstances:

- The uitabgroup becomes visible for the first time.
- The uitabgroup is visible while its size changes.
- The uitabgroup becomes visible for the first time after its size changes. This situation occurs when the size changes while the uitabgroup is invisible, and then it becomes visible later.

These are some of the important characteristics of the SizeChangedFcn callback and some recommended best practices:

- Consider delaying the display of the figure until after all the variables that the uitabgroup SizeChangedFcn uses are defined. This practice can prevent the uitabgroup SizeChangedFcn callback from returning an error. To delay the display of the figure, set its Visible property to `'off'`. Then, set the Visible property to `'on'` after you define all the variables that your SizeChangedFcn callback uses.
- Use the `gcbo` function in your SizeChangedFcn code to get the uitabgroup object that the user is resizing. See the description of the figure `SizeChangedFcn` for an example that uses `gcbo`.
- All visible Uitabgroups that are specified in normalized units resize before the parent figure does. All visible, nested uitabgroups resize from inside-out.

Tip As an easy alternative to specifying a SizeChangedFcn callback, you can set the Units property of all the objects you put inside the uitabgroup to `'normalized'`. Doing so makes those components scale proportionally with the uitabgroup.

See “Managing the Layout in Resizable UIs” for more information.

Example: `@myfun`

Example: `{@myfun,x}`

Interactive Control

`SelectionChangedFcn` — Uitab selection callback

`''` (default) | function handle | cell array | string

Uitab selection callback, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different tab within the uitabgroup.

If you set the SelectionChangedFcn property to a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Callbackdata Field	Description
`OldValue`	Previously selected uitab, or `[]` if none was selected
`NewValue`	Currently selected uitab
`Source`	The parent uitabgroup object
`EventName`	`'SelectionChange'`

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

Example: `@myfun`

Example: `{@myfun,x}`

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

The ButtonDownFcn callback is a function that executes when the user clicks a mouse button on the uitabgroup.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`UIContextMenu` — Uitabgroup context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Uitabgroup context menu, specified as a uicontextmenu object. Use this property to display a context menu when the user right-clicks on the uitabgroup. Create the context menu using the `uicontextmenu` function.

Callback Execution Control

`Interruptible` — Callback interruption

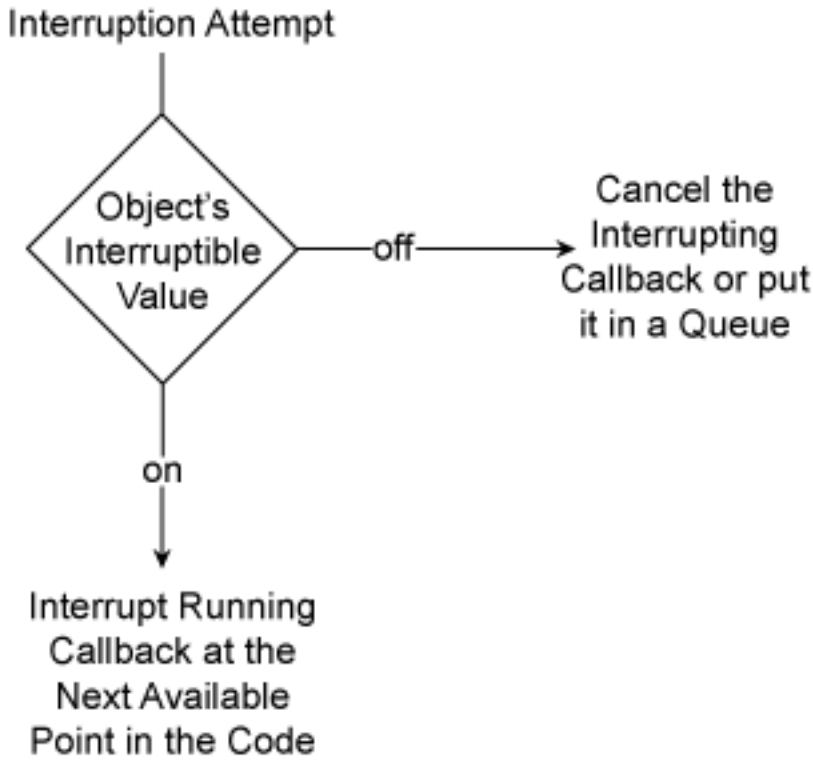
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uitabgroup callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See "Interrupt Callback Execution" for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uitabgroup

`'off'` (default) | `'on'`

Deletion status of uitabgroup, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uitabgroup begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uitabgroup no longer exists.

Check the value of the BeingDeleted property to verify that the uitabgroup is not about to be deleted before querying or modifying it.

`CreateFcn` — Uitabgroup creation function

function handle | cell array | string

Uitabgroup creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the uitabgroup. MATLAB initializes all uitabgroup property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uitabgroup that is being created.

Setting the CreateFcn property on an existing uitabgroup has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uitabgroup object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uitabgroup deletion function

function handle | cell array | string

Uitabgroup deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uitabgroup (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the uitabgroup. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uitabgroup that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`SelectedTab` — Currently selected tab

uitab object

Currently selected tab, specified as a uitab object.

Use this property to determine the currently selected uitab within a uitabgroup. You can also use this property to set the default uitab selection.

The default value of the SelectedTab property is the first uitab that you add to the uitabgroup.

`Type` — Type of graphics object

`'uitabgroup'`

Type of graphics object, returned as `'uitabgroup'`.

`Tag` — Uitabgroup identifier

`''` (default) | string

Uitabgroup identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uitabgroup. When you need access to the uitabgroup elsewhere in your code, you can use the `findobj` function to search for the uitabgroup based on the Tag value.

Example: `'tabgroup1'`

Data Types: `char`

`UserData` — Data to associate with the uitabgroup object

empty array (default) | array

Data to associate with the uitabgroup object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uitabgroup parent

figure | uipanel | uitab

Uitabgroup parent, specified as a figure, uipanel, or uitab. You can move a uitabgroup to a different figure, uipanel, or uitab by setting this property to the handle of the target figure, uipanel, or uitab.

`Children` — Children of uitabgroup

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Children of the uitabgroup, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of uitabgroup are uitabs.

You cannot add or remove children using the `Children` property of the uitabgroup. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the tabs displayed on the screen.

To add a child to this list, set the `Parent` property of the child component to be the uitabgroup object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of Uitabgroup handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uitabgroup handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uitabgroup handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`,

and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uitabgroup handle is always visible.
`'callback'`	The uitabgroup handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uitabgroup at the command-line, but allows callback functions to access it.
`'off'`	The uitabgroup handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`uitab` | `uitabgroup`

More About

- "Access Property Values"
- "Default Property Values"

uitable

Create table UI component

Syntax

```
t = uitable
t = uitable(Name,Value,...)
t = uitable(parent)
t = uitable(parent,Name,Value,...)
```

Description

`t = uitable` creates a uitable in an existing figure and returns the uitable object, `t`. If there is no figure available, then MATLAB creates a new figure to serve as the parent.

`t = uitable(Name,Value,...)` creates a uitable and specifies one or more uitable property names and corresponding values. Use this syntax to override the default uitable properties.

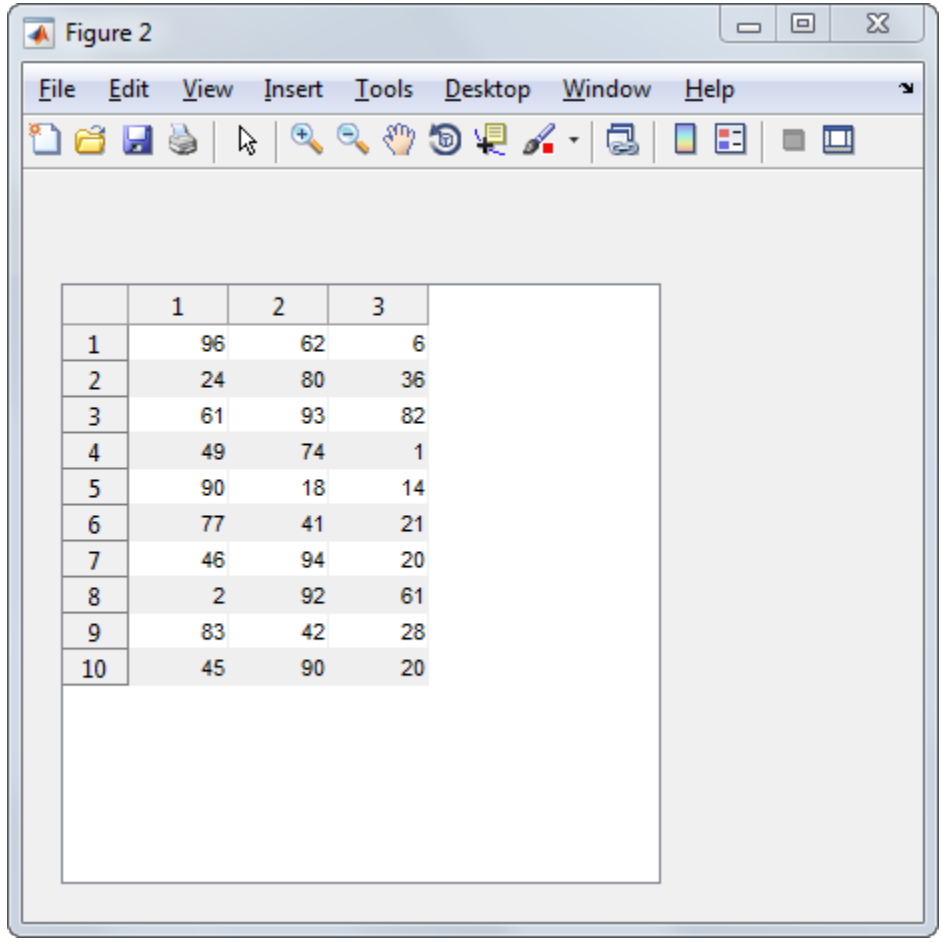
`t = uitable(parent)` creates a uitable and designates a specific parent object. The `parent` argument can be a figure, uipanel, uibuttongroup, or uitab object.

`t = uitable(parent,Name,Value,...)` creates a uitable with a specific parent and one or more uitable properties.

Examples

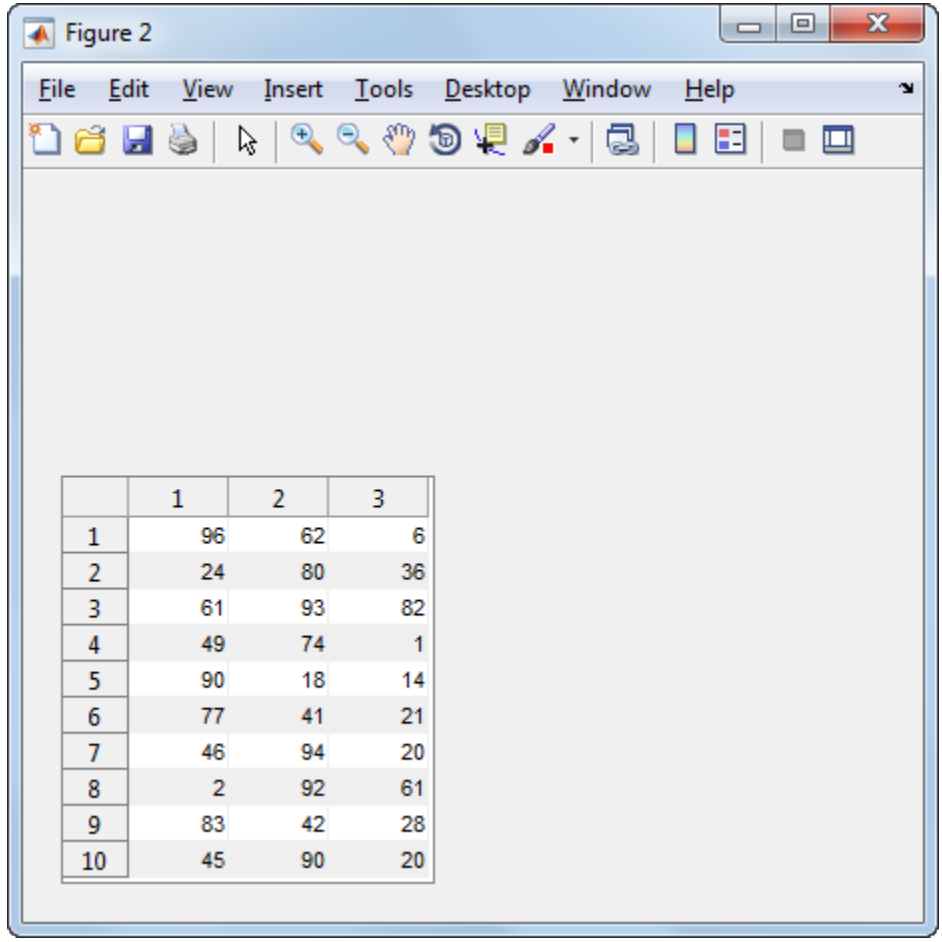
This example creates a uitable containing three columns of random numbers. The columns in this uitable are 50 pixels wide.

```
f = figure;
d = gallery('integerdata',100,[10 3],0);
t = uitable(f,'Data',d,'ColumnWidth',{50});
```



Next, set the width and height of the uitable to match the size of the enclosing rectangle:

```
t.Position(3) = t.Extent(3);
t.Position(4) = t.Extent(4);
```

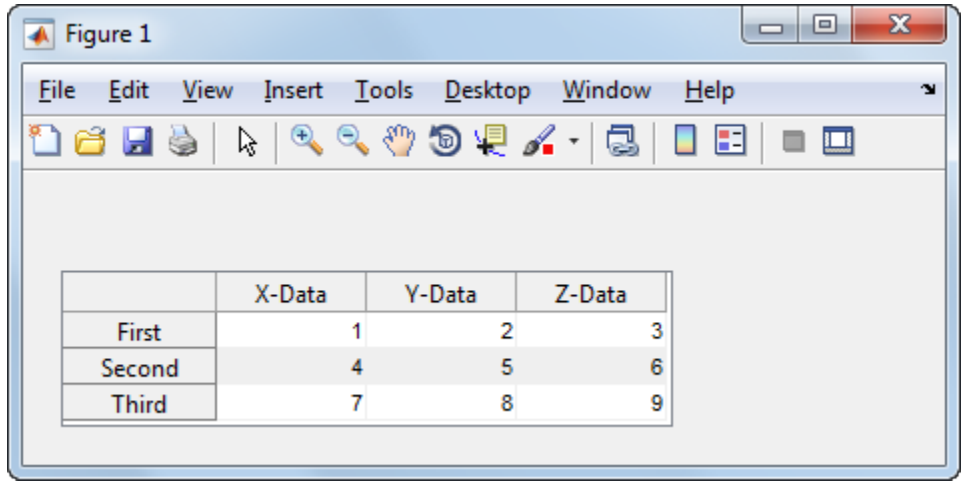


	1	2	3
1	96	62	6
2	24	80	36
3	61	93	82
4	49	74	1
5	90	18	14
6	77	41	21
7	46	94	20
8	2	92	61
9	83	42	28
10	45	90	20

This example creates a uitable with specific column names and row names:

```
f = figure('Position',[440 500 461 146]);

% create the data
d = [1 2 3; 4 5 6; 7 8 9];

% Create the column and row names in cell arrays
cnames = {'X-Data','Y-Data','Z-Data'};
rnames = {'First','Second','Third'};
```

```
% Create the uitable
t = uitable(f,'Data',d,...
            'ColumnName',cnames,...
            'RowName',rnames);

% Set width and height
t.Position(3) = t.Extent(3);
t.Position(4) = t.Extent(4);
```

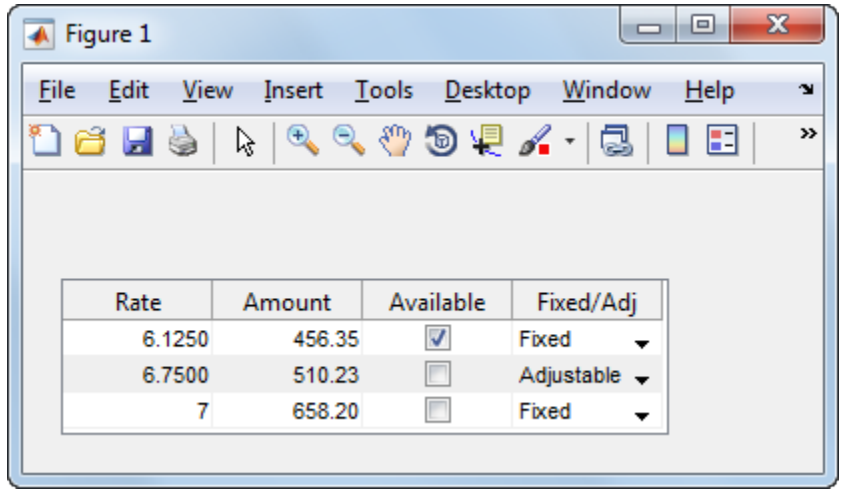


This example creates a table containing different types of data:

- First two columns are numeric. The first column displays values with up to four decimal places. The second column displays values in currency format.
- Third column contains check boxes (specified as `logical` values in the Data array).
- Fourth column contains pop-up menus, each having two choices: `Fixed` and `Adjustable`.

```
f = figure('Position',[100 100 400 150]);

% Column names and column format
columnname = {'Rate','Amount','Available','Fixed/Adj'};
columnformat = {'numeric','bank','logical',{'Fixed' 'Adjustable'}};

% Define the data
d =    {6.125678  456.3457  true   'Fixed';...
        6.75   510.2342  false  'Adjustable';...
```

```
        7      658.2     false   'Fixed';};

% Create the uitable
t = uitable('Data', d,...
            'ColumnName', columnname,...
            'ColumnFormat', columnformat,...
            'ColumnEditable', [false false true true],...
            'RowName',[]);

% Set width and height
t.Position(3) = t.Extent(3);
t.Position(4) = t.Extent(4);
```



More About

Tips

If the `ColumnEditable` property is set to `true` for specific columns, the user can edit values in that column. By default, this property is set to `false` for all columns. If a noneditable column contains pop-up choices, then only the current choice is visible in the table.

The CellEditCallback function executes after the user edits a value and perform any of these actions:

- Press the **Enter** key.
- Click anywhere else within the table.
- Click anywhere else within the same figure window, or in another figure window.

The CellSelectionCallback function executes when the user selects one or more table cells or modifies the current selection.

You cannot select table cells programmatically. Directly clicking cells is the only method of selection.

- “Access Property Values”

See Also

`figure` | `format` | Uitable Properties

Introduced in R2008a

Uitable Properties

Control appearance and behavior of table

Uitables are tables that list data in a figure. The `uitable` function creates a table and sets any required properties before displaying it. By changing uitable property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
t = uitable;
fgcolor = t.ForegroundColor;
t.ForegroundColor = [0 0 1];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uitable visibility

`'on'` (default) | `'off'`

Uitable visibility, specified as `'on'` or `'off'`. When Visible is `'off'`, the uitable is not visible, but you can query and set its properties.

To make your program start faster, set the Visible property of all uitables that are not initially displayed to `'off'`.

`BackgroundColor` — Uitable cell background color

RGB triplet | 2-by-3 matrix of RGB triplets

Uitable cell background color, specified as an RGB triplet or a 2-by-3 matrix of RGB triplets. An RGB triplet is a row vector that specifies the intensities of the red, green, and blue components of the color. The intensities must be in the range, `[0,1]`. If you specify a 2-by-3 matrix, then each row of the matrix must be an RGB triplet. Color names are not valid.

If you specify the background color as a 2-by-3 matrix, then MATLAB uses the second row of the matrix when the RowStriping property is `'on'`. The table background is not striped unless both RowStriping is `'on'` and the background color is a 2-by-3 matrix.

The following table lists the RGB triplets for commonly used colors.

Color	RGB Triplet
Yellow	`[1 1 0]`
Magenta	`[1 0 1]`
Cyan	`[0 1 1]`
Red	`[1 0 0]`
Green	`[0, 1 0]`
Blue	`[0 0 1]`
White	`[1 1 1]`
Black	`[0 0 0]`

Example: `[1 1 1]`

Example: `[1 1 1; .9 .9 .9]`

`ForegroundColor` — Cell text color

`[0 0 0]` (default) | RGB triplet | short name | long name

Cell text color, specified as an RGB triplet, short name, or long name. The color you specify sets the text color for all cells.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`

Long Name	Short Name	RGB Triplet
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black	'k'	[0 0 0]

Example: [0 0 1]

Example: 'b'

Example: 'blue'

Location and Size

Position — Location and size of uitable

[left bottom width height]

Location and size of the uitable relative to the parent, specified as the vector, [left bottom width height]. This table describes each element in the vector.

Element	Description
left	Distance from the inner left edge of the parent container to the outer left edge of the uitable
bottom	Distance from the inner bottom edge of the parent container to the outer bottom edge of the uitable
width	Distance between the right and left outer edges of the uitable
height	Distance between the top and bottom outer edges of the uitable

All measurements are in units specified by the Units property.

Note: If the parent of the uitable is a figure, then the Position values are relative to the figure's *drawable area*. The drawable area of a figure is the area inside the window borders, excluding the menu bar and tool bar.

Use the Extent property to determine proper sizing for a uitable with respect to the table data. Set the `width` and `height` of Position property to the width and height of the Extent property. Be aware that if the table has large extents, doing this can cause the table to extend beyond the right or top edge of its parent container.

Example: `[0 0 300 300]`

Example: Set Width and Height to Accommodate Data Size

You can combine dot notation and array indexing when you want to change one or two consecutive values in the Position vector. For example, this code sets the width and height of the uitable to match the Extent of the table:

```
t = uitable('Data',rand(10,3));
t.Position(3:4) = t.Extent(3:4);
```

`Units` — Units of measurement

`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as `'pixels'`, `'normalized'`, `'inches'`, `'centimeters'`, `'points'`, or `'characters'`.

MATLAB uses these units to interpret the location and size values of the Position property:

- The size of a uitable specified in pixel units depends on the system display settings and resolution.
- Normalized units map the lower left corner of the parent container to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- Inches, centimeters, and points are absolute units. 1 point = 1/72 inch.
- Character units are determined by the default system font. Each character unit is the width of the letter x. The height of each character unit is the distance between the baselines of two lines of text.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the Units property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the Units property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the Units before the Position property, then MATLAB sets Position using the units you specify.
- If you specify the Units property after the Position property, MATLAB sets the position using the default Units. Then, MATLAB converts the Position value to the equivalent value in units you specify.

`Extent` — Size of uitable rectangle

four-element row vector

Size of the uitable rectangle, returned as a four-element row vector. The first two elements of the vector are always zero. The third and fourth elements are the width and height of the rectangle containing the uitable, respectively. All measurements are in units specified by the Units property.

MATLAB determines the size of the rectangle based on the current Data, RowName and ColumnName property values. MATLAB estimates the width and height values using the column and row widths. The estimated extent can be larger than the figure.

When the uitable Units property is set to `'normalized'`, the Extent values are measured relative to the figure, regardless of whether the uitable is contained in (parented to) a uipanel.

Consider using the Extent property value when specifying the width and height values of the uitable Position property.

Font Style

`FontName` — Font for displaying cell content

string

Font for displaying the cell content, specified as a string. To display and print properly, this must be a font that your system supports. To display and print properly, the font name must refer to a font that the user's system supports. The default font is system dependent.

To use a fixed-width font, set the FontName property to the string, `'FixedWidth'`. This setting instructs MATLAB to use the value of the root FixedWidthFontName property, which you can set in the `startup.m` file.

For more information about the `startup.m` file, see "Startup Options in MATLAB Startup File"

Example: `'Arial'`

`FontSize` — Font size for uitable cell content

positive number

Font size for the uitable cell content, specified as a positive number. MATLAB uses the units specified by FontUnits. The default size is system-dependent. If you set FontUnits to `'normalized'`, then the FontSize value is a number between 0 and 1.

Example: `12`

Example: `12.5`

Data Types: `double`

`FontUnits` — Units of font size for uitable cell contents

`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Units of font size for the uitable cell contents, specified as `'points'`, `'normalized'`, `'inches'`, `'centimeters'`, or `'pixels'`.

If you set this property to `'normalized'`, then MATLAB interprets the font size as a fraction of the uitable height. When you resize the uitable, MATLAB scales the displayed font to maintain that fraction.

The other FontUnits options (`pixels`, `inches`, `centimeters`, and `points`) are absolute units. 1 point = 1/72 inch.

`FontWeight` — Font weight for uitable cell contents

`'normal'` (default) | `'bold'`

Font weight for the uitable cell contents, specified as a value from the following table.

FontWeight Value	Description
`'normal'`	Normal font weight
`'bold'`	Heavy font weight

Not all fonts support all font weights. Therefore, if you specify an unsupported value for the `FontWeight` property, the result might appear the same as the default.

Note: The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

`FontAngle` — Character slant of uitable cell contents

`'normal'` (default) | `'italic'`

Character slant of the uitable cell contents, specified as `'normal'` or `'italic'`. MATLAB uses this property to select a font from those available on your system. Setting this property to `'italic'` selects a slanted version of the font, if it is available on your system.

Note: The `'oblique'` value has been removed. Use `'italic'` instead.

Interactive Control

`CellEditCallback` — Cell edit callback function

function handle | cell array | string

Cell edit callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This function makes the uitable respond when the user changes the contents of a table cell. You can use this callback function to perform calculations or validate user input.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description
`Indices`	1-by-2 array containing the row and column indices of the cell the user edited.
`PreviousData`	Previous cell data. The default is an empty matrix, `[]`.
`EditData`	User-entered string.
`NewData`	Value that MATLAB wrote to the Data property array. It is either the same as EditData or a converted value. The `NewData` property is empty if MATLAB detects an error in the user-entered data.
`Error`	Error message returned if MATLAB detects an error in the user-entered data. The `Error` property is empty when MATLAB successfully writes the value to the Data property. If the `Error` property is not empty, then the CellEditCallback can display the string, or it can attempt to fix the problem.

When the user edits a cell, MATLAB attempts to store the user-entered value in the Data property, converting the value if necessary. Next, MATLAB calls the CellEditCallback function and passes the callback data. If there is no CellEditCallback function, and the user-entered data results in an error, then the contents of the cell reverts to its previous value and no error displays.

Example: Evaluate User Input

This example shows how to create a callback function that evaluates a user-entered data in a numeric table cell. Paste this code into an editor and run it to see how it works.

```
function myui
    f = figure;
    myData = { 'A '  31; 'B'  41; 'C'  5; 'D' 2.6};
    t = uitable('Parent',f,...
                'Position', [25 25 700 200], ...
                'Data',myData,...
                'ColumnEditable', [false true], ...
                'CellEditCallback',@converttonum);
        function converttonum(hObject,callbackdata)
             numval = eval(callbackdata.EditData);
```

```
            r = callbackdata.Indices(1)
            c = callbackdata.Indices(2)
            hObject.Data{r,c} = numval;
        end
end
```

When you run `myui`, you can change a value in the second column of the table. In response, the `converttonum` callback function executes. The `converttonum` function uses the `eval` function to evaluate your input. Then, it sets the cell data to the value of `numval`. For example, if you enter `pi` or`1+1`, the `converttonum` function sets the table cell value to a numeric representation of the input. Because there is no error checking in the `converttonum` function, invalid expressions return an error and the new value of the table cell becomes `NaN`.

`CellSelectionCallback` — Cell selection callback function

function handle | cell array | string

Cell selection callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback function executes when the user performs one of the following actions:

- Highlights a data cell (not a row or column header cell) by clicking it or navigating to it with an arrow key
- Adds a new cell to their selection by **Shift**-clicking it
- Deselects a cell by **Ctrl**-clicking it

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains one property, `Indices`. The `Indices` property is an n-by-2 array containing the row index and column index of the selected cells.

Example: `@myfun`

Example: `{@myfun,x}`

`ColumnEditable` — Ability to edit column cells

empty matrix ([ ]) (default) | logical 1-by-n array | scalar logical value

Ability to edit column cells, specified as:

- An empty matrix ([ ]) — No columns are editable.
- A logical 1-by-n array — Specify which columns are editable. The value of n is equal to the number of columns in the table. Each value in the array corresponds to a table column. A value of `true` in the array makes the cells in that column editable. A value of `false` makes the cells in that column uneditable.
- A scalar logical value — Make the entire table editable or uneditable.

Table columns that contain check boxes or pop-up menus must be editable so the user can interact with these controls.

Example: `[ ]`

Example: `[false true true]`

Example: `false`

`RearrangeableColumns` — Ability to rearrange table columns

`'off'` (default) | `'on'`

Ability to rearrange table columns, specified as `'off'` or `'on'`. This property provides a way to let users reorder the table columns (but not the labels) by clicking and dragging the column headers.

Note: Rearranging table columns in the UI does not affect the columns in the Data property array.

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | string

Button-press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

The ButtonDownFcn callback is a function that executes when the user clicks a mouse button on the uitable. The callback executes in the following situations:

- The user right-clicks the uitable, and the uitable Enable property is set to `'on'`.
- The end user right-clicks or left-clicks the Uitable, and the uitable Enable property is set to `'off'` or `'inactive'`.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`KeyPressFcn` — Key press callback function

`''` (default) | function handle | cell array | string

Key press callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see “How to Specify Callback Property Values”.

This callback function executes when the uitable object has focus and the user presses a key. If you do not define a function for this property, MATLAB passes key presses to the parent figure. Repeated key presses retain the focus of the uitable, and the function executes with each key press. If the user presses multiple keys at approximately the same time, MATLAB detects the key press for the last key pressed.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Character`	The character that displays as a result of pressing a key or keys. The character can be empty or unprintable.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	A cell array containing the names of one or more modifier keys that are being pressed (such as, **Ctrl**, **Alt**, **Shift**).	`{1x0 cell}`	`{1x0 cell}`	`{'shift'}`	`{'shift'}`
`Key`	The key being pressed, identified by the (lowercase) label on the key, or a descriptive string.	`'a'`	`'equal'`	`'shift'`	`'a'`
`Source`	The object that has focus when the user presses the key.	uitable object	uitable object	uitable object	uitable object
`EventName`	The action that caused the callback function to execute.	`'KeyPres`	`'KeyPre`	`'KeyPress`	`'KeyPress'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the CurrentCharacter property of the figure to determine which character the user pressed.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`KeyReleaseFcn` — Key-release callback function

`''` (default) | function handle | cell array | string

Key-release callback function, specified as one of these values:

- Function handle

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This callback function executes when the uitable object has focus and the user releases a key.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Character`	Character interpretation of the key that was released.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	Current modifier, such as `'control'`. This value is always an empty cell array for aseFcn callbacks.	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`
`Key`	Name of the key that was released, identified by the lowercase label on the key, or a descriptive string.	`'a'`	`'equal'`	`'shift'`	`'a'`
`Source`	The object that has focus when the user presses the key.	uitable object	uitable object	uitable object	uitable object
`EventName`	The action that caused the callback function to execute.	`'ase'`	`'ase'`	`'ase'`	`'ase'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the CurrentCharacter property of the figure to determine which character the user pressed.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`Enable` — Operational state of uitable

`'on'` (default) | `'inactive'` | `'off'`

Operational state of uitable, specified as `'on'`, `'off'`, or `'inactive'`. The Enable property controls whether the uitable responds to button clicks. The are three possible values:

- `'on'` – The uitable is operational.
- `'off'` – The uitable is not operational and appears grayed-out.
- `'inactive'` – The uitable is not operational, but it has the same appearance as when Enable is `'on'`.

The value of the Enable property and the type of button click determine how the UI responds.

Enable Value	Response to Left-Click	Response to Right-Click
`'on'`	The uitable's CellSelectionCallback function executes (only for table cells, not header cells). The `Indices` property updates in the callback data object that MATLAB passes to the callback function.	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The uitable `ButtonDownFcn` callback executes.
`'off'` or `'inactive'`	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The uitable's `ButtonDownFcn` callback executes.	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The uitable's `ButtonDownFcn` callback executes.

`TooltipString` — Tooltip text

string

Tooltip text, specified as a string. When the user hovers the mouse pointer over the uitable and leaves it there, the tooltip displays. If you want to create a tooltip that has

more than one line of text, use `sprintf` to generate a string containing newline (`\n`) characters and then set the TooltipString to that string.

Example: `'Some string'`

Example: Specify TooltipString Containing Two Lines

```
t = uitable;
s = sprintf('UITable tooltip line 1\nUITable tooltip line 2');
t.TooltipString = s;
```

Data Types: `char`

`UIContextMenu` — Uitable context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Uitable context menu, specified as a uicontextmenu object. Use this property to display a context menu when the user right-clicks on the uitable. Create the context menu using the `uicontextmenu` function.

`Selected` — Selection status of uitable

`'off'` (default) | `'on'`

Note: The behavior of the Selected property changed in R2014b, and it is not recommended. It no longer has any effect on uitables. This property might be removed in a future release.

`SelectionHighlight` — Ability to highlight selection handles

`'on'` (default) | `'off'`

Note: The behavior of the SelectionHighlight property changed in R2014b, and it is not recommended. It no longer has any effect on uitables. This property might be removed in a future release.

Callback Execution Control

`Interruptible` — Callback interruption

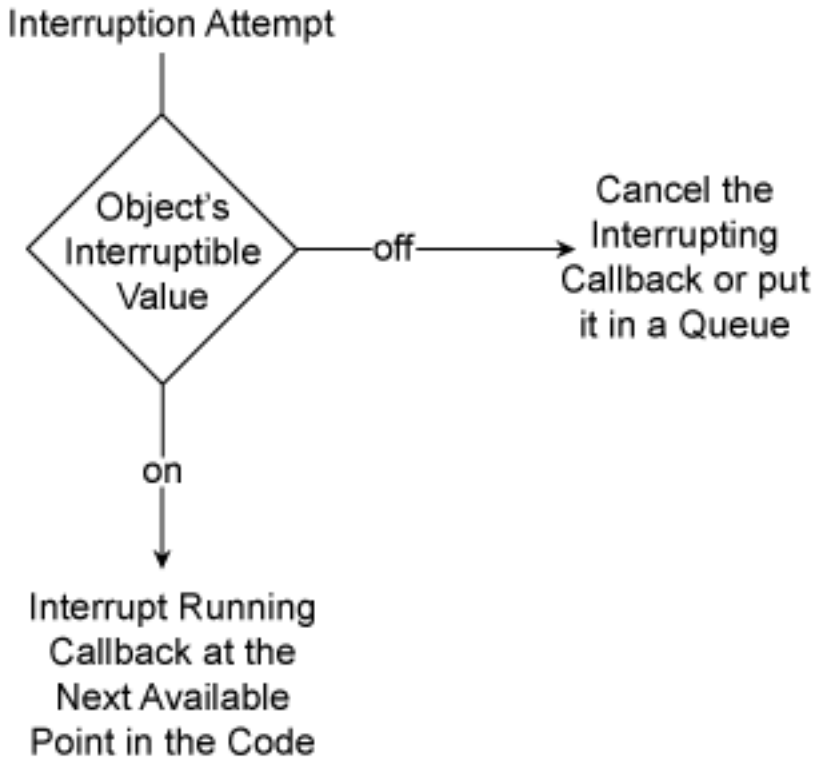
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uitable callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

Ability to become current object, specified as `'on'` or `'off'`:

- Setting the value to `'on'` allows the uitable to become the current object when the user clicks on it. A value of `'on'` also allows the figure CurrentObject property and the `gco` function to report the uitable as the current object.
- Setting the value to `'off'` sets the figure CurrentObject property to an empty `GraphicsPlaceholder` array when the user clicks on the uitable.

Note: Use the Enable property to enable or disable a uitable.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uitable

`'off'` (default) | `'on'`

Deletion status of uitable, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uitable begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uitable no longer exists.

Check the value of the BeingDeleted property to verify that the uitable is not about to be deleted before querying or modifying it.

`CreateFcn` — Uitable creation function

function handle | cell array | string

Uitable creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uitable. MATLAB initializes all uitable property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uitable that is being created.

Setting the CreateFcn property on an existing uitable has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uitable object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uitable deletion function

function handle | cell array | string

Uitable deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uitable (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the uitable. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uitable that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object

`'uitable'`

Type of graphics object, returned as `'uitable'`.

`Tag` — Uitable identifier

`''` (default) | string

Uitable identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uitable. When you need access to the uitable elsewhere in your code, you can use the `findobj` function to search for the uitable based on the Tag value.

Example: `'table1'`

Data Types: `char`

`UserData` — Data to associate with the uitable object

empty array (default) | array

Data to associate with the uitable object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uitable parent

figure | uipanel | uibuttongroup | uitab

Uitable parent, specified as a figure, uipanel, uibuttongroup, or uitab. You can move a uitable to a different figure, uipanel, uibuttongroup, or uitab by setting this property to the handle of the target figure, uipanel, uibuttongroup, or uitab.

`Children` — Children of uitable

empty array

Children of uitable, returned as an empty array. Uitable objects have no children. Setting this property has no effect.

`HandleVisibility` — Visibility of Uitable handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uitable handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uitable handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions

that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uitable handle is always visible.
`'callback'`	The uitable handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uitable at the command-line, but allows callback functions to access it.
`'off'`	The uitable handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

Table Data

`Data` — Table content

numeric array | cell array

Table content, specified as a numeric array or cell array. The table data can be any numeric type, `logical`, or `char`. Use a cell array to specify a mixture of different data types.

The ColumnFormat property specifies the format for displaying the table contents. Therefore, if there is a mismatch between the data type of the Data property array and the ColumnFormat property, MATLAB converts the data value or displays a warning. See the ColumnFormat property description for more information.

To prevent warnings that might occur when users enter invalid data, write a CellEditCallback function to convert the data to the appropriate type.

If the number of rows in the Data property array does not match the number of elements in the RowName array, then the number of rows in the resulting table is the larger of the two values. The same is true when the ColumnName property does not match the number of columns in the Data property array.

Example: `rand(10,3)`

Example: `{'blue' 5 true; 'orange' 25 false}`

Table Layout

`RowName` — Row heading names

`'numbered'` (default) | n-by-1 cell array of strings | empty cell array (`{}`) | empty matrix (`[]`)

Row heading names, specified as one of these values:

- `'numbered'` — The row headings are sequential numbers that start at `1`.
- Cell array — Each element of the cell array becomes the name of a row. Row names are restricted to one line of text. If you specify a 1-by-n cell array, MATLAB stores and returns the value as a n-by-1 cell array.
- Empty cell array (`{}`) — The resulting table has no row headings.
- Empty matrix (`[]`) — The resulting table has no row headings

If the number of rows in the Data property array does not match the number of elements in the RowName array, then the number of rows in the resulting table is the larger of the two values.

Example: `{'Name';'Telephone Number'}`

Example: `{'Name';[]}`

`ColumnName` — Column heading names

`'numbered'` (default) | n-by-1 cell array of strings | empty cell array (`{}`) | empty matrix (`[]`)

Column heading names, specified as one of these values:

- `'numbered'` — The column headings are sequential numbers that start at `1`.
- Cell array — Each element of the cell array becomes the name of a column. If you specify a 1-by-n cell array, MATLAB stores and returns the value as a n-by-1 cell array. Specify a multiline column name as a string vector separated by vertical slash (`|`) characters. For example, the value, `'Telephone|Number'`, produces a column heading with a newline character between the words, "Telephone" and "Number".
- Empty cell array (`{}`) — The resulting table has no column headings.
- Empty matrix (`[]`) — The resulting table has no column headings

If the number of columns in the Data property array does not match the number of elements in the ColumnName array, then the number of columns in the resulting table is the larger of the two values.

Example: `{'Name'; 'Telephone Number'}`

Example: `{'Name'; []}`

Example: `{'Name'; 'Telephone|Number'}`

`ColumnWidth` — Width of table columns

`'auto'` (default) | 1-by-n cell array

Width of table columns, specified as a 1-by-n cell array or `'auto'`.

Each column in the cell array corresponds to a column in the table. The values are in pixel units. If you specify `'auto'`, then MATLAB calculates the width of the column automatically using several factors, one of which is the ColumnName property value.

You can combine fixed column widths and `'auto'` column widths in a cell array, or you can specify a single value of `'auto'` to make all column widths automatic.

Selecting **Auto Width** in the Table Property Editor has the same effect as setting the ColumnWidth property to `'auto'`.

Example: `'auto'`

Example: `{64 60 40}`

Example: `{64 'auto' 40}`

`ColumnFormat` — Cell display format

empty cell array ({}) (default) | 1-by-n cell array of strings

Cell display format, specified as an empty cell array or a cell array of strings.

This property determines how the data in each column displays and the constraints for editing that data in the UI. The length of the cell array must equal the number of columns in the table, and the elements of the cell array correspond to columns in the Data property array. If you do not want to specify a display format for a particular column, specify `[]` for that column. If you do not specify a format for a column, MATLAB determines the default display by the data type of the data in the cell.

Elements of the cell array must be one of the strings described in the following table.

Cell Format	Description
`'char'`	Display a left-justified string. To edit a cell, the user types a string that replaces the existing string.
`'logical'`	Display a check box. To edit a cell, the user selects or clears the check box. Then, MATLAB sets the corresponding Data value to `true` or `false` accordingly. The ColumnEditable property value must be `true` to allow users to select or deselect the check boxes. Initially, a check box is selected when the corresponding Data value is `true`. The corresponding values in the Data property array must be of type `logical` to ensure the data displays correctly in the table.
`'numeric'`	Display a right-justified string equivalent to the Command Window display for numeric data. If an element in the Data property array is logical, then `1` or `0` appears in the table. If an element in the Data property array is not numeric and not logical, then `NaN` appears in the table. To edit a cell, the user can enter any string. If a user enters a string representing a constant, such as `pi`, you can code the CellEditCallback function to convert the value to the numeric equivalent. In this case,

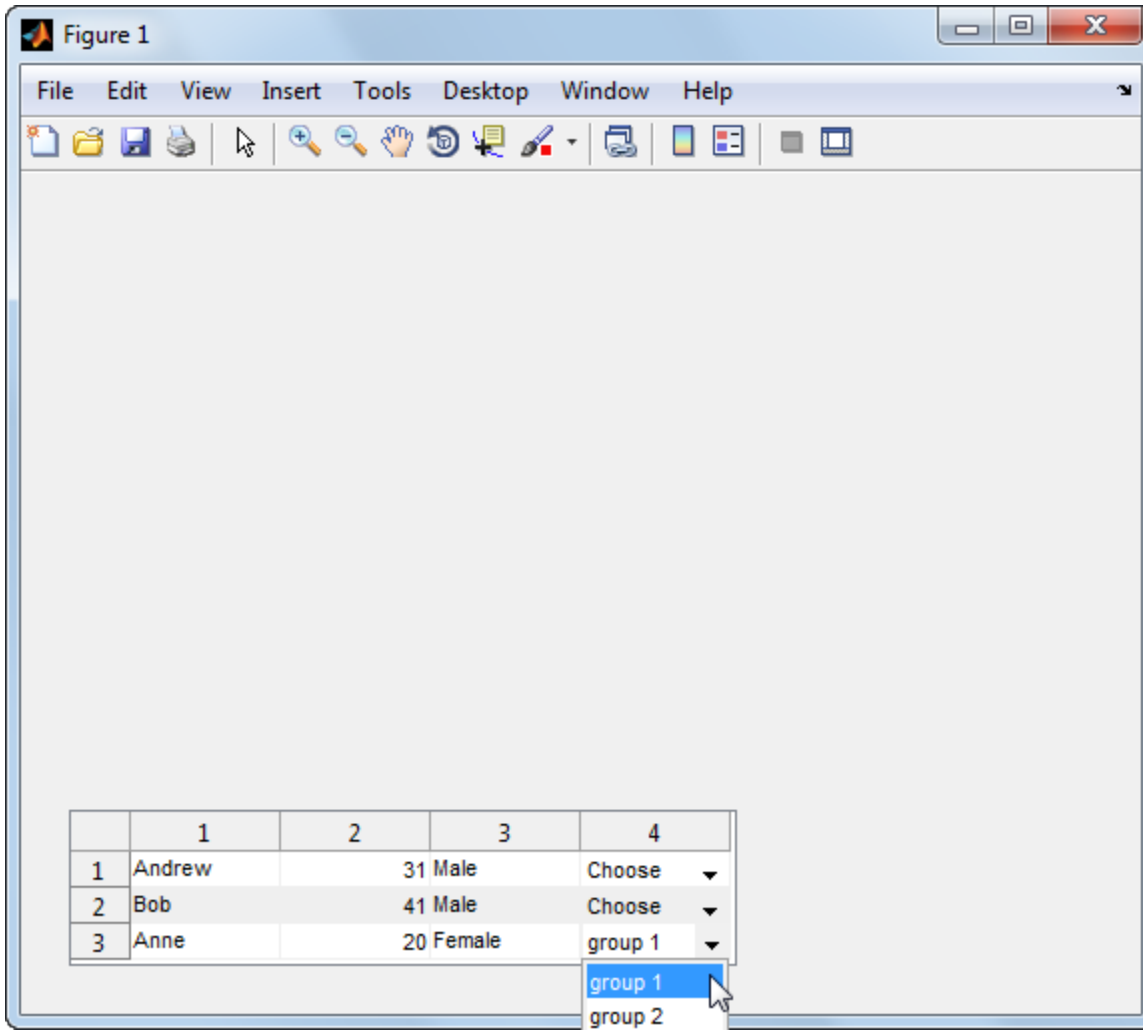
Cell Format	Description
	MATLAB attempts to convert the user-entered string to a numeric value and stores it in the Data property. Then, the CellEditCallback function executes. See the CellEditCallback description for an example.
A 1-by-n cell array of strings. For example, `{'one' 'two' 'three'}`	Display a pop-up menu. To edit a cell, the user selects an item from the pop-up menu. MATLAB sets the corresponding Data property array value to the selected menu item. The ColumnEditable property value must be `true` to allow users to select items in the pop-up menu.
A string accepted by the `format` function, such as: `'short'`, `'bank'`, `'long'`	Display the Data property values using the specified format.

Effect of Pop-Up Menu ColumnFormat and Various Data Types

The initial Data values do not have to be items in the pop-up menu. If the ColumnFormat value defines a pop-up menu, that pop-up menu displays the corresponding value in Data property until the user makes a selection.

For instance, suppose the Data property value for a given column is the string, `'Choose'` in all the rows, and the ColumnFormat value specifies a pop-up menu with the choices of `'group 1'` and `'group 2'`. When MATLAB creates the table, those table cells display `'Choose'` until the user selects an item in the pop-up menu:

```
f = figure;
myData = {'Andrew' 31 'Male' 'Choose'; ...
          'Bob' 41 'Male' 'Choose';  ...
          'Anne' 20 'Female' 'Choose';};
t = uitable('Parent', f,...
            'Position', [25 25 334 78],...
            'ColumnFormat',({[] [] [] {'group 1' 'group 2'}}),...
            'ColumnEditable', true,...
            'Data', myData);
```



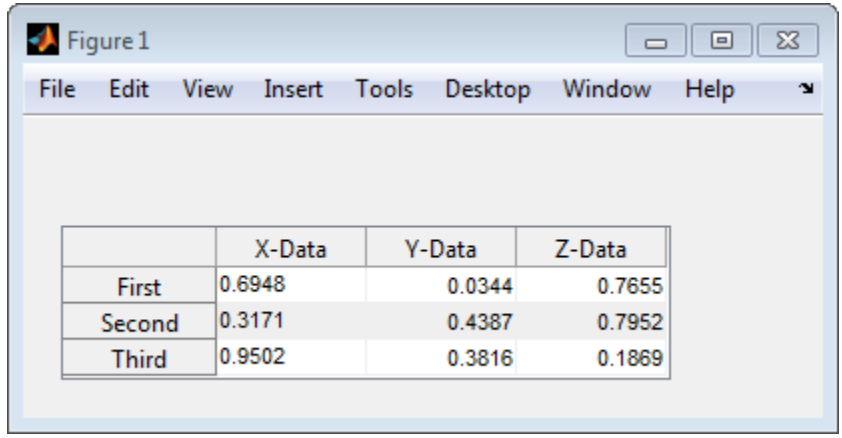
Data Display of Editable Columns

This table describes how various data types display with specific ColumnFormat values.

	ColumnFormat

		'numeric'	'char'	'logical'
Data Type of Data Array Value	Any numeric type	Table displays number as-is.	MATLAB converts the value to a text string and displays it left-justified in the table.	Not recommended. MATLAB might return a warning when the user edits the cell, unless you define a CellEditCallback function.
	`char`	MATLAB converts the text string to a `double` value. If MATLAB cannot convert the string, then**NaN** displays. See `str2double` for more information.	Table displays the string as is.	Not recommended. MATLAB might return a warning when the user edits the cell, unless you define a CellEditCallback function.
	`logical`	Table displays logical values as numbers. MATLAB might return a warning when the user edits the cell, unless you define a CellEditCallback function.	Table displays `'true'` or `'false'`. MATLAB might return a warning when the user edits the cell, unless you define a CellEditCallback function.	Table displays logical values as check boxes.

For example, in the following table, the first column (X-Data) is left justified because the ColumnFormat value for that column is `'char'`.



`RowStriping` — Alternate row shading
`'on'` (default) | `'off'`

Alternate row shading, specified as `'on'` or `'off'`. This property controls the shading pattern of the table rows.

When the RowStriping value is set to `'on'`, the BackgroundColor matrix specifies the background color of consecutive rows . The first color matrix row defines the shading for odd-numbered rows in the table. The second color matrix row defines the shading for even-numbered rows. If the BackgroundColor matrix has only one row, then the shading is the same in all table rows.

When RowStriping is set to `'off'`, then the first color in the BackgroundColor property defines the shading for all rows in the table.

See Also
`uitable`

More About
- "Access Property Values"
- "Default Property Values"

uitoggletool

Create toggle button on toolbar

Syntax

```
t = uitoggletool
t = uitoggletool(Name,Value,...)
t = uitoggletool(parent)
t = uitoggletool(parent,Name,Value,...)
```

Description

`t = uitoggletool` creates a uitoggletool in the current figure's uitoolbar and returns the uitoggletool object, `t`. If there is no uitoolbar available, then MATLAB creates a new uitoolbar in the current figure to serve as the parent. Similarly, if there no figure is available, then MATLAB creates a new figure with a uitoolbar.

`t = uitoggletool(Name,Value,...)` creates a uitoggletool and specifies one or more uitoggletool property names and corresponding values. Use this syntax to override the default uitoggletool properties.

`t = uitoggletool(parent)` creates a uitoggletool and designates a specific parent object. The `parent` argument must be a uitoolbar object.

`t = uitoggletool(parent,Name,Value,...)` creates a uitoggletool with a specific parent and one or more uitoggletool properties.

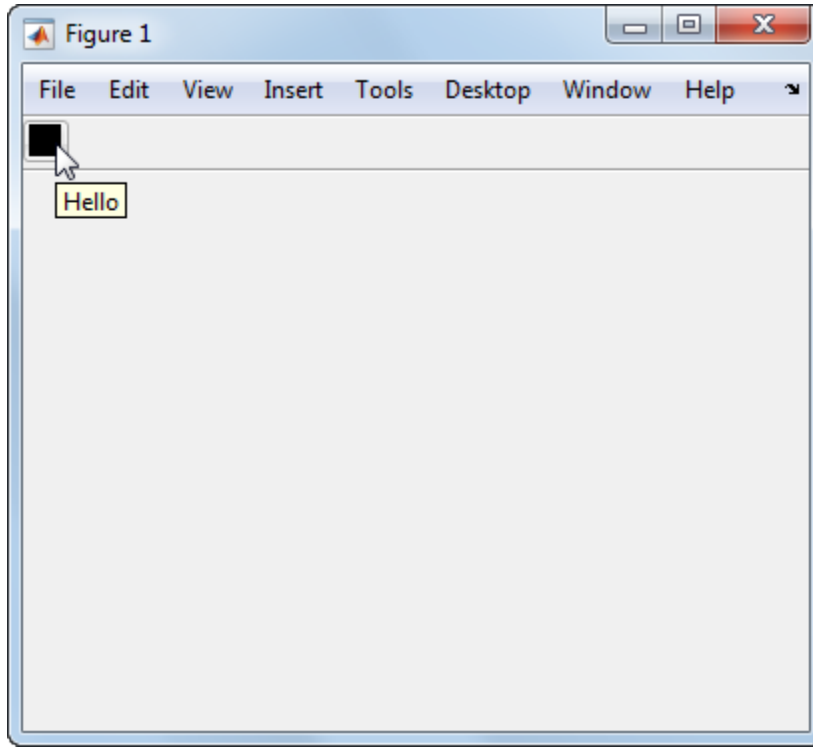
A uitoggletool is a toggle button that appears in the figure's tool bar. The button has no icon, but its borders highlight when the user hovers over it with the mouse. You can create a button icon by setting the uitoggletool's `CData` property.

Examples

This example creates a `uitoolbar` and places a `uitoggletool` inside it.

```
f = figure('ToolBar','none');
tb = uitoolbar(f);
img = zeros(16,16,3);
```

```
t = uitoggletool(tb,'CData',img,'TooltipString','Hello');
```



The `CData` property is set to a 16-by-16-by-3 array in which all elements are `0`. The zero values make the uitoggletool appear black.

Alternatives

You can create toolbars with toggle tools using GUIDE.

More About

Tips

- Uitoolbars (and their child uitoggletools) do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a uitoolbar is

changed to `'Modal'`, the uitoolbar still exists in the `Children` property of the figure. However, the uitoolbar does not display while WindowStyle is set to `'Modal'`.

- "Access Property Values"
- "Create Toolbars for Programmatic UIs"

See Also

`uipushtool` | Uitoggletool Properties | `uitoolbar`

Introduced before R2006a

Uitoggletool Properties

Control appearance and behavior of uitoggletool

Uitoggletools are toggle buttons that appear on the tool bar at the top of the a figure. The `uitoggletool` function creates a toggle button on a tool bar and sets any required properties before displaying it. By changing uitoggletool property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
t = uitoggletool;
sep = t.Separator;
t.Separator = 'on';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uitoggletool visibility

`'on'` (default) | `'off'`

Uitoggletool visibility, specified as `'on'` or `'off'`. When the Visible property is set to `'off'`, the uitoggletool is not visible, but you can query and set its properties.

`Separator` — Separator line mode

`'off'` (default) | `'on'`

Separator line mode, specified as `'off'` or `'on'`. Setting this property to `'on'` draws a dividing line to left of the uitoggletool.

`CData` — Optional image to display on uitoggletool

3-D array of truecolor RGB values

Optional image to display on the uitoggletool, specified as a 3-D array of truecolor RGB values. The values in the array can be:

- Double-precision values between `0.0` and `1.0`
- `uint8` values between `0` and `255`

The length of the array's first and second dimensions must be less than or equal to 16. Otherwise, it might be clipped or distorted when it displays.

Data Types: `double` | `uint8`

Interactive Control

`State` — Toggle button state

`'off'` (default) | `'on'`

Toggle button state, specified as `'off'` or `'on'`. When this property value is `'on'`, the toggle button appears in the down (depressed) position. When this property value is `'off'`, the toggle button appears in the up position. Changing the state causes the appropriate OnCallback or OffCallback function to execute.

`ClickedCallback` — Callback function that executes when user clicks uitoggletool

`''` (default) | function handle | cell array | string

Callback function that executes when a user clicks the uitoggletool, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

This callback function executes after the OnCallback function or OffCallback function executes (depending on the state of the button).

For more information about specifying callback functions as function handles, cell arrays, or strings, see “How to Specify Callback Property Values”.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle` | `cell` | `char`

`OnCallback` — Callback to execute when user turns on toggle button

`''` (default) | function handle | cell array | string

Callback to execute when user turns on the toggle button, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying callback functions as function handles, cell arrays, or strings, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`OffCallback` — Callback to execute when user turns off the toggle button

`''` (default) | function handle | cell array | string

Callback to execute when user turns off toggle button, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying callback functions as function handles, cell arrays, or strings, see "How to Specify Callback Property Values".

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`Enable` — Operational state of uitoggletool

`'on'` (default) | `'off'`

Operational state of uitoggletool, specified as `'on'` or `'off'`. The `Enable` property controls whether the uitoggletool responds to button clicks. The are two possible values:

- `'on'` – The uitoggletool is operational.
- `'off'` – The uitoggletool is not operational and appears grayed-out.

The value of the `Enable` property and the type of button click determine the response.

Enable Value	Response to Left-Click	Response to Right-Click
`'on'`	**1** The OnCallback or OffCallback function execute, depending on the current state of the uitoggletool. **2** The uitoggletool's `ClickedCallback` function executes.	The uitoggletool is not operational. No callback executes.
`'off'`	The uitoggletool is not operational. No callback executes.	The uitoggletool is not operational. No callback executes.

`TooltipString` — Tooltip text

string

Content of uitoggletool tooltip, specified as a string. When the user moves the mouse pointer over the uitoggletool and leaves it there, the tooltip displays.

Example: `'Some string'`

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu handle

This property has no effect on uitoggletools.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

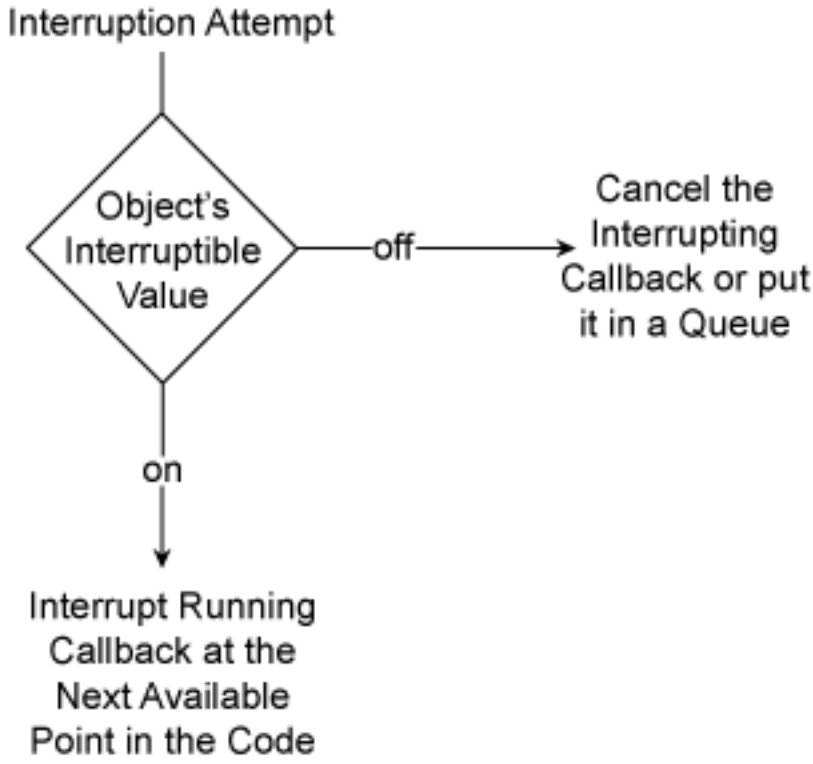
Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback

determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uitoggletool callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

This property has no effect on the uitoggletool.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uitoggletool

`'off'` (default) | `'on'`

Deletion status of uitoggletool, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uitoggletool begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uitoggletool no longer exists.

Check the value of the BeingDeleted property to verify that the uitoggletool is not about to be deleted before querying or modifying it.

`CreateFcn` — Uitoggletool creation function

function handle | cell array | string

Uitoggletool creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uitoggletool. MATLAB initializes all uitoggletool property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uitoggletool that is being created.

Setting the CreateFcn property on an existing uitoggletool has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uitoggletool object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uitoggletool deletion function

function handle | cell array | string

Uitoggletool deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uitoggletool (for example, when the end user deletes the figure). MATLAB executes

the DeleteFcn callback before destroying the properties of the uitoggletool. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uitoggletool that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object

`'uitoggletool'`

Type of graphics object, returned as `'uitoggletool'`.

`Tag` — Uitoggletool identifier

`''` (default) | string

Uitoggletool identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uitoggletool. When you need access to the uitoggletool elsewhere in your code, you can use the `findobj` function to search for the uitoggletool based on the Tag value.

Example: `'toggletool1'`

Data Types: `char`

`UserData` — Data to associate with the uitoggletool object

empty array (default) | array

Data to associate with the uitoggletool object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uitoggletool parent

uitoolbar

Uitoggletool parent, specified as a uitoolbar. You can move a uitoggletool to a different uitoolbar by setting this property to the handle of the target uitoolbar.

`HandleVisibility` — Visibility of Uitoggletool handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uitoggletool handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uitoggletool handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uitoggletool handle is always visible.
`'callback'`	The uitoggletool handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uitoggletool at the command-line, but allows callback functions to access it.
`'off'`	The uitoggletool handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`uitoggletool` | `uitoolbar`

More About

- "Access Property Values"
- "Default Property Values"

uitoolbar

Create toolbar on figure

Syntax

```
t = uitoolbar
t = uitoolbar(Name,Value,...)
t = uitoolbar(parent)
t = uitoolbar(parent,Name,Value,...)
```

Description

`t = uitoolbar` creates a uitoolbar in an existing figure and returns the uitoolbar object, `t`. If there is no figure available, then MATLAB creates a new figure to serve as the parent.

`t = uitoolbar(Name,Value,...)` creates a uitoolbar and specifies one or more uitoolbar property names and corresponding values. Use this syntax to override the default uitoolbar properties.

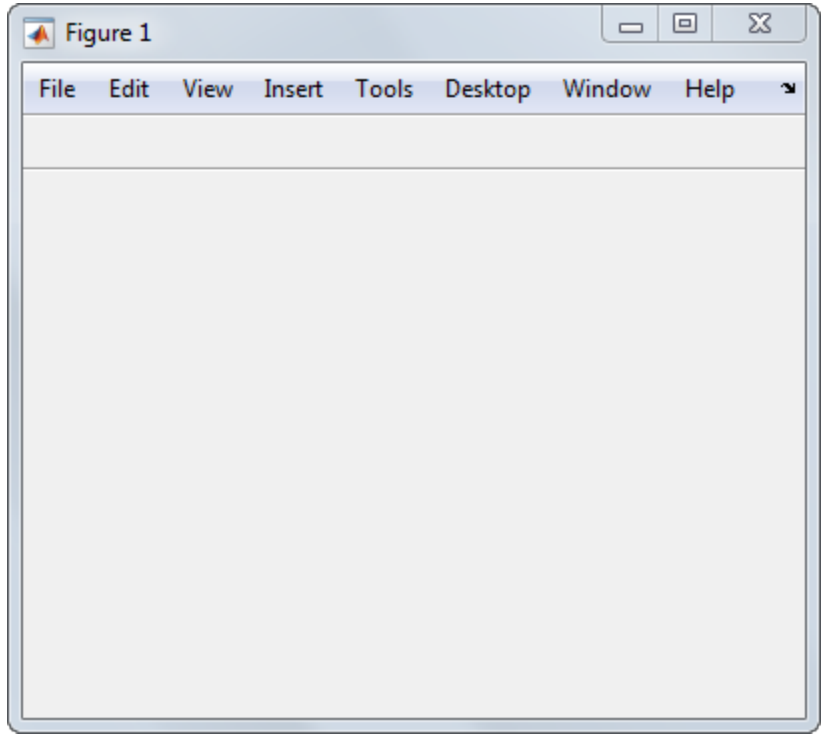
`t = uitoolbar(parent)` creates a uitoolbar in a specific parent figure. The `parent` input argument is the parent figure object.

`t = uitoolbar(parent,Name,Value,...)` creates a uitoolbar with a specific parent and one or more uitoolbar properties.

Examples

This example creates a figure containing an empty uitoolbar.

```
f = figure('ToolBar','none')
t = uitoolbar(f)
```



More About

Tips

Uitoolbars do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a uitoolbar child is changed to `'Modal'`, the uitoolbar child still exist in the `Children` property of the figure. However, the uitoolbar does not display while WindowStyle is set to `'Modal'`.

- "Create Toolbars for Programmatic UIs"
- "Create Toolbars for GUIDE UIs"
- "Access Property Values"

See Also

`figure` | `uipushtool` | `uitoggletool` | Uitoolbar Properties

Introduced before R2006a

Uitoolbar Properties

Control appearance and behavior tool bar

A uitoolbar is a tool bar containing buttons at the top of a figure window. The `uitoolbar` function creates a tool bar in a figure and sets any required properties before displaying it. By changing uitoolbar property values, you can modify certain aspects of its appearance and behavior.

Starting in R2014b, you can use dot notation to query and set properties.

```
t = uitoolbar;
isvisible = t.Visible;
t.Visible = 'off';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Appearance

`Visible` — Uitoolbar visibility

`'on'` (default) | `'off'`

Uitoolbar visibility, specified as `'on'` or `'off'`. When the Visible property is set to `'off'`, the uitoolbar is not visible, but you can query and set its properties.

Interactive Control

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu handle

This property has no effect on uitoolbars.

Callback Execution Control

`Interruptible` — Callback interruption

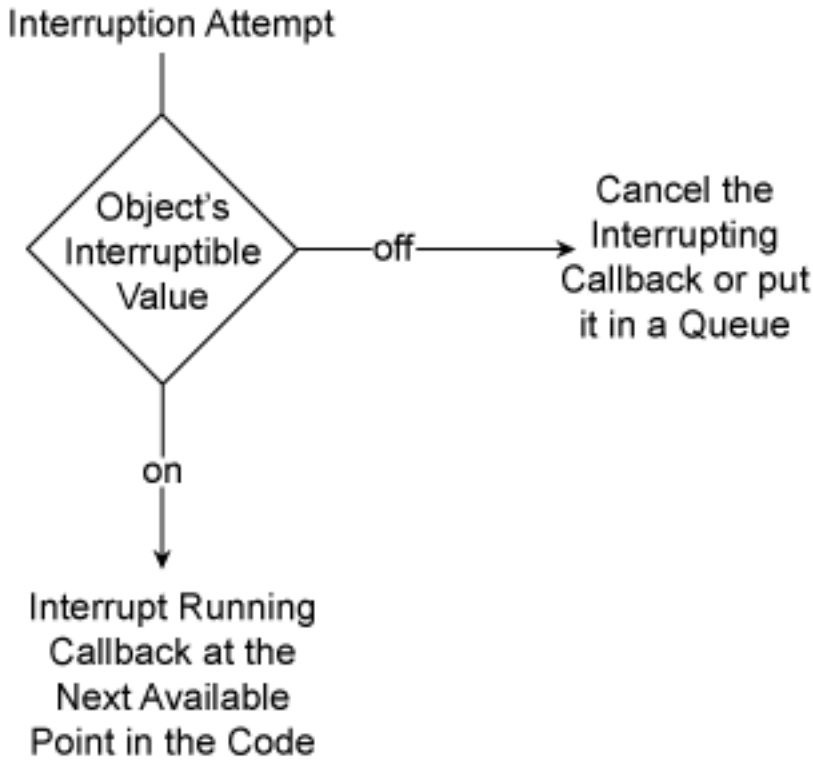
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The Interruptible property determines if a running callback can be interrupted.

There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback. The Interruptible property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put into a queue.



If a uitoolbar callback is the running callback, then the Interruptible property determines if it can be interrupted by another callback. The Interruptible property has two possible values:

- `'on'` — A callback can interrupt the running callback. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — A callback cannot interrupt the running callback. MATLAB finishes executing the running callback without any interruptions. This is the default behavior.

Note: Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a DeleteFcn, CloseRequestFcn, or SizeChangedFcn callback, then the interruption occurs regardless of the Interruptible property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the Interruptible property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the Interruptible and BusyAction properties affect the behavior of a program.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The *running* callback is the currently executing callback.
- The *interrupting* callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

This property has no effect on the uitoolbar.

Creation and Deletion Control

`BeingDeleted` — Deletion status of uitoolbar

`'off'` (default) | `'on'`

Deletion status of uitoolbar, returned as `'on'` or `'off'`. MATLAB sets the BeingDeleted property to `'on'` when the delete function of the uitoolbar begins execution (see the DeleteFcn property). The BeingDeleted property remains set to `'on'` until the uitoolbar no longer exists.

Check the value of the BeingDeleted property to verify that the uitoolbar is not about to be deleted before querying or modifying it.

`CreateFcn` — Uitoolbar creation function

function handle | cell array | string

Uitoolbar creation function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

This property specifies a callback function to execute when MATLAB creates the uitoolbar. MATLAB initializes all uitoolbar property values before executing the CreateFcn callback. If you do not specify the CreateFcn property, then MATLAB executes a default creation function.

Use the `gcbo` function in your CreateFcn code to get the handle to the uitoolbar that is being created.

Setting the CreateFcn property on an existing uitoolbar has no effect.

Note: Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a CreateFcn. Copying the uitoolbar object causes the CreateFcn callback to execute repeatedly.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

`DeleteFcn` — Uitoolbar deletion function

function handle | cell array | string

Uitoolbar deletion function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- String that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or string, see "How to Specify Callback Property Values".

The DeleteFcn property specifies a callback function to execute when MATLAB deletes the uitoolbar (for example, when the end user deletes the figure). MATLAB executes the DeleteFcn callback before destroying the properties of the uitoolbar. If you do not specify the DeleteFcn property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your DeleteFcn code to get the handle to the uitoolbar that is being deleted.

Example: `@myfun`

Example: `{@myfun,x}`

Data Types: `function_handle | cell | char`

Identifiers

`Type` — Type of graphics object
`'uitoolbar'`

Type of graphics object, returned as `'uitoolbar'`.

`Tag` — Uitoolbar identifier
`''` (default) | string

Uitoolbar identifier, specified as a string. You can specify a unique Tag value to serve as an identifier for the uitoolbar. When you need access to the uitoolbar elsewhere in your code, you can use the `findobj` function to search for the uitoolbar based on the Tag value.

Example: `'toolbar1'`

Data Types: `char`

`UserData` — Data to associate with the uitoolbar object
empty array (default) | array

Data to associate with the uitoolbar object, specified as any array. Specifying UserData can be useful for sharing data values within and across UIs. See “Share Data Among Callbacks” for more information.

Example: `[1 2 3]`

Example: `'April 21'`

Example: `struct('value1',[1 2 3],'value2','April 21')`

Example: `{[1 2 3],'April 21'}`

Parent/Child

`Parent` — Uitoolbar parent

figure

Uitoolbar parent, specified as a figure. You can move a uitoolbar to a different figure by setting this property to the handle of the target figure.

`Children` — Children of uitoolbar

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Children of uitoolbar, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of uitoolbars are uipushtools and uitoggletools.

You cannot add or remove children using the `Children` property of the uitoolbar. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the components displayed in the Uitoolbar.

To add a child to this list, set the `Parent` property of the child component to be the uitoolbar object.

Objects with the HandleVisibility property set to `'off'` do not list in the Children property. For more information, see the HandleVisibility property description.

`HandleVisibility` — Visibility of Uitoolbar handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of Uitoolbar handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the uitoolbar handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions

that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The HandleVisibility property also controls the visibility of the object's handle in the parent figure's CurrentObject property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The uitoolbar handle is always visible.
`'callback'`	The uitoolbar handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the uitoolbar at the command-line, but allows callback functions to access it.
`'off'`	The uitoolbar handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the HandleVisibility to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root ShowHiddenHandles property to `'on'` to make all handles visible, regardless of their HandleVisibility value. This setting has no effect on their HandleVisibility values.

See Also

`uitoolbar`

More About

- "Access Property Values"
- "Default Property Values"

uiwait

Block program execution and wait to resume

Syntax

```
uiwait
uiwait(h)
uiwait(h,timeout)
```

Description

`uiwait` blocks execution until `uiresume` is called or the current figure is deleted. This syntax is the same as `uiwait(gcf)`.

`uiwait(h)` blocks execution until `uiresume` is called or the figure `h` is deleted.

`uiwait(h,timeout)` blocks execution until `uiresume` is called, the figure `h` is deleted, or `timeout` seconds elapse. The minimum value of `timeout` is 1. If `uiwait` receives a smaller value, it issues a warning and uses a 1 second `timeout`.

Examples

This example creates a UI with a **Continue** push button. The example calls `uiwait` to block MATLAB execution until `uiresume` is called. This happens when the user clicks the **Continue** push button because the push button's `Callback`, which responds to the click, calls `uiresume`.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

`gcbf` is the handle of the figure that contains the object whose callback is executing.

More About

Tips

The `uiwait` and `uiresume` functions block and resume MATLAB and Simulink program execution. `uiwait` also blocks the execution of Simulink models. The functions `pause` (with no argument) and `waitfor` also block execution in this manner. `uiwait` is a convenient way to use the `waitfor` command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the MATLAB program that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, `uiwait` can block the execution of the program file *and* restrict user interaction to the dialog only.

See Also

`dialog` | `figure` | `uicontrol` | `uimenu` | `uiresume` | `waitfor`

Introduced before R2006a

uminus, -

Unary minus

Syntax

```
C = -A
C = uminus(A)
```

Description

`C = -A` negates the elements of `A` and stores the result in `C`.

`C = uminus(A)` is an alternative way to execute `-A`, but is rarely used. It enables operator overloading for classes.

Examples

Negate Elements of Matrix

Create a 2-by-2 matrix, `A`.

```
A = [1 -3; -2 4]
```

```
A =

     1    -3
    -2     4
```

Negate the elements of `A`.

```
C = -A
```

```
C =

    -1     3
```

```
     2    -4
```

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. A can be a numeric array, logical array, character array, duration array, or calendar duration array.
Complex Number Support: Yes

More About

- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`minus` | `uplus`

undocheckout

(To be removed) Undo previous checkout from source control system (UNIX platforms)

Note: `undocheckout` will be removed in a future release.

Syntax

```
undocheckout('filename')
undocheckout({'filename1','filename2', ...,'filenamen'})
```

Description

`undocheckout('filename')` makes the file `filename` available for checkout, where `filename` does not reflect any of the changes you made after you last checked it out. Use the full path for `filename` and include the file extension.

`undocheckout({'filename1','filename2', ...,'filenamen'})` makes `filename1` through `filenamen` available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full paths for the file names and include the file extensions.

Examples

Undo the checkouts of `/myserver/myfiles/clock.m` and `/myserver/myfiles/calendar.m` from the source control system:

```
undocheckout({'/myserver/myfiles/clock.m', ...
'/myserver/myfiles/calendar.m'})
```

Introduced before R2006a

unicode2native

Convert Unicode character representation to numeric bytes

Syntax

```
bytes = unicode2native(unicodestr)
bytes = unicode2native(unicodestr,encoding)
```

Description

`bytes = unicode2native(unicodestr)` converts a `char` vector of Unicode character representations, `unicodestr`, to the user default encoding, and returns the bytes as a `uint8` vector, `bytes`. Output vector `bytes` has the same general array shape as the `unicodestr` input. You can save the output of `unicode2native` to a file using the `fwrite` function.

`bytes = unicode2native(unicodestr,encoding)` converts `unicodestr` to the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. If `encoding` is unspecified or is the empty string (`''`), the default encoding scheme is used.

Examples

This example begins with two strings containing Unicode character representations. It assumes that string `str1` contains text in a Western European language and string `str2` contains Japanese text. The example writes both strings into the same file, using the ISO-8859-1 character encoding scheme for the first string and the Shift-JIS encoding scheme for the second string. The example uses `unicode2native` to convert the two strings to the appropriate encoding schemes.

```
fid = fopen('mixed.txt', 'w');
bytes1 = unicode2native(str1, 'ISO-8859-1');
fwrite(fid, bytes1, 'uint8');
bytes2 = unicode2native(str2, 'Shift_JIS');
```

```
fwrite(fid, bytes2, 'uint8');
fclose(fid);
```

See Also

native2unicode

Introduced before R2006a

union

Set union of two arrays

Syntax

```
C = union(A,B)
C = union(A,B,'rows')
[C,ia,ib] = union(A,B)
[C,ia,ib] = union(A,B,'rows')

[C,ia,ib] = union( ___ ,setOrder)

[C,ia,ib] = union(A,B,'legacy')
[C,ia,ib] = union(A,B,'rows','legacy')
```

Description

`C = union(A,B)` returns the combined data from `A` and `B` with no repetitions.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then `union` returns the combined values from `A` and `B`. The values of `C` are in sorted order.
- If `A` and `B` are tables, then `union` returns the combined set of rows from both tables. The rows of table `C` are in sorted order.

`C = union(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the combined rows from `A` and `B` with no repetitions. The rows of `C` are in sorted order.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia,ib] = union(A,B)` also returns index vectors `ia` and `ib`.

- If `A` and `B` are numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, or cell arrays of strings, then the values in `C` are a sorted combination of the values of `A(ia)` and `B(ib)`.

- If A and B are tables, then C is a sorted combination of the rows of A(ia,:) and B(ib,:).

[C,ia,ib] = union(A,B,'rows') also returns index vectors ia and ib, such that the rows of C are a sorted combination of the rows of A(ia,:) and B(ib,:).

[C,ia,ib] = union(___ ,setOrder) returns C in a specific order using any of the input arguments in the previous syntaxes. setOrder='sorted' returns the values (or rows) of C in sorted order. setOrder='stable' returns the values (or rows) of C in the same order as A, and then B. If no value is specified, the default is 'sorted'.

[C,ia,ib] = union(A,B,'legacy') and [C,ia,ib] = union(A,B,'rows','legacy') preserve the behavior of the union function from R2012b and prior releases.

The 'legacy' option does not support categorical arrays, tables, datetime arrays, or duration arrays.

Examples

Union of Two Vectors

Define two vectors with a value in common.

```
A = [5 7 1]; B = [3 1 1];
```

Find the union of vectors A and B.

```
C = union(A,B)
```

```
C =

     1     3     5     7
```

Union of Two Tables

Define two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
A =
```

```
    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    2       B       true
    3       C       false
    4       D       true
    5       E       false


B =

    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    3       C       false
    5       E       false
    7       G       false
    9       I       false
```

Find the union of tables `A` and `B`.

```
C = union(A,B)
```

```
C =

    Var1    Var2    Var3
    ----    ----    -----
    1       A       false
    2       B       true
    3       C       false
    4       D       true
    5       E       false
    7       G       false
    9       I       false
```

Union of Two Vectors and Their Indices

Define two vectors with a value in common.

```
A = [5 7 1]; B = [3 1 1];
```

Find the union of vectors `A` and `B`, as well as the index vectors, `ia` and `ib`.

```
[C,ia,ib] = union(A,B)
```

```
C =

     1     3     5     7


ia =

     3
     1
     2


ib =

     1
```

The values in `C` are the combined values of `A(ia)` and `B(ib)`.

Union of Two Tables and Their Indices

Define a table, `A`, of gender, age, and height for three people.

```
A = table(['M';'M';'F'],[27;52;31],[74;68;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty'})
```

```
A =

              Gender    Age    Height
              ------    ---    ------
    Ted       M         27     74
    Fred      M         52     68
    Betty     F         31     64
```

Define a table, `B` with the same variables as `A`.

```
B = table(['F';'M'],[64;68],[31;47],...
'VariableNames',{'Gender' 'Height' 'Age'},...
'RowNames',{'Meg' 'Joe'})
```

```
B =

            Gender    Height    Age
            ------    ------    ---
    Meg     F         64        31
```

```
    Joe    M         68        47
```

Find the union of tables `A` and `B`, as well as the index vectors, `ia` and `ib`.

```
[C,ia,ib] = union(A,B)
```

```
C =

            Gender    Age    Height
            ------    ---    ------
    Betty   F         31     64
    Ted     M         27     74
    Joe     M         47     68
    Fred    M         52     68


ia =

     3
     1
     2


ib =

     2
```

The data for `Meg` and `Betty` are the same. `union` only returns the index from `A`, which corresponds to `Betty`.

Union of Rows in Two Matrices

Define two matrices with a row in common.

```
A = [2 2 2; 0 0 1];
B = [1 2 3; 2 2 2; 2 2 2];
```

Find the combined rows of `A` and `B`, with no repetition, as well as the index vectors `ia` and `ib`.

```
[C,ia,ib] = union(A,B,'rows')
```

```
C =

     0     0     1
```

```
     1     2     3
     2     2     2


ia =

     2
     1


ib =

     1
```

The rows of `C` are the combined rows of `A(ia,:)` and `B(ib,:)`.

Union of Two Vectors with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` if you want the values in `C` to have the same order as in `A` and `B`.

```
A = [5 7 1]; B = [3 1 1];
[C,ia,ib] = union(A,B,'stable')
```

```
C =

     5     7     1     3


ia =

     1
     2
     3


ib =

     1
```

Alternatively, you can specify `'sorted'` order.

```
A = [5 7 1]; B = [3 1 1];
[C,ia,ib] = union(A,B,'sorted')
```

```
C =

     1     3     5     7


ia =

     3
     1
     2


ib =

     1
```

Union of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN 1]; B = [4 NaN NaN];
```

Find the union of vectors `A` and `B`.

```
C = union(A,B)
```

```
C =

     1     4     5   NaN   NaN   NaN
```

`union` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Combine the elements of `A` and `B`.

```
[C,ia,ib] = union(A,B)
```

```
C =

    'cat'    'dog'    'dog '    'fish'    'fish '    'horse'


ia =

     2
     1
     3
     4


ib =

     1
     3
```

`union` treats trailing white space in cell arrays of strings as distinct characters.

Union of Vectors of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A =

A
B
C


ans =

char
```

Create a row vector containing elements of numeric type `double`.

```
B = [68 69 70], class(B)
```

```
B =
```

```
    68    69    70

ans =

double
```

The `union` of `A` and `B` returns a column vector character array.

```
C = union(A,B), class(C)


C =

A
B
C
D
E
F


ans =

char
```

Union of Char and Cell Array of Strings

Create a character array containing the letters `a` , `b`, and `c`.

```
A = ['a';'b';'c'];
class(A)

ans =

char
```

Create a cell array of strings containing the letters `c`, `d`, and `e`.

```
B = {'c','d','e'};
class(B)

ans =

cell
```

Combine the elements of A and B.

```
C = union(A,B)
```

```
C =

    'a'
    'b'
    'c'
    'd'
    'e'
```

The result, C, is a cell array of strings.

```
class(C)
```

```
ans =

cell
```

Preserve Legacy Behavior of union

Use the `'legacy'` flag to preserve the behavior of `union` from R2012b and prior releases in your code.

Find the union of A and B with the current behavior.

```
A = [5 7 1]; B = [3 1 1];
[C1,ia1,ib1] = union(A,B)
```

```
C1 =

     1     3     5     7


ia1 =

     3
     1
     2


ib1 =

     1
```

Find the union of A and B, and preserve the legacy behavior.

```
A = [5 7 1]; B = [3 1 1];
[C2,ia2,ib2] = union(A,B,'legacy')
```

```
C2 =

     1     3     5     7


ia2 =

     1     2


ib2 =

     3     1
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of strings | tables

Input arrays, specified as numeric arrays, logical arrays, character arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of strings, or tables.

A and B must be of the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.
- Categorical arrays can combine with cell arrays of strings or single strings.
- Datetime arrays can combine with cell arrays of date strings or single date strings.

If A and B are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither A nor B are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of C are the sorted union of the categories from A and B.

If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

If `A` and `B` are tables, they must have the same variable names. Conversely, the row names do not matter. Two rows that have the same values, but different names, are considered equal.

If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

Furthermore, `A` and `B` can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class.

`setOrder` — Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Order Flag	Meaning
`'sorted'`	The values (or rows) in `C` return in sorted order. For example: `C = union([5 5 3],[1 2],'sorted')` returns `C = [1 2 3 5]`.
`'stable'`	The values (or rows) in `C` return in the same order as they appear in `A` and `B`. For example: `C = union([5 5 3],[1 2],'stable')` returns `C = [5 3 1 2]`.

Output Arguments

`C` — Combined data of `A` and `B`

vector | matrix | table

Combined data of `A` and `B`, returned as a vector, matrix, or table. If the inputs `A` and `B` are tables, the order of the variables in the resulting table, `C`, is the same as the order of the variables in `A`.

The following describes the shape of `C` when the inputs are vector or matrices and when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified, then `C` is a column vector unless both `A` and `B` are row vectors.
- If the `'rows'` flag is not specified and both `A` and `B` are row vectors, then `C` is a row vector.
- If the `'rows'` flag is specified, then `C` is a matrix containing the combined rows of `A` and `B`.

The class of the inputs `A` and `B` determines the class of `C`:

- If the class of `A` and `B` are the same, then `C` is the same class.
- If you combine a `char` or nondouble numeric class with `double`, then `C` is the same class as the nondouble input.
- If you combine a `logical` class with `double`, then `C` is `double`.
- If you combine a cell array of strings with `char`, then `C` is a cell array of strings.
- If you combine a categorical array with a cell array of strings or single string, then `C` is a categorical array.
- If you combine a datetime array with a cell array of date strings or single date string, then `C` is a datetime array.

`ia` — Index to `A`

column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` indicates the values (or rows) in `A` that contribute to the union. If a value (or row) appears multiple times in `A`, then `ia` contains the index to the first occurrence of the value (or row). If a value appears in both `A` and `B`, then `ia` contains the index to the first occurrence in `A`.

`ib` — Index to `B`

column vector

Index to `B`, returned as a column vector when the `'legacy'` flag is not specified. `ib` indicates the values (or rows) in `B` that contribute to the union. If there is a repeated value (or row) appearing exclusively in `B`, then `ib` contains the index to the first occurrence of the value. If a value (or row) appears in both `A` and `B`, then `ib` does not contain an index to the value (or row).

More About

Tips

- To find the union with respect to a subset of variables from a table, you can use column subscripting. For example, you can use `union(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.
- "Combine Categorical Arrays"

See Also

`intersect` | `ismember` | `issorted` | `setdiff` | `setxor` | `sort` | `unique`

Introduced before R2006a

unique

Unique values in array

Syntax

```
C = unique(A)
C = unique(A,'rows')
[C,ia,ic] = unique(A)
[C,ia,ic] = unique(A,'rows')

[C,ia,ic] = unique(A,setOrder)
[C,ia,ic] = unique(A,'rows',setOrder)

[C,ia,ic] = unique(A,'legacy')
[C,ia,ic] = unique(A,'rows','legacy')
[C,ia,ic] = unique(A,occurrence,'legacy')
[C,ia,ic] = unique(A,'rows',occurrence,'legacy')
```

Description

`C = unique(A)` returns the same data as in `A`, but with no repetitions.

- If A is a numeric array, logical array, character array, categorical array, datetime array, duration array, or a cell array of strings, then `unique` returns the unique values in A. The values of C are in sorted order.
- If A is a table, then `unique` returns the unique rows in A. The rows of table C are in sorted order.

`C = unique(A,'rows')` treats each row of A as a single entity and returns the unique rows of A. The rows of the array C are in sorted order.

The `'rows'` option does not support cell arrays.

`[C,ia,ic] = unique(A)` also returns index vectors ia and ic.

- If A is a vector, then `C = A(ia)` and `A = C(ic)`.
- If `A` is a matrix or array, then `C = A(ia)` and `A(:) = C(ic)`.

- If A is a table, then `C = A(ia,:)` and `A = C(ic,:)`.

`[C,ia,ic] = unique(A,'rows')` also returns index vectors ia and ic, such that `C = A(ia,:)` and `A = C(ic,:)`.

`[C,ia,ic] = unique(A,setOrder)` and `[C,ia,ic] = unique(A,'rows', setOrder)` return C in a specific order. `setOrder='sorted'` returns the values (or rows) of C in sorted order. `setOrder='stable'` returns the values (or rows) of C in the same order as A. If no value is specified, the default is `'sorted'`.

`[C,ia,ic] = unique(A,'legacy')`, `[C,ia,ic] = unique(A,'rows','legacy')`, `[C,ia,ic] = unique(A,occurrence,'legacy')`,and `[C,ia,ic] = unique( A,'rows',occurrence,'legacy')` preserve the behavior of the `unique` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays or tables.

Examples

Unique Values in Vector

Define a vector with a repeated value.

```
A = [9 2 9 5];
```

Find the unique values of A.

```
C = unique(A)
```

```
C =

     2     5     9
```

Unique Rows in Table

Define a table with repeated data.

```
Name = {'Fred';'Betty';'Bob';'George';'Jane'};
Age = [38;43;38;40;38];
Height = [71;69;64;67;64];
Weight = [176;163;131;185;131];

A = table(Age,Height,Weight,'RowNames',Name)
```

```
A =

              Age    Height    Weight
              ---    ------    ------
    Fred      38     71        176
    Betty     43     69        163
    Bob       38     64        131
    George    40     67        185
    Jane      38     64        131
```

Find the unique rows of `A`.

```
C = unique(A)
```

```
C =

              Age    Height    Weight
              ---    ------    ------
    Bob       38     64        131
    Fred      38     71        176
    George    40     67        185
    Betty     43     69        163
```

`unique` returns the rows of `A` in sorted order by the first variable, `Age` and then by the second variable, `Height`.

Unique Values and Their Indices

Define a vector with a repeated value.

```
A = [9 2 9 5];
```

Find the unique values of `A` and the index vectors `ia` and `ic`, such that `C = A(ia)` and `A = C(ic)`.

```
[C, ia, ic] = unique(A)
```

```
C =

     2     5     9


ia =

     2
     4
```

```
     1


ic =

     3
     1
     3
     2
```

Unique Rows in Matrix

Define a matrix with a repeated row.

```
A = [9 2 9 5; 9 2 9 0; 9 2 9 5];
```

Find the unique rows of `A` and the index vectors `ia` and `ic`, such that `C = A(ia,:)` and `A = C(ic,:)`.

```
[C, ia, ic] = unique(A,'rows')
```

```
C =

     9     2     9     0
     9     2     9     5


ia =

     2
     1


ic =

     2
     1
     2
```

Notice that the second row in `A` is identified as unique even though the elements `9`, `2`, and `9` are repeated in the other rows.

Unique Values in Vector with Specified Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` if you want the values in `C` to have the same order as in `A`.

```
A = [9 2 9 5];
[C, ia, ic] = unique(A,'stable')
```

```
C =

     9     2     5


ia =

     1
     2
     4


ic =

     1
     2
     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C, ia, ic] = unique(A,'sorted')
```

```
C =

     2     5     9


ia =

     2
     4
     1


ic =

     3
     1
     3
```

```
     2
```

Unique Values in Array Containing NaNs

Define a vector containing `NaN`.

```
A = [5 5 NaN NaN];
```

Find the unique values of `A`.

```
C = unique(A)
```

```
C =

     5   NaN   NaN
```

`unique` treats `NaN` values as distinct.

Unique Entries in Cell Array of Strings

Define a cell array of strings.

```
A = {'one','two','twenty-two','One','two'};
```

Find the unique strings contained in `A`.

```
C = unique(A)
```

```
C =

    'One'    'one'    'twenty-two'    'two'
```

Cell Array of Strings with Trailing White Space

Define a cell array of strings, `A`, where some of the strings have trailing white space.

```
A = {'dog','cat','fish','horse','dog ','fish '};
```

Find the unique strings contained in `A`.

```
C = unique(A)
```

```
C =

    'cat'    'dog'    'dog '    'fish'    'fish '    'horse'
```

`unique` treats trailing white space in cell arrays of strings as distinct characters.

Preserve Legacy Behavior of unique

Use the `'legacy'` flag to preserve the behavior of `unique` from R2012b and prior releases in your code.

Find the unique elements of `A` with the current behavior.

```
A = [9 2 9 5];
[C1, ia1, ic1] = unique(A)
```

```
C1 =

     2     5     9


ia1 =

     2
     4
     1


ic1 =

     3
     1
     3
     2
```

Find the unique elements of `A`, and preserve the legacy behavior.

```
[C2, ia2, ic2] = unique(A, 'legacy')
```

```
C2 =

     2     5     9


ia2 =

     2     4     3
```

```
ic2 =

     3     1     3     2
```

Input Arguments

A — Input array

numeric array | logical array | character array | categorical array | datetime arrays | duration arrays | cell array of strings | table

Input array, specified as a numeric array, logical array, character array, categorical array, datetime array, duration array, cell array of strings, or table.

If A is a table, `unique` does not take row names into account. Two rows that have the same values, but different names, are considered equal.

Furthermore, A can be an object with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class.

`setOrder` — Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in C.

Order Flag	Meaning
`'sorted'`	The values (or rows) in C return in sorted order. For example: `C = unique([5 5 3 4],'sorted')` returns `C = [3 4 5]`.
`'stable'`	The values (or rows) in C return in the same order as in A. For example: `C = unique([5 5 3 4],'stable')` returns `C = [5 3 4]`.

`occurrence` — Occurrence flag for legacy behavior

`'last'` (default) | `'first'`

Occurrence flag for legacy behavior, specified as `'first'` or `'last'`, indicates whether ia should contain the first or last indices to repeated values found in A.

Occurrence Flag	Meaning
`'last'`	If there are repeated values (or rows) in A, then ia contains the index to the last occurrence of the repeated value. For example: `[C,ia,ic] = unique([9 9 9],'last','legacy')` returns `ia = 3`.
`'first'`	If there are repeated values (or rows) in A, then ia contains the index to the first occurrence of the repeated value. For example: `[C,ia,ic] = unique([9 9 9],'first','legacy')` returns `ia = 1`.

Output Arguments

`C` — Unique data of `A`

vector | matrix | table

Unique data of A, returned as a vector, matrix, or table. The following describes the shape of C when the input, A, is a vector or a matrix:

- If the `'rows'` flag is not specified and A is a row vector, then C is a row vector.
- If the `'rows'` flag is not specified and A is not a row vector, then C is a column vector.
- If the `'rows'` flag is specified, then C is a matrix containing the unique rows of A.

The class of C is the same as the class of the input A.

`ia` — Index to `A`

column vector

Index to A, returned as a column vector of indices to the *first* occurrence of repeated elements. When the `'legacy'` flag is specified, ia is a row vector that contains indices to the *last* occurrence of repeated elements.

The indices generally satisfy `C = A(ia)`. If `A` is a table, or if the `'rows'` option is specified, then `C = A(ia,:)`.

`ic` — Index to `C`

column vector

Index to C, returned as a column vector when the `'legacy'` flag is not specified. ic contains indices that satisfy the following properties.

- If `A` is a vector, then `A = C(ic)`.
- If `A` is a matrix or array, then `A(:) = C(ic)`.
- If `A` is a table, or if the `'rows'` option is specified, then `A = C(ic,:)`.

More About

Tips

- To find unique rows in table `A` with respect to a subset of variables, you can use column subscripting. For example, you can use `unique(A(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.

See Also

`intersect` | `ismember` | `issorted` | `setdiff` | `setxor` | `sort` | `union` | `uniquetol`

Introduced before R2006a

uniquetol

Set unique within a tolerance

`uniquetol` is similar to `unique`. Whereas `unique` performs exact comparisons, `uniquetol` performs comparisons using a tolerance.

Syntax

```
C = uniquetol(A,tol)
C = uniquetol(A)
[C,IA,IC] = uniquetol( ___ )
[ ___ ] = uniquetol( ___ ,Name,Value)
```

Description

`C = uniquetol(A,tol)` returns the unique elements in `A` using tolerance `tol`. Two values, `u` and `v`, are within tolerance if

```
abs(u-v) <= tol*max(abs(A(:)))
```

That is, `uniquetol` scales the `tol` input based on the magnitude of the data.

`C = uniquetol(A)` uses a default tolerance of `1e-6` for single-precision inputs and `1e-12` for double-precision inputs.

`[C,IA,IC] = uniquetol( ___ )` returns index vectors `IA` and `IC`, such that `C = A(IA)` and `A~C(IC)` (or `A(:)~C(IC)` if `A` is a matrix), where ~ means the values are within tolerance of each other. You can use any of the input arguments in previous syntaxes.

`[ ___ ] = uniquetol( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments using any of the input or output argument combinations in previous syntaxes. For example, `uniquetol(A,'ByRows',true)` determines the unique rows in `A`.

Examples

Unique Elements in Presence of Numerical Error

Create a vector, `x`. Obtain a second vector, `y`, by transforming and untransforming `x`. This transformation introduces round-off differences to `y`.

```
x = (1:6)'*pi;
y = 10.^log10(x);
```

Verify that `x` and `y` are not identical by taking the difference.

```
x-y
```

```
ans =

   1.0e-14 *

    0.0444
         0
         0
         0
         0
   -0.3553
```

Use `unique` to find the unique elements in the concatenated vector `[x;y]`. The `unique` function performs exact comparisons and determines that some values in `x` are not exactly equal to values in `y`. These are the same elements that have a nonzero difference in `x-y`. Thus, `c` contains values that *appear* to be duplicates.

```
c = unique([x;y])
```

```
c =

    3.1416
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
```

```
   18.8496
   18.8496
```

Use `uniquetol` to perform the comparison using a small tolerance. `uniquetol` treats elements that are within tolerance as equal.

```
C = uniquetol([x;y])
```

```
C =

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
```

Determine Unique Rows

By default, `uniquetol` looks for unique *elements* that are within tolerance, but it also can find unique *rows* of a matrix that are within tolerance.

Create a numeric matrix, `A`. Obtain a second matrix, `B`, by transforming and untransforming `A`. This transformation introduces round-off differences to `B`.

```
A = [0.05 0.11 0.18; 0.18 0.21 0.29; 0.34 0.36 0.41; 0.46 0.52 0.76];
B = log10(10.^A);
```

Use `unique` to find the unique rows in `A` and `B`. The `unique` function performs exact comparisons and determines that all of the rows in the concatenated matrix `[A;B]` are unique, even though some of the rows differ by only a small amount.

```
unique([A;B],'rows')
```

```
ans =

    0.0500    0.1100    0.1800
    0.0500    0.1100    0.1800
    0.1800    0.2100    0.2900
```

```
    0.1800    0.2100    0.2900
    0.3400    0.3600    0.4100
    0.3400    0.3600    0.4100
    0.4600    0.5200    0.7600
    0.4600    0.5200    0.7600
```

Use `uniquetol` to find the unique rows. `uniquetol` treats rows that are within tolerance as equal.

```
uniquetol([A;B],'ByRows',true)
```

```
ans =

    0.0500    0.1100    0.1800
    0.1800    0.2100    0.2900
    0.3400    0.3600    0.4100
    0.4600    0.5200    0.7600
```

Prepare Vectors for Exact Comparison

Create a vector, `x`. Obtain a second vector, `y`, by transforming and untransforming `x`. This transformation introduces round-off differences to some elements in `y`.

```
x = (1:6)'*pi;
y = 10.^log10(x);
```

Combine `x` and `y` into a single tall vector, `A`. Use `uniquetol` to reconstruct `A`, treating the values that are within tolerance as equal.

```
A = [x;y]
[C,IA,IC] = uniquetol(A);
newA = C(IC)
```

```
A =

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
```

```
   18.8496
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496


newA =

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
```

You can use `newA` with `==` or functions that use exact equality like `isequal` or `unique` in subsequent code.

```
D1 = unique(A)
D2 = unique(newA)


D1 =

    3.1416
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
   18.8496


D2 =
```

```
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
```

Subset Data Using Large Tolerance

Create a cloud of 2-D sample points constrained to be inside a circle of radius `0.5` centered at the point $(\frac{1}{2}, \frac{1}{2})$.

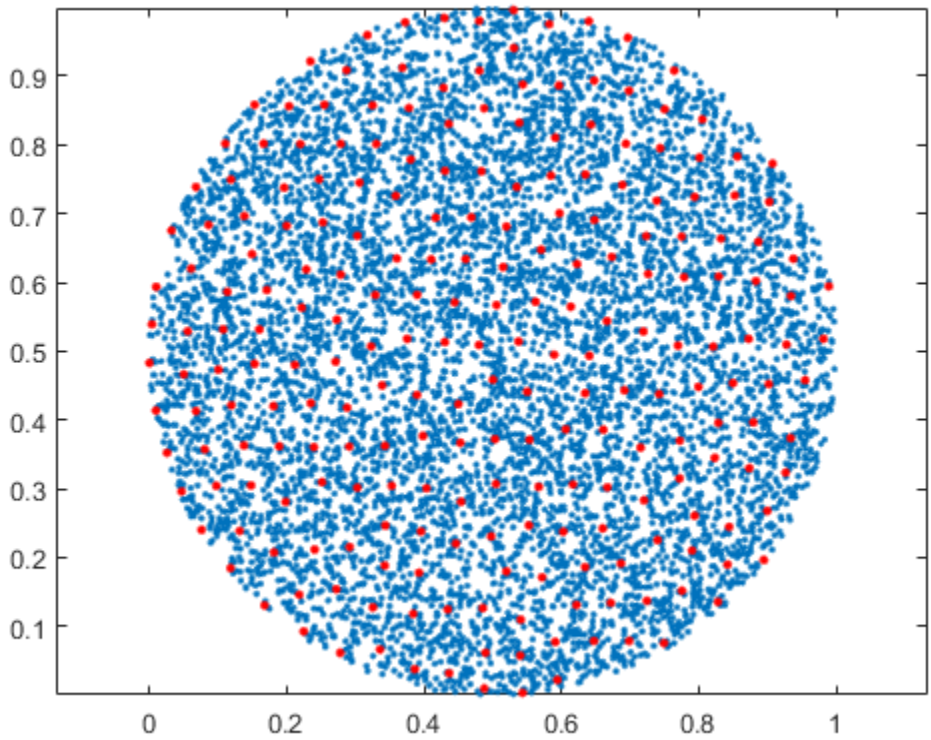
```
x = rand(10000,2);
insideCircle = sqrt((x(:,1)-.5).^2+(x(:,2)-.5).^2)<0.5;
y = x(insideCircle,:);
```

Find a reduced set of points, such that each point of the original dataset is within tolerance of a point.

```
tol = 0.05;
C = uniquetol(y,tol,'ByRows',true);
```

Plot the reduced set of points as red dots on top of the original data set. The red dots are all members of the original data set. All the red dots are at least a distance `tol` apart.

```
plot(y(:,1),y(:,2),'.')
hold on
axis equal
plot(C(:,1), C(:,2), '.r', 'MarkerSize', 10)
```



Average Similar Values in Vector

Create a vector of random numbers and determine the unique elements using a tolerance. Specify `OutputAllIndices` as `true` to return all of the indices for the elements that are within tolerance of the unique values.

```
A = rand(100,1);
[C,IA] = uniquetol(A,1e-2,'OutputAllIndices',true);
```

Find the average value of the elements that are within tolerance of the value `C(2)`.

```
C(2)
allA = A(IA{2})
aveA = mean(allA)
```

```
ans =

    0.0318


allA =

    0.0357
    0.0318
    0.0344


aveA =

    0.0340
```

Specify Absolute Tolerance

By default, `uniquetol` uses a tolerance test of the form `abs(u-v) <= tol*DS`, where `DS` automatically *scales* based on the magnitude of the input data. You can specify a different `DS` value to use with the `DataScale` option. However, absolute tolerances (where `DS` is a scalar) do not scale based on the magnitude of the input data.

First, compare two small values that are a distance `eps` apart. Specify `tol` and `DS` to make the within tolerance equation: `abs(u-v) <= 10^-6`.

```
x = 0.1;
uniquetol([x, exp(log(x))], 10^-6, 'DataScale', 1)
```

```
ans =

    0.1000
```

Next, increase the magnitude of the values. The round-off error in the calculation `exp(log(x))` is proportional to the magnitude of the values, specifically to `eps(x)`. Even though the two large values are a distance `eps` from one another, `eps(x)` is now much larger. Therefore, `10^-6` is no longer a suitable tolerance.

```
x = 10^10;
uniquetol([x, exp(log(x))], 10^-6, 'DataScale', 1)
```

```
ans =

   1.0e+10 *

    1.0000    1.0000
```

Correct this issue by using the default (scaled) value of `DS`.

```
format long
Y = [0.1 10^10];
uniquetol([Y, exp(log(Y))])
```

```
ans =

   1.0e+10 *

   0.000000000010000   1.000000000000000
```

Specify DataScale by Column

Create a set of random 2-D points, then use `uniquetol` to group the points into vertical bands that have a similar (within tolerance) x-coordinate. Use these options with `uniquetol`:

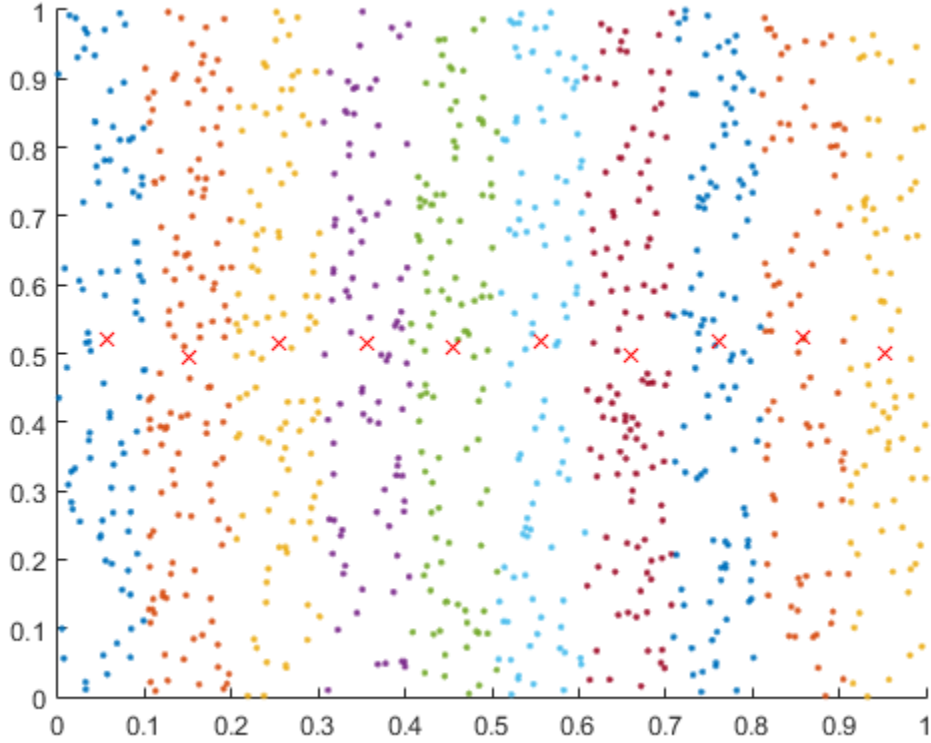
- Specify `ByRows` as `true` since the point coordinates are in the rows of `A`.
- Specify `OutputAllIndices` as `true` to return the indices for all points that have an x-coordinate within tolerance of each other.
- Specify `DataScale` as `[1 Inf]` to use an absolute tolerance for the `x`-coordinate while ignoring the `y`-coordinate.

```
A = rand(1000,2);
DS = [1 Inf];
[C,IA] = uniquetol(A, 0.1, 'ByRows', true, ...
    'OutputAllIndices', true, 'DataScale', DS);
```

Plot the points and average value for each band.

```
hold on
for k = 1:length(IA)
    plot(A(IA{k},1), A(IA{k},2), '.')
```

```
    meanAi = mean(A(IA{k},:));
    plot(meanAi(1), meanAi(2), 'xr')
end
```



- “Average Similar Data Points Using a Tolerance”

Input Arguments

A — Query array

scalar | vector | matrix

Query array, specified as a scalar, vector, or matrix. A must be full.

Data Types: `single | double`

`tol` — Comparison tolerance

positive, real scalar

Comparison tolerance, specified as a positive, real scalar. `uniquetol` scales the `tol` input using the maximum absolute value in input array `A`. Then `uniquetol` uses the resulting scaled comparison tolerance to determine which elements in `A` are unique. If two elements in `A` are within tolerance of each other, then `uniquetol` considers them to be equal.

Two values, `u` and `v`, are within tolerance if `abs(u-v) <= tol*max(abs(A))`.

To specify an absolute tolerance, specify both `tol` and the `'DataScale'` Name-Value pair.

Example: `tol = 0.05`

Example: `tol = 1e-8`

Example: `tol = eps`

Data Types: `single | double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `C = uniquetol(A,'ByRows',true)`

`'OutputAllIndices'` — Output index type

`false` (default) | `true` | `0` | `1`

Output index type, specified as the comma-separated pair consisting of `'OutputAllIndices'` and either `false` (default), `true`, `0`, or `1`. `uniquetol` interprets numeric `0` as `false` and numeric `1` as `true`.

When `OutputAllIndices` is `true`, the `uniquetol` function returns the second output, `IA`, as a cell array. The cell array contains the indices for *all* elements in `A` that are within tolerance of a value in `C`. That is, each cell in `IA` corresponds to a value in `C`, and the values in each cell correspond to locations in `A`.

Example: `[C,IA] = uniquetol(A,tol,'OutputAllIndices',true)`

`'ByRows'` — Row comparison toggle
`false` (default) | `true` | `0` | `1`

Row comparison toggle, specified as the comma-separated pair consisting of `'ByRows'` and either `false` (default), `true`, `0`, or `1`. `uniquetol` interprets numeric `0` as `false` and numeric `1` as `true`. Use this option to find rows in `A` that are unique, within tolerance.

When `ByRows` is `true`:

- `uniquetol` compares the rows of `A` by considering each column separately. For two rows to be within tolerance of one another, each column has to be in tolerance.
- Each row in `A` is within tolerance of a row in `C`. However, no two rows in `C` are within tolerance of each other.

Two rows, `u` and `v`, are within tolerance if `abs(u-v) <= tol*max(abs(A),[],1)`.

Example: `C = uniquetol(A,tol,'ByRows',true)`

`'DataScale'` — Scale of data
scalar | vector

Scale of data, specified as the comma-separated pair consisting of `'DataScale'` and either a scalar or vector. Specify `DataScale` as a numeric scalar, `DS`, to change the tolerance test to be `abs(u-v) <= tol*DS`.

When used together with the `ByRows` option, the `DataScale` value also can be a vector. In this case, each element of the vector specifies `DS` for a corresponding column in `A`. If a value in the `DataScale` vector is `Inf`, then `uniquetol` ignores the corresponding column in `A`.

Example: `C = uniquetol(A,'DataScale',1)`

Example: `[C,IA,IC] = uniquetol(A,'ByRows',true,'DataScale',[eps(1) eps(10) eps(100)])`

Data Types: `single` | `double`

Output Arguments

`C` — Unique elements in `A`
vector | matrix

Unique elements in `A` (within tolerance), returned as a vector or matrix. If `A` is a row vector, then `C` is also a row vector. Otherwise, `C` is a column vector. The elements in `C` are sorted in ascending order. Each element in `A` is within tolerance of an element in `C`, but no two elements in `C` are within tolerance of each other.

If the `ByRows` option is `true`, then `C` is a matrix containing the unique rows in `A`. In this case, the rows in `C` are sorted in ascending order by the first column. Each row in `A` is within tolerance of a row in `C`, but no two rows in `C` are within tolerance of each other.

`IA` — Index to `A`

column vector | cell array

Index to `A`, returned as a column vector of indices to the first occurrence of repeated elements, or as a cell array. `IA` generally satisfies `C = A(IA)`, with the following exceptions:

- If the `ByRows` option is `true`, then `C = A(IA,:)`.
- If the `OutputAllIndices` option is `true`, then `IA` is a cell array and `C(i)~A(IA{i})` where ~ means the values are within tolerance of each other.

`IC` — Index to `C`

column vector

Index to `C`, returned as a column vector of indices. `IC` satisfies the following properties, where ~ means the values are within tolerance of each other.

- If `A` is a vector, then `A~C(IC)`.
- If `A` is a matrix, then `A(:)~C(IC)`.
- If the `ByRows` option is `true`, then `A~C(IC,:)`.

More About

Tips

- There can be multiple valid `C` outputs that satisfy the condition, *no two elements in `C` are within tolerance of each other*. The `uniquetol` function just returns one of the valid outputs.

  `uniquetol` sorts the input lexicographically, and then starts at the lowest value to find unique values within tolerance. As a result, changing the sorting of the input

could change the output. For example, `uniquetol(-A)` might not give the same results as `-uniquetol(A)`.

See Also

`eps` | `ismember` | `ismembertol` | `unique`

Introduced in R2015a

matlab.unittest

Summary of packages and classes in MATLAB Unit Test Framework

Description

The `matlab.unittest` package consists of the following classes and packages:

matlab.unittest.constraints	Summary of classes in MATLAB Constraints Interface
matlab.unittest.diagnostics	Summary of classes in MATLAB Diagnostics Interface
matlab.unittest.fixtures	Summary of classes in MATLAB Fixtures Interface
matlab.unittest.FunctionTestCase	TestCase used for function-based tests
matlab.unittest.parameters	Summary of classes associated with MATLAB Unit Test parameters
matlab.unittest.plugins	Summary of classes in MATLAB Plugins Interface
matlab.unittest.plugins.plugindata	Summary of classes in MATLAB Plugin Data Interface
matlab.unittest.qualifications	Summary of classes in MATLAB Qualifications Interface
matlab.unittest.selectors	Summary of classes in MATLAB Selectors Interface
matlab.unittest.Test	Specification of a single test method

matlab.unittest.TestCase	Superclass of all `matlab.unittest` test classes
matlab.unittest.TestResult	Result of running test suite
matlab.unittest.TestRunner	Class for running tests in `matlab.unittest` framework
matlab.unittest.TestSuite	Class for grouping tests to run
matlab.unittest.Verbosity	Verbosity level enumeration class

unix

Execute UNIX command and return output

Syntax

```
status = unix(command)
[status,cmdout] = unix(command)
[status,cmdout] = unix(command,'-echo')
```

Description

`status = unix(command)` calls the UNIX operating system to execute the specified command. The operation waits for the command to finish execution before returning the exit status of the command to the `status` variable.

`[status,cmdout] = unix(command)` additionally returns the standard output of the command to `cmdout`. This syntax is most useful for commands that do not require user input.

`[status,cmdout] = unix(command,'-echo')` additionally displays (echoes) the command output in the MATLAB Command Window. This syntax is most useful for commands that require user input and that run correctly in the MATLAB Command Window.

Examples

Save UNIX Command Exit Status and Output

List all users who are currently logged in, and save the command exit status and output. Then, view the status.

```
command = 'who';
[status,cmdout] = unix(command);
status
```

```
status =
```

```
     0
```

A `status` of zero indicates that the command completed successfully. MATLAB returns a string containing the list of users in `cmdout`.

Input Arguments

`command` — UNIX command

string

UNIX command, specified as a string. The `command` executes in a UNIX shell, which might not be the shell from which you launched MATLAB.

Example: `'ls'`

Output Arguments

`status` — Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (`&`), then `status` is the exit status upon command launch.
- If `command` does not include the ampersand character (`&`), then `status` is the exit status upon command completion.

`cmdout` — Output of operating system command

string

Output of the operating system command, returned as a string.

More About

Tips

- To execute the operating system command in the background, include the trailing character, `&`, in the `command` argument (for example, `'emacs &'`). The exit status

is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

Note: If `command` includes the trailing `&` character, `cmdout` is empty.

- The `unix` function redirects `stdin` to the invoked command, `command`, by default. This redirection also forwards MATLAB script commands and the keyboard type-ahead buffer to the invoked command while the `unix` function executes. This can lead to corrupted output when `unix` does not complete execution immediately. To disable `stdin` and type-ahead redirection, include the formatted string `< /dev/null` in the call to the invoked command.

Algorithms

MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.

- "Run External Commands, Scripts, and Programs"

See Also

`!` (exclamation point) | `computer` | `dos` | `perl` | `system`

Introduced before R2006a

unloadlibrary

Unload shared library from memory

Syntax

```
unloadlibrary libname
```

Description

`unloadlibrary libname` unloads library `libname` from memory.

Examples

Unload Library

Add path to examples folder.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Load the library, if it is not already loaded.

```
if ~libisloaded('shrlibsample')
   loadlibrary('shrlibsample')
end
```

Clean up.

```
unloadlibrary shrlibsample
```

Input Arguments

`libname` — Name of shared library

string

Name of shared library, specified as a string. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

More About

Tips

- To unload a MEX-file, use the `clear` function.

See Also

`clear` | `libisloaded` | `loadlibrary`

Introduced before R2006a

unmesh

Convert edge matrix to coordinate and Laplacian matrices

Syntax

```
[L,XY] = unmesh(E)
```

Description

`[L,XY] = unmesh(E)` returns the Laplacian matrix `L` and mesh vertex coordinate matrix `XY` for the `M`-by-4 edge matrix `E`. Each row of the edge matrix must contain the coordinates `[x1 y1 x2 y2]` of the edge endpoints.

Input Arguments

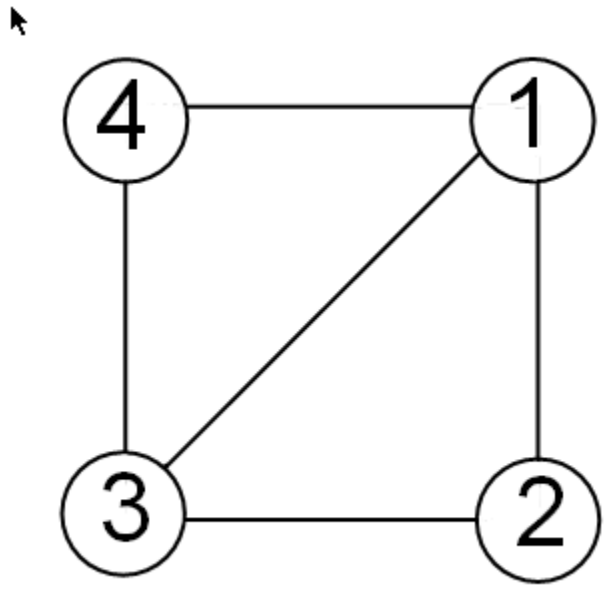
`E`	M-by-4 edge matrix `E`.

Output Arguments

`L`	Laplacian matrix representation of the graph.
`XY`	Mesh vertex coordinate matrix.

Examples

Take a simple example of a square with vertices at (1,1), (1,–1),(–1,–1), and (–1,1), where the connections between vertices are the four perpendicular edges of the square plus one diagonal connection between (–1, –1) and (1,1).



The edge matrix `E` for this graph is:

```
E=[1 1 1 -1;  % edge from 1 to 2
1 -1 -1 -1;   % edge from 2 to 3
-1 -1 -1 1;   % edge from 3 to 4
-1 -1 1 1;    % edge from 4 to 1
-1 1 1 1]     % edge from 3 to 1
```

Use `unmesh` to create the output matrices,

```
[A,XY]=unmesh(E);
4 vertices:
4/4
```

The Laplacian matrix is defined as

$$L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

`unmesh` returns the Laplacian matrix `L` in sparse notation.

```
L
```

```
L =

   (1,1)         3
   (2,1)        -1
   (3,1)        -1
   (4,1)        -1
   (1,2)        -1
   (2,2)         2
   (4,2)        -1
   (1,3)        -1
   (3,3)         2
   (4,3)        -1
   (1,4)        -1
   (2,4)        -1
   (3,4)        -1
```

To see `L` in regular matrix notation, use the `full` command.

```
full(L)
```

```
ans =

     3    -1    -1    -1
    -1     2     0    -1
    -1     0     2    -1
    -1    -1    -1     3
```

The mesh coordinate matrix `XY` returns the coordinates of the corners of the square.

```
XY
```

```
XY =

    -1    -1
    -1     1
     1    -1
     1     1
```

See Also

`gplot` | `treeplot`

unmkpp

Piecewise polynomial details

Syntax

```
[breaks,coefs,l,k,d] = unmkpp(pp)
```

Description

`[breaks,coefs,l,k,d] = unmkpp(pp)` extracts, from the piecewise polynomial `pp`, its breaks `breaks`, coefficients `coefs`, number of pieces `l`, order `k`, and dimension `d` of its target. Create `pp` using `spline` or the spline utility `mkpp`.

Examples

This example creates a description of the quadratic polynomial

$$\frac{-x^2}{4} + x$$

as a piecewise polynomial `pp`, then extracts the details of that description.

```
pp = mkpp([-8 -4],[-1/4 1 0]);
[breaks,coefs,l,k,d] = unmkpp(pp)

breaks =
    -8    -4

coefs =
   -0.2500    1.0000         0

l =
     1

k =
     3
```

```
d =
     1
```

See Also

`mkpp` | `ppval` | `spline`

Introduced before R2006a

unregisterallevents

Unregister all event handlers associated with COM object events at run time

Syntax

```
unregisterallevents(h)
```

Description

`unregisterallevents(h)` unregisters all events previously registered with COM object `h`. After calling `unregisterallevents`, the object no longer responds to any events until you register them again using the `registerevent` function.

COM functions are available on Microsoft Windows systems only.

Examples

Register and unregister events for an instance of the `mwsamp` control, using the `eventlisteners` function to see the event handler associated with each event:

1. Register three events and their respective handler routines.

```
f = figure('position',[100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ...
    [0 0 200 200],f, ...
    {'Click' 'myclick'; 'DblClick' 'my2click'; ...
    'MouseDown' 'mymoused'});
eventlisteners(h)

ans =
    'click'         'myclick'
    'dblclick'      'my2click'
    'mousedown'     'mymoused'
```

2. Unregister all events simultaneously with `unregisterallevents`. `eventlisteners` returns an empty cell array, indicating that there are no longer any events registered with the control:

```
unregisterallevents(h);
eventlisteners(h)

ans =
     {}
```

See Also

events (COM) | eventlisteners | registerevent | unregisterevent | isevent

Introduced before R2006a

unregisterevent

Unregister event handler associated with COM object event at run time

Syntax

```
unregisterevent(h,eventhandler)
```

Description

`unregisterevent(h,eventhandler)` unregisters specific event handler routines from their corresponding events. Once you unregister an event, the object no longer responds to the event.

You can unregister events at any time after creating a control. The `eventhandler` argument, which is a cell array, specifies both events and event handlers.

```
unregisterevent(h,{'event_name',@event_handler});
```

Specify events in the `eventhandler` argument using the names of the events. Strings used in the `eventhandler` argument are not case-sensitive. `unregisterevent` does not accept numeric event identifiers.

COM functions are available on Microsoft Windows systems only.

Examples

Unregister events for a control:

1. Create an `mwsamp` control and register all events with the same handler routine, `sampev`. Use `eventlisteners` to see the event handler used by each event. In this case, each event, when fired, calls `sampev.m`:

   ```
   f = figure('position',[100 200 200 200]);
   h = actxcontrol('mwsamp.mwsampctrl.2',...
       [0 0 200 200],f,'sampev');
   eventlisteners(h)
   ```

```
ans =
    'Click'          'sampev'
    'DblClick'       'sampev'
    'MouseDown'      'sampev'
    'Event_Args'     'sampev'
```

2. Unregister just the `dblclick` event. Now, when you list the registered events using `eventlisteners`, `dblclick` is no longer registered and the control does not respond when you double-click the mouse over it:

```
unregisterevent(h,{'dblclick' 'sampev'});
eventlisteners(h)
```

```
ans =
    'Click'          'sampev'
    'MouseDown'      'sampev'
    'Event_Args'     'sampev'
```

3. Now, register the `click` and `dblclick` events with a different event handler for `myclick` and `my2click`, respectively:

```
unregisterallevents(h);
registerevent(h,{'click' 'myclick';'dblclick' 'my2click'});
eventlisteners(h)
```

```
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
```

4. Unregister these same events by specifying event names and their handler routines in a cell array. `eventlisteners` now returns an empty cell array, meaning that no events are registered for the `mwsamp` control:

```
unregisterevent(h,{'click' 'myclick'; 'dblclick' 'my2click'});
eventlisteners(h)
```

```
ans =
     {}
```

Unregister Microsoft Excel workbook events:

1. Create a `Workbook` object and register two events with the event handler routines, `EvtActivateHndlr` and `EvtDeactivateHndlr`:

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
```

```
wb = Add(wbs);
registerevent(wb,{'Activate' 'EvtActivateHndlr';...
    'Deactivate' 'EvtDeactivateHndlr'})
eventlisteners(wb)

ans =
    'Activate'      'EvtActivateHndlr'
    'Deactivate'    'EvtDeactivateHndlr'
```

MATLAB shows the events with the corresponding event handlers.

2 Next, unregister the `Deactivate` event handler:

```
unregisterevent(wb,{'Deactivate' 'EvtDeactivateHndlr'})
eventlisteners(wb)

ans =
    'Activate'    'EvtActivateHndlr'
```

MATLAB shows the remaining registered event (`Activate`) with its corresponding event handler.

More About

- "Writing Event Handlers"

See Also

`events (COM)` | `eventlisteners` | `registerevent` | `unregisterallevents` | `isevent`

Introduced before R2006a

unstack

Unstack data from single variable into multiple variables

Syntax

```
W = unstack(T,vars,ivar)
W = unstack(T,vars,ivar,Name,Value)
[W,it] = unstack( ___ )
```

Description

`W = unstack(T,vars,ivar)` converts the tall table, `T`, to an equivalent table, `W`, that is in wide format. `unstack` unstacks a single variable in `T`, specified by `vars`, into multiple variables in `W`. In general, `W` contains more variables, but fewer rows, than `T`.

An indicator variable in `T`, specified by `ivar`, determines which variable in `W` contains each value in `var` after it is unstacked. `unstack` treats the remaining variables in `T` as grouping variables. Each unique combination of values in the grouping variables identifies a group of rows in `T` that will be unstacked into a single row of `W`.

`W = unstack(T,vars,ivar,Name,Value)` converts the table `T` to wide format with additional options specified by one or more Name,Value pair arguments.

For example, you can specify how `unstack` converts variables from `T` to variables in `W`.

`[W,it] = unstack( ___ )` also returns an index vector, it, indicating the correspondence between rows in `W` and rows in `T`. You can use any of the previous input arguments.

Examples

Separate One Variable into Three Variables

Create a table indicating the amount of snowfall in various towns for various storms.

```
Storm = [3;3;1;3;1;1;4;2;4;2;4;2];
Town = {'T1';'T3';'T1';'T2';'T2';'T3';...
    'T2';'T1';'T3';'T3';'T1';'T2'};
Snowfall = [0;3;5;5;9;10;12;13;15;16;17;21];

T = table(Storm,Town,Snowfall)

T =

    Storm    Town    Snowfall
    _____    ____    ________

    3        'T1'     0
    3        'T3'     3
    1        'T1'     5
    3        'T2'     5
    1        'T2'     9
    1        'T3'    10
    4        'T2'    12
    2        'T1'    13
    4        'T3'    15
    2        'T3'    16
    4        'T1'    17
    2        'T2'    21
```

`T` contains three snowfall entries for each storm, one for each town.

Separate the variable `Snowfall` into three variables, one for each town specified in the variable, `Town`.

```
W = unstack(T,'Snowfall','Town')

W =

    Storm    T1    T2    T3
    _____    __    __    __

    3         0     5     3
    1         5     9    10
    4        17    12    15
    2        13    21    16
```

Each row in W contains data from rows in T that have the same value in the grouping variable, Storm. The order of the unique values in Storm determines the order of the data in W.

Apply Aggregation Function to Each Group

Unstack data and apply an aggregation function to multiple rows in the same group that have the same values in the indicator variable.

Create a table containing data on the price of two stocks over 2 days.

```
Date = [repmat({'4/12/2008'},6,1);...
    repmat({'4/13/2008'},5,1)];
Stock = {'Stock1';'Stock2';'Stock1';'Stock2';...
    'Stock2';'Stock2';'Stock1';'Stock2';...
    'Stock2';'Stock1';'Stock2'};
Price = [60.35;27.68;64.19;25.47;28.11;27.98;...
    63.85;27.55;26.43;65.73;25.94];

T = table(Date,Stock,Price)
```

```
T =

       Date          Stock      Price
    ____________    ________    _____

    '4/12/2008'     'Stock1'    60.35
    '4/12/2008'     'Stock2'    27.68
    '4/12/2008'     'Stock1'    64.19
    '4/12/2008'     'Stock2'    25.47
    '4/12/2008'     'Stock2'    28.11
    '4/12/2008'     'Stock2'    27.98
    '4/13/2008'     'Stock1'    63.85
    '4/13/2008'     'Stock2'    27.55
    '4/13/2008'     'Stock2'    26.43
    '4/13/2008'     'Stock1'    65.73
    '4/13/2008'     'Stock2'    25.94
```

T contains two prices for STOCK1 during the first day and four prices for STOCK2 during the first day.

Create a table containing separate variables for each stock and one row for each day. Use Date as the grouping variable and apply the aggregation function, @mean, to the numeric values from the variable, Price, for each group.

```
[W,it] = unstack(T,'Price','Stock',...
    'AggregationFunction',@mean)
```

```
W =

       Date         Stock1    Stock2
    ____________    ______    ______

    '4/12/2008'     62.27     27.31
    '4/13/2008'     64.79     26.64


it =

     1
     7
```

`W` contains the average price for each stock grouped by date.

`it` identifies the index of the first value for each group of rows in `T`. The first value for the group `'4/13/2008'` is in the seventh row of `T`.

Input Arguments

`T` — Tall table

table

Tall table, specified as a table. `T` must contain data variables to unstack, `vars`, and an indicator variable, `ivars`. The remaining variables in `T` are either grouping variables or constant variables.

`vars` — Variables in `T` to unstack

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables in `T` to unstack, specified as a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

`ivar` — Indicator variable in `T`

positive integer | variable name

Indicator variable in `T`, specified as a positive integer or a variable name. The variable specified by `ivar` indicates which variable in `W` each value in `var` is unstacked into.

`ivar` can be a numeric vector, logical vector, character array, cell array of strings, or categorical vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AggregationFunction',@mean` applies the aggregation function `@mean` to the values in `vars`.

`'GroupingVariables'` — Grouping variables in `T` that define groups of rows
positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Grouping variables in `T` that define groups of rows, specified as the comma-separated pair consisting of `'GroupingVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector. Each group of rows in `T` becomes one row in `W`.

The default is all the variables in `T` not listed in vars or ivar.

`'ConstantVariables'` — Variables constant within a group
positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

Variables constant within a group, specified as the comma-separated pair consisting of `'ConstantVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector. The default is no variables.

The values for these variables in `W` are taken from the first row in each group in `T`.

`'NewDataVariableNames'` — Names for new data variables in `W`
cell array of strings

Names for new data variables in `W`, specified as the comma-separated pair consisting of `'NewDataVariableNames'` and a cell array of strings. The default is strings based on the values of the grouping variable specified in ivar.

`'AggregationFunction'` — Aggregation function from values in `vars` to single value
function handle

Aggregation function from values in `vars` to single value, specified as the comma-separated pair consisting of `'AggregationFunction'` and a function handle. `unstack` applies this function to rows from the same group that have the same value in ivar. The function must aggregate the data values into a single value.

For a numeric data variable, the default is `@sum`. For nonnumeric variables, there is no default function, and you must specify the `'AggregationFunction'` name-value pair argument if multiple rows in the same group have the same value in ivar.

Output Arguments

`W` — Wide table

table

Wide table, returned as a table. `W` contains the unstacked data variables, the grouping variables, and the first value of each group from any constant variables.

The order of the data in `W` is based on the order of the unique values in the grouping variables.

You can store additional metadata such as descriptions, variable units, variable names, and row names in the table. For more information, see `Table Properties`.

`it` — Index to `T`

column vector

Index to `T`, returned as a column vector. For each row in `W`, the index vector, `it`, identifies the index of the first value in the corresponding group of rows in `T`.

More About

Grouping Variables

Grouping variables are utility variables used to group, or categorize, data. Grouping variables are useful for summarizing or visualizing data by group. You can define groups in your table by specifying one or more grouping variables.

A grouping variable can be any of the following:

- Categorical vector
- Cell array of strings
- Character array
- Numeric vector, typically containing positive integers
- Logical vector

Rows that have the same grouping variable value belong to the same group. If you use multiple grouping variables, rows that have the same combination of grouping variable values belong to the same group.

Tips

- You can specify more than one data variable in `T`, and each variable becomes a set of unstacked data variables in `W`. Use a vector of positive integers, a cell array containing multiple variable names, or a logical vector to specify `vars`. The one indicator variable, specified by the input argument, `ivar`, applies to all data variables specifies by `vars`.

See Also

`join` | `stack`

untar

Extract contents of tar file

Syntax

```
untar(tarfilename)
untar(tarfilename,outputdir)
untar(url, ___ )
filenames = untar( ___ )
```

Description

`untar(tarfilename)` extracts the archived contents of `tarfilename` into the current directory and sets the files' attributes. It overwrites any existing files with the same names as those in the archive if the existing files' attributes and ownerships permit it. For example, if you rerun `untar` on the same `tarfilename`, MATLAB software does not overwrite files with a read-only attribute; instead, `untar` displays a warning for such files. On Microsoft Windows platforms, the hidden, system, and archive attributes are not set.

`tarfilename` is a string specifying the name of the tar file. `tarfilename` is gunzipped to a temporary directory and deleted if its extension ends in `.tgz` or `.gz`. If an extension is omitted, `untar` searches for `tarfilename` appended with `.tgz`, `.tar.gz`, or `.tar`. `tarfilename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

`untar(tarfilename,outputdir)` uncompresses the archive `tarfilename` into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`untar(url, ___ )` extracts the tar archive from an Internet URL. The URL must include the protocol type (for example, `'http://'` or `'ftp://'`). MATLAB downloads the URL to a temporary directory, and then deletes it.

`filenames = untar( ___ )` extracts the tar archive and returns the names of the extracted files in the string cell array `filenames`. If `outputdir` specifies a relative path, `filenames` contains the relative path. If `outputdir` specifies an absolute path, `filenames` contains the absolute path.

Examples

Using tar and untar to Copy Files

Copy all `.m` files in the current directory to the directory `backup`.

```
tar('mymfiles.tar.gz','*.m');
untar('mymfiles','backup');
```

Using untar with URL

Run `untar` to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`.

```
url ='http://www.mathworks.com/moler/ncm.tar.gz';
ncmFiles = untar(url,'ncm')
```

See Also

`gzip` | `gunzip` | `tar` | `unzip` | `zip`

Introduced before R2006a

unwrap

Correct phase angles to produce smoother phase plots

Syntax

```
Q = unwrap(P)
Q = unwrap(P,tol)
Q = unwrap(P,[],dim)
Q = unwrap(P,tol,dim)
```

Description

`Q = unwrap(P)` corrects the radian phase angles in a vector `P` by adding multiples of $\pm 2\pi$ when absolute jumps between consecutive elements of `P` are greater than or equal to the default jump tolerance of π radians. If `P` is a matrix, `unwrap` operates columnwise. If `P` is a multidimensional array, `unwrap` operates on the first nonsingleton dimension.

`Q = unwrap(P,tol)` uses a jump tolerance `tol` instead of the default value, π.

`Q = unwrap(P,[],dim)` unwraps along `dim` using the default tolerance.

`Q = unwrap(P,tol,dim)` uses a jump tolerance of `tol`.

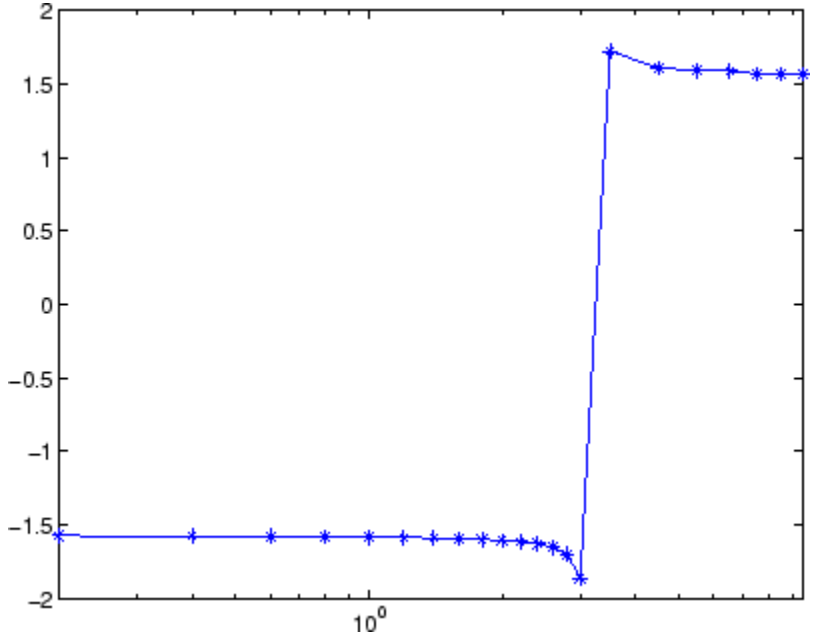
Note A jump tolerance less than π has the same effect as a tolerance of π. For a tolerance less than π, if a jump is greater than the tolerance but less than π, adding $\pm 2\pi$ would result in a jump larger than the existing one, so `unwrap` chooses the current point. If you want to eliminate jumps that are less than π, try using a finer grid in the domain.

Examples

Example 1

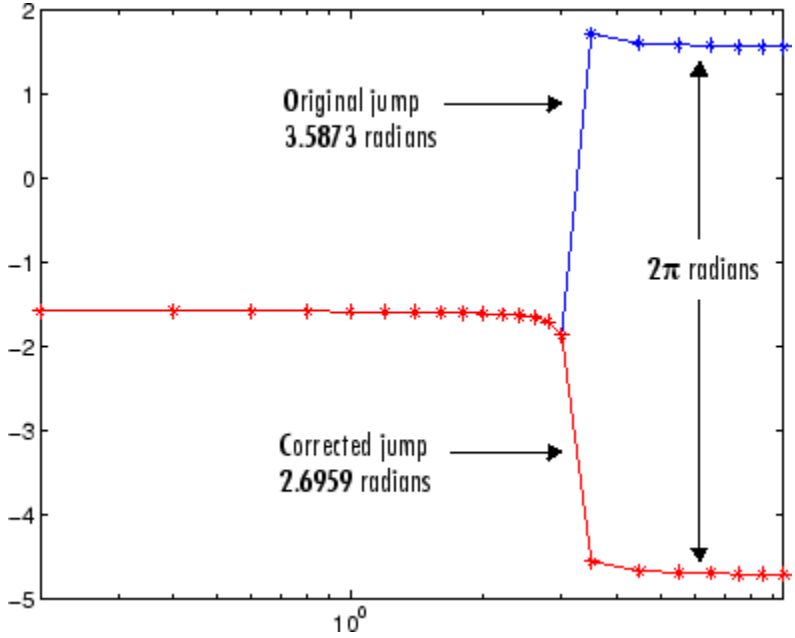
The following phase data comes from the frequency response of a third-order transfer function. The phase curve jumps `3.5873` radians between `w = 3.0` and `w = 3.5`, from `-1.8621` to `1.7252`.

```
w = [0:.2:3,3.5:1:10];
p = [    0
     -1.5728
     -1.5747
     -1.5772
     -1.5790
     -1.5816
     -1.5852
     -1.5877
     -1.5922
     -1.5976
     -1.6044
     -1.6129
     -1.6269
     -1.6512
     -1.6998
     -1.8621
      1.7252
      1.6124
      1.5930
      1.5916
      1.5708
      1.5708
      1.5708 ];
semilogx(w,p,'b*-'), hold
```



Using `unwrap` to correct the phase angle, the resulting jump is `2.6959`, which is less than the default jump tolerance π. This figure plots the new curve over the original curve.

```
semilogx(w,unwrap(p),'r*-')
```



Example 2

Array `P` features smoothly increasing phase angles except for discontinuities at elements `(3,1)` and `(1,2)`.

```
P = [      0    7.0686    1.5708    2.3562
      0.1963    0.9817    1.7671    2.5525
      6.6759    1.1781    1.9635    2.7489
      0.5890    1.3744    2.1598    2.9452 ]
```

The function `Q = unwrap(P)` eliminates these discontinuities.

```
Q =
            0    7.0686    1.5708    2.3562
      0.1963    7.2649    1.7671    2.5525
      0.3927    7.4613    1.9635    2.7489
      0.5890    7.6576    2.1598    2.9452
```

See Also

`abs` | `angle`

Introduced before R2006a

unzip

Extract contents of zip file

Syntax

```
unzip(zipfilename)
unzip(zipfilename,outputdir)
filenames = unzip(zipfilename,outputdir)
```

Description

`unzip(zipfilename)` extracts the archived contents of `zipfilename` into the current folder, preserving the files' attributes and timestamps. The `unzip` function can extract files from your local system or files from an Internet URL.

`unzip(zipfilename,outputdir)` extracts the contents of `zipfilename` into the folder `outputdir`.

`filenames = unzip(zipfilename,outputdir)` returns the names of the extracted files in the string cell array `filenames`. Specifying `outputdir` is optional.

Input Arguments

`zipfilename`

String that specifies the name of the zip file.

If `zipfilename` does not include the full path, `unzip` searches for the file in the current folder and along the MATLAB path. If you do not specify the file extension, `unzip` appends `.zip`.

If you are downloading a URL, `zipfilename` must include the protocol type (for example, `http://`). The `unzip` function downloads the URL to the temporary folder on your system, and deletes the URL on cleanup.

`outputdir`

String that specifies the target folder for the extracted files.

Default: current folder (`'.'`)

Output Arguments

`entrynames`

Cell array of strings that contain the paths of the extracted files.

If `outputdir` specifies a relative path, `filenames` contains the relative path. If `outputdir` specifies an absolute path, `filenames` contains the absolute path.

Examples

Copy the example MAT-files to the folder `archive`:

```
% Zip the example MAT-files to examples.zip
zip('examples.zip','*.mat',...
    fullfile(matlabroot,'toolbox','matlab','demos'))

% Unzip examples.zip to the folder 'archive'
unzip('examples','archive')
```

Download Cleve Moler's "Numerical Computing with MATLAB" examples to the output folder `ncm`:

```
url ='http://www.mathworks.com/moler/ncm.zip';
ncmFiles = unzip(url,'ncm')
```

Alternatives

To extract files from a zip file using the Current Folder browser, select the zip file, right-click to open the context menu, and then select **Extract**.

More About

Tips

- `unzip` does not support password-protected or encrypted zip archives.

- If any files in the target folder have the same name as files in the zip file, and you have write permission to the files, `unzip` overwrites the existing files with the archived versions. If you do not have write permission, `unzip` issues a warning.
- Extract files that contain non-7-bit ASCII characters on a machine that has the appropriate language/encoding settings.

See Also

`gzip` | `gunzip` | `tar` | `untar` | `zip` | `fileattrib`

Introduced before R2006a

uplus, +

Unary plus

Syntax

```
C = +A
C = uplus(A)
```

Description

`C = +A` returns array `A` and stores it in `C`.

`C = uplus(A)` is an alternate way to execute `+A`, but is rarely used. It enables operator overloading for classes.

Examples

Unary Plus of Matrix

Create a 2-by-2 matrix, `A`.

```
A = [1 -3; -2 4]
```

```
A =

     1    -3
    -2     4
```

Use unary plus on `A`.

```
C = +A
```

```
C =

     1    -3
    -2     4
```

`C` and `A` are the same.

Input Arguments

`A` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

More About

- "Array vs. Matrix Operations"
- "Operator Precedence"

See Also

`plus` | `uminus`

upper

Convert string to uppercase

Syntax

```
t = upper('str')
B = upper(A)
```

Description

`t = upper('str')` converts any lowercase characters in the string `str` to the corresponding uppercase characters and leaves all other characters unchanged.

`B = upper(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `upper` to each string within `A`.

Examples

`upper('attention!')` is `ATTENTION!`.

More About

Tips

Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`lower`

Introduced before R2006a

urlread

Download URL content to MATLAB string (not recommended)

Compatibility

`urlread` is not recommended. Use `webread` or `webwrite` instead.

Syntax

```
str = urlread(URL)
str = urlread(URL,Name,Value)
[str,status] = urlread( ___ )
```

Description

`str = urlread(URL)` downloads the HTML web content from the specified `URL` into the string `str`. `urlread` does not retrieve hyperlink targets and images.

`str = urlread(URL,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[str,status] = urlread( ___ )` suppresses the display of error messages, using any of the input arguments in the previous syntaxes. When the operation is successful, `status` is `1`. Otherwise, `status` is `0`

Examples

Download Web Content by Specifying Complete URL

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlread`.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
```

```
    '?term=urlread'];
str = urlread(fullURL);
```

`urlread` reads from the specified URL and downloads the HTML content to the string `str`.

Download Web Content Related to a Term

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlread`.

```
URL = 'http://www.mathworks.com/matlabcentral/fileexchange';
str = urlread(URL,'Get',{'term','urlread'});
```

`urlread` reads from http://www.mathworks.com/matlabcentral/fileexchange/?term=urlread and downloads the HTML content to the string `str`.

Specify Timeout Duration

Download content from a page on the MATLAB Central File Exchange as in the first example, and specify a timeout duration of 5 seconds.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
    '?term=urlread'];
str = urlread(fullURL,'Timeout',5);
```

Input Arguments

`URL` — Content location

string

Content location, specified as a string. Include the transfer protocol, such as `http`, `ftp`, or `file`.

Example: `'http://www.mathworks.com/matlabcentral'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Timeout',10,'Charset','UTF-8'` specifies that `urlread` should time out after 10 seconds, and the character encoding of the file is UTF-8.

`'Get'` — Data to send to the web form using the GET method

cell array

Parameters of the data to send to the web form using the GET method, specified as the comma-separated pair consisting of `'get'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Get'` includes the data in the URL, separated by `?` and `&` characters.

Example: `'Get',{'term','urlread'}`

`'Post'` — Data to send to the web form using the POST method

cell array

Parameters of the data to send to the web form using the POST method, specified as the comma-separated pair consisting of `'post'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Post'` submits the data as part of the request headers, not explicitly in the URL.

`'Charset'` — Character encoding

string

Character encoding, specified as the comma-separated pair consisting of `'Charset'` and a string. If you do not specify `Charset`, the function attempts to determine the character encoding from the headers of the file. If the character encoding cannot be determined, `Charset` defaults to the native encoding for the file protocol, and UTF-8 for all other protocols.

Example: `'Charset','ISO-8859-1'`

`'Timeout'` — Timeout duration

scalar

Timeout duration in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a scalar. The timeout duration determines when the function errors rather than continues to wait for the server to respond or send data.

Example: `'Timeout',10`

`'UserAgent'` — Client user agent identification
string

Client user agent identification, specified as the comma-separated pair consisting of `'UserAgent'` and a string.

Example: `'UserAgent','MATLAB R2012b'`

`'Authentication'` — HTTP authentication mechanism
`'Basic'`

HTTP authentication mechanism, specified as the comma-separated pair consisting of `'Authentication'` and a string. Currently, only the value `'Basic'` is supported. `'Authentication','Basic'` specifies basic authentication.

If you include the `Authentication` argument, you must also include the `Username` and `Password` arguments.

`'Username'` — User identifier
string

User identifier, specified as the comma-separated pair consisting of `'Username'` and a string. If you include the `Username` argument, you must also include the `Password` and `Authentication` arguments.

Example: `'Username','myName'`

`'Password'` — User authentication password
string

User authentication password, specified as the comma-separated pair consisting of `'Password'` and a string. If you include the `Password` argument, you must also include the `Username` and `Authentication` arguments.

Example: `'Password','myPassword123'`

Output Arguments

`str` — Contents of the file at the specified URL
string

Contents of the file at the specified URL, returned as a string. For example, if the URL corresponds to an HTML page, `str` contains the text and markup in the HTML file. If the URL corresponds to a binary file, `str` is not readable.

`status` — Download status

`1 | 0`

Download status, returned as either `1` or `0`. When the download is successful, `status` is `1`. Otherwise, `status` is `0`.

More About

Tips

- `urlread` saves web content to a string. To save content to a file, use `urlwrite`.
- `urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.
- "Specify Proxy Server Settings for Connecting to the Internet"

See Also

`ftp | mget | urlwrite | web`

Introduced before R2006a

urlwrite

Download URL content and save to file (not recommended)

Compatibility

`urlwrite` is not recommended. Use `websave` instead.

Syntax

```
urlwrite(URL,filename)
urlwrite(URL,filename,Name,Value)

[filestr,status] = urlwrite( ___ )
```

Description

`urlwrite(URL,filename)` reads web content at the specified `URL` and saves it to the file specified by `filename`.

`urlwrite(URL,filename,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[filestr,status] = urlwrite( ___ )` stores the file path in variable `filestr`, and suppresses the display of error messages, using any of the input arguments in the previous syntaxes. When the operation is successful, `status` is `1`. Otherwise, `status` is `0`.

Examples

Download Web Content by Specifying Complete URL

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlwrite`. Save the results to `samples.html` in the current folder.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
           '?term=urlwrite'];
filename = 'samples.html';
urlwrite(fullURL,filename);
```

View the file.

```
web(filename)
```

Download Web Content Related to a Term

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlwrite`. Save the results to `samples.html` in the current folder.

```
URL = 'http://www.mathworks.com/matlabcentral/fileexchange';
filename = 'samples.html';
urlwrite(URL,filename,'get',{'term','urlwrite'});
```

`urlwrite` downloads the HTML content from http://www.mathworks.com/matlabcentral/fileexchange/?term=urlwrite and writes it to `samples.html`.

Specify Timeout Duration

Download content from a page on the MATLAB Central File Exchange as in the first example, and specify a timeout duration of 5 seconds.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
    '?term=urlwrite'];
filename = 'samples.html';
urlwrite(fullURL,filename,'Timeout',5);
```

Input Arguments

`URL` — Content location

string

Content location, specified as a string. Include the transfer protocol, such as `http`, `ftp`, or `file`.

Example: `'http://www.mathworks.com/matlabcentral'`

`filename` — Name of file to store web content

string

Name of the file to store the web content, specified as a string. If you do not specify the path for `filename`, `urlwrite` saves the file in the current folder.

Example: `'myfile.html'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Timeout',10,'Charset','UTF-8'` specifies that `urlread` should time out after 10 seconds, and the character encoding of the file is UTF-8.

`'Get'` — Data to send to the web form using the GET method

cell array

Parameters of the data to send to the web form using the GET method, specified as the comma-separated pair consisting of `'get'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Get'` includes the data in the URL, separated by `?` and `&` characters.

Example: `'Get',{'term','urlread'}`

`'Post'` — Data to send to the web form using the POST method

cell array

Parameters of the data to send to the web form using the POST method, specified as the comma-separated pair consisting of `'post'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Post'` submits the data as part of the request headers, not explicitly in the URL.

`'Charset'` — Character encoding

string

Character encoding, specified as the comma-separated pair consisting of `'Charset'` and a string. If you do not specify `Charset`, the function attempts to determine the character encoding from the headers of the file. If the character encoding cannot be determined, `Charset` defaults to the native encoding for the file protocol, and UTF-8 for all other protocols.

Example: `'Charset','ISO-8859-1'`

`'Timeout'` — Timeout duration
scalar

Timeout duration in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a scalar. The timeout duration determines when the function errors rather than continues to wait for the server to respond or send data.

Example: `'Timeout',10`

`'UserAgent'` — Client user agent identification
string

Client user agent identification, specified as the comma-separated pair consisting of `'UserAgent'` and a string.

Example: `'UserAgent','MATLAB R2012b'`

`'Authentication'` — HTTP authentication mechanism
`'Basic'`

HTTP authentication mechanism, specified as the comma-separated pair consisting of `'Authentication'` and a string. Currently, only the value `'Basic'` is supported. `'Authentication','Basic'` specifies basic authentication.

If you include the `Authentication` argument, you must also include the `Username` and `Password` arguments.

`'Username'` — User identifier
string

User identifier, specified as the comma-separated pair consisting of `'Username'` and a string. If you include the `Username` argument, you must also include the `Password` and `Authentication` arguments.

Example: `'Username','myName'`

`'Password'` — User authentication password
string

User authentication password, specified as the comma-separated pair consisting of `'Password'` and a string. If you include the `Password` argument, you must also include the `Username` and `Authentication` arguments.

Example: `'Password','myPassword123'`

Output Arguments

`filestr` — Path of the file

string

Path of the file specified by `filename`, returned as a string.

`status` — Download status

`1` | `0`

Download status, returned as either `1` or `0`. When the download is successful, `status` is `1`. Otherwise, `status` is `0`.

More About

Tips

- `urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.
- "Specify Proxy Server Settings for Connecting to the Internet"

See Also

`ftp` | `mget` | `urlread` | `web`

Introduced before R2006a

usejava

Determine if Java feature is available

Syntax

```
tf = usejava(feature)
```

Description

`tf = usejava(feature)` returns logical `1` (`true`) if the specified feature is supported. Otherwise, it returns logical `0` (`false`). Use for error handling if Java feature is unavailable.

Examples

Display Error Message

Use the following code snippet to test that the AWT GUI components are available before attempting to display a Java Frame.

```
if usejava('awt')
   myFrame = java.awt.Frame;
else
   disp('Unable to open a Java Frame');
end
```

If the AWT is not available on your system, MATLAB displays the message.

Call error Function

Use the following code snippet to terminate a script if MATLAB does not have access to JVM software.

The variable, `filename`, is a function that contains Java code.

```
if ~usejava('jvm')
   error([filename ' requires Java to run.']);
```

```
end
```

Input Arguments

`feature` — Java feature

`'awt' | 'desktop' | 'jvm' | 'swing'`

Java feature, specified as one of these values:

`'awt'`	UI components in the Java Abstract Window Toolkit (AWT) are available.
`'desktop'`	MATLAB interactive desktop is running.
`'jvm'`	Java Virtual Machine software (JVM) is running.
`'swing'`	Swing components (Java lightweight UI components in the Java Foundation Classes) are available.

More About

- "Bringing Java Classes into MATLAB Workspace"

See Also

`error` | `javachk`

Introduced before R2006a

userpath

View or change user portion of search path

Syntax

```
userpath

userpath(newpath)
userpath('reset')
userpath('clear')
```

Description

`userpath` returns a string specifying the first folder on the search path. MATLAB adds the *`userpath`* to the search path upon startup.

`userpath(newpath)` sets the primary *`userpath`* folder to `newpath`. The `newpath` folder appears at the top of the search path immediately and at startup in future sessions. MATLAB removes the folder previously specified by *`userpath`* from the search path.

`userpath('reset')` sets the primary *`userpath`* folder to the default for that platform, creating the `Documents/MATLAB` (or `My Documents/MATLAB`) folder, if it does not exist. MATLAB immediately adds the default folder to the top of the search path, and also adds it to the search path at startup in future sessions. On Linux, the `MATLAB` directory is not created if the `Documents` directory does not exist.

`userpath('clear')` clears the value for the primary *`userpath`* immediately, and for future MATLAB sessions. MATLAB removes the folder previously specified by *`userpath`* from the search path.

Examples

View userpath Folder

This example assumes `userpath` is set to the default value on the Windows XP platform, `My Documents\MATLAB`. Start MATLAB and display the current folder:

```
pwd
```

```
H:\My Documents\MATLAB
```

In this example, `H` is the drive at which `My Documents` is located.

Confirm that the current folder is the *`userpath`*.

```
userpath
```

```
H:\My Documents\MATLAB;
```

Display the search path.

```
path
```

```
MATLABPATH

 H:\My Documents\MATLAB
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

MATLAB returns the search path. The *`userpath`* portion is at the top.

Set New Value for userpath

Assume *`userpath`* is set to the default value on the Windows XP platform, `My Documents\MATLAB`. Change the value from the default for *`userpath`* to `C:\Research_Project`.

```
newpath = 'C:\Research_Project';
userpath(newpath)
```

View the effect of the change on the search path.

```
path
```

```
MATLABPATH

 C:\Research_Project
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

MATLAB displays the search path, with the new value for *userpath* portion at the top. Note that MATLAB automatically removed the previous value of *userpath*, `H:\My Documents\MATLAB`, from the search path when you assigned a new value to *userpath*.

Clear the Value for userpath

Assume `userpath` is set to the default value and you do not want any folders to be added to the search path upon startup. Confirm the default is currently set.

```
userpath
```

```
H:\My Documents\MATLAB;
```

Verify that the `userpath` folder is at the top of the search path.

```
path
```

```
MATLABPATH

H:\My Documents\MATLAB
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Clear the value.

```
userpath('clear')
```

Verify the result.

```
userpath
```

```
ans =
     ''
```

Confirm the *userpath* folder was removed from the search path.

```
path
```

```
MATLABPATH

C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Note: If you clear the value for *userpath*, the startup folder will not necessarily be on the search path. Removing the *userpath* folder from the search path *and* saving the changes to the path has the same effect.

Input Arguments

newpath — New userpath value

string

New userpath value, specified as a string. `newpath` must be an absolute path.

Example: `'C:\myFolder'`

More About

Tips

- By default, the startup folder is the *userpath* folder when you start MATLAB by double-clicking the shortcut on Windows platforms, or by double-clicking the application on Mac platforms. If you clear the value for *userpath*, the startup folder will not necessarily be on the search path. Removing the *userpath* folder from the search path *and* saving the changes to the path has the same effect.

  To specify the current folder, set the **Initial working folder** preference, described in "General Preferences".
- On Mac and Linux platforms, you can automatically add subfolders to the top of the search path upon startup by specifying the path for the subfolders via the `MATLABPATH` environment variable.

- "What Is the MATLAB Search Path?"
- "MATLAB Startup Folder"

See Also

`addpath` | `path` | `rmpath` | `savepath` | `startup`

Using ValueIterator Objects

An iterator over intermediate values for use with mapreduce

The `mapreduce` function automatically creates a `ValueIterator` object during execution and uses it to store the values associated with each unique intermediate key added by the map function. Although you never need to explicitly create a `ValueIterator` object to use `mapreduce`, you do need to interact with this object in the reduce function. Use the `hasnext` and `getnext` object functions to retrieve the values associated with each unique key in the intermediate `KeyValueStore` object.

Examples

Get Values from ValueIterator in Reduce Function

Use the `hasnext` and `getnext` functions in a `while` loop within the reduce function to iteratively get values from the `ValueIterator`. For example,

```
function MeanDistReduceFun(sumLenKey, sumLenIter, outKVStore)
    sumLen = [0, 0];
    while hasnext(sumLenIter)
        sumLen = sumLen + getnext(sumLenIter);
    end
    add(outKVStore, 'Mean', sumLen(1)/sumLen(2));
end
```

Always call `hasnext` before `getnext` to confirm availability of a value. `mapreduce` returns an error if you call `getnext` with no remaining values in the `ValueIterator`.

Properties

`Key` — Intermediate key

numeric scalar | string

Intermediate key, specified as a numeric scalar or string. `Key` is one of the unique keys added by a map function. All the values in the `ValueIterator` object are associated with this key.

Object Functions

`hasnext` `getnext`

Create Object

The `mapreduce` function automatically creates `ValueIterator` objects during execution.

See Also

`mapreduce`

More About

- “Getting Started with MapReduce”
- Using KeyValueStore Objects

validateattributes

Check validity of array

Syntax

```
validateattributes(A,classes,attributes)
validateattributes(A,classes,attributes,argIndex)
validateattributes(A,classes,attributes,funcName)
validateattributes(A,classes,attributes,funcName,varName)
validateattributes(A,classes,attributes,funcName,varName,argIndex)
```

Description

`validateattributes(A,classes,attributes)` validates that array `A` belongs to at least one of the specified classes (or its subclass) and has all of the specified attributes. If `A` does not meet the criteria, MATLAB throws an error and displays a formatted error message. Otherwise, `validateattributes` completes without displaying any output.

`validateattributes(A,classes,attributes,argIndex)` includes the position of the input in your function argument list as part of any generated error messages.

`validateattributes(A,classes,attributes,funcName)` includes the specified function name in generated error identifiers.

`validateattributes(A,classes,attributes,funcName,varName)` includes the specified variable name in generated error messages.

`validateattributes(A,classes,attributes,funcName,varName,argIndex)` includes the specified information in generated error messages or identifiers.

Examples

Validate the Size of an Array

```
classes = {'numeric'};
attributes = {'size',[4,6,2]};
```

```
A = rand(3,5,2);
validateattributes(A,classes,attributes)
```

```
Expected input to be of size 4x6x2 when it is actually size 3x5x2.
```

Because A did not match the specified attributes, MATLAB throws an error message.

Validate Array Monotonicity

Determine if an array is increasing or nondecreasing.

```
A = [1 5 8 2;
     9 6 9 4]
validateattributes(A, {'double'},{'nondecreasing'})
validateattributes(A, {'double'},{'increasing'})
```

```
A =

     1     5     8     2
     9     6     9     4
```

Since `A` is both increasing and nondecreasing, `validateattributes` does not throw an error for either attribute check.

Setting `A(2,3)` equal to `A(1,3)` results in a column that is no longer strictly increasing, so `validateattributes` throws an error.

```
A(2,3) = 8
validateattributes(A, {'double'},{'increasing'})
```

```
A =

     1     5     8     2
     9     6     8     4

Expected input to be increasing valued.
```

However, the columns remain nondecreasing since each column element is equal to or greater than the next. The following code does not throw an error.

```
validateattributes(A, {'double'},{'nondecreasing'})
```

Check the Attributes of a Complex Number

Assuming that `a` is to be the second input argument to a function, check that it is nonnegative.

```
a = complex(1,1);
validateattributes(a,{'numeric'},{'nonnegative'},2)

Expected input number 2 to be nonnegative.
```

Because complex numbers lack a well-defined ordering in the complex plane, `validateattributes` does not recognize them as positive or negative.

Ensure Array Values Are Within a Specified Range

Check that the values in an array are 8-bit integers between 0 and 10.

Assume that this code occurs in a function called `Rankings`.

```
classes = {'uint8','int8'};
attributes = {'>',0,'<',10};
funcName = 'Rankings';
A = int8(magic(4));

validateattributes(A,classes,attributes,funcName)

Error using Rankings
Expected input to be an array with all of the values < 10.
```

Validate Function Input Parameters Using `inputParser`

Create a custom function that checks input parameters with `inputParser`, and use `validateattributes` as the validating function for the `addRequired` and `addOptional` methods.

Define the function.

```
function a = findArea(shape,dim1,varargin)
   p = inputParser;
   charchk = {'char'};
   numchk = {'numeric'};
   nempty = {'nonempty'};

   addRequired(p,'shape',@(x)validateattributes(x,charchk,nempty))
   addRequired(p,'dim1',@(x)validateattributes(x,numchk,nempty))
   addOptional(p,'dim2',1,@(x)validateattributes(x,numchk,nempty))
   parse(p,shape,dim1,varargin{:})

   switch shape
```

```
      case 'circle'
         a = pi * dim1.^2;
      case 'rectangle'
         a = dim1 .* p.Results.dim2;
   end
end
```

Call the function with a nonnumeric third input.

```
myarea = findArea('rectangle',3,'x')
```

```
Error using findArea (line 10)
Argument 'dim2' failed validation with error:
Expected input to be one of these types:

double, single, uint8, uint16, uint32, uint64, int8, int16, int32, int64

Instead its type was char.
```

Validate Arguments of a Function

Check the inputs of a function and include information about the input name and position in generated error.

Define the function.

```
function v = findVolume(shape,ht,wd,ln)
   validateattributes(shape,{'char'},{'nonempty'},mfilename,'Shape',1)
   validateattributes(ht,{'numeric'},{'nonempty'},mfilename,'Height',2)
   validateattributes(wd,{'numeric'},{'nonempty'},mfilename,'Width',3)
   validateattributes(ln,{'numeric'},{'nonempty'},mfilename,'Length',4)
```

Call the function without the `shape` input string.

```
vol = findVolume(10,7,4)
```

```
Error using findVolume
Expected input number 1, Shape, to be one of these types:

char

Instead its type was double.

Error in findVolume (line 2)
validateattributes(shape,{'char'},{'nonempty'},mfilename,'Shape',1)
```

The function name becomes part of the error identifier.

```
MException.last.identifier
```

```
ans =

MATLAB:findVolume:invalidType
```

Input Arguments

A — Input

any type of array

Input, specified as any type of array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `cell` | `function_handle`
Complex Number Support: Yes

`classes` — Valid data types

cell array of strings

Valid data types, specified as a cell array of strings. Each string can be the name of any built-in or custom class, including:

`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `true` or `false`
`'char'`	Character or string

`'struct'`	Structure array
`'cell'`	Cell array
`'function_handle'`	function handle
`'numeric'`	Any data type for which the `isa(A,'numeric')` function returns true, including `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `single`, or `double`
`'<class_name>'`	Any other class name

Data Types: `cell`

`attributes` — Valid attributes

cell array of strings

Valid attributes, specified as a cell array of strings.

Some attributes also require numeric values, such as attributes that specify the size or number of elements of A. For these attributes, the numeric value or vector must immediately follow the attribute name string in the cell array.

Attributes that describe the size and shape of array A:

`'2d'`	Two-dimensional array, including scalars, vectors, matrices, and empty arrays
`'3d'`	Array with three or fewer dimensions
`'column'`	Column vector, `N`-by-1
`'row'`	Row vector, 1-by-`N`
`'scalar'`	Scalar value, 1-by-1
`'vector'`	Row or column vector, or a scalar value
`'size', [d1,...,dN]`	Array with dimensions `d1`-by-...-by-`dN`. To skip checking a particular dimension, specify `NaN` for that dimension, such as `[3,4,NaN,2]`.
`'numel', N`	Array with `N` elements
`'ncols', N`	Array with `N` columns
`'nrows', N`	Array with `N` rows
`'ndims', N`	`N`-dimensional array

`'square'`	Square matrix; in other words, a two-dimensional array with equal number of rows and columns
`'diag'`	Diagonal matrix
`'nonempty'`	No dimensions that equal zero
`'nonsparse'`	Array that is not sparse

Attributes that specify valid ranges for values in A:

`'>', N`	All values greater than `N`
`'>=', N`	All values greater than or equal to `N`
`'<', N`	All values less than `N`
`'<=', N`	All values less than or equal to `N`

Attributes that check types of values in a numeric or logical array, `A`:

`'binary'`	Array of ones and zeros
`'even'`	Array of even integers (includes zero)
`'odd'`	Array of odd integers
`'integer'`	Array of integer values
`'real'`	Array of real values
`'finite'`	Array of finite values
`'nonnan'`	No NaN (Not a Number) elements
`'nonnegative'`	No elements less than zero
`'nonzero'`	No elements equal to zero
`'positive'`	No elements less than or equal to zero
`'decreasing'`	Each element of a column is less than the previous element and no element is `NaN`
`'increasing'`	Each element of a column is greater than the previous element and no element is `NaN`
`'nondecreasing'`	Each element of a column is greater than or equal to the previous element and no element is `NaN`
`'nonincreasing'`	Each element of a column is less than or equal to the previous element and no element is `NaN`

Data Types: `cell`

`funcName` — Name of function whose input you are validating

string

Name of function whose input you are validating, specified as a string. If you specify an empty string, `''`, the `validateattribute` function ignores the `funcName` input.

Data Types: `char`

`varName` — Name of input variable

string

Name of input variable, specified as a string. If you specify an empty string, `''`, the `validateattribute` function ignores the `varName` input.

Data Types: `char`

`argIndex` — Position of input argument

positive integer

Position of input argument, specified as a positive integer.

Data Types: `double`

See Also

`inputParser` | `is*` | `isa` | `isnumeric` | `validatestring`

validatestring

Check validity of text string

Syntax

```
validStr = validatestring(str,validStrings)
validStr = validatestring(str,validStrings,argIndex)
validStr = validatestring(str,validStrings,funcName)
validStr = validatestring(str,validStrings,funcName,varName)
validStr = validatestring(str,validStrings,funcName,varName,
argIndex)
```

Description

`validStr = validatestring(str,validStrings)` checks the validity of text string `str`. If `str` is an unambiguous, case-insensitive match to a string in cell array `validStrings`, the `validatestring` function returns the matching string in `validstr`. Otherwise, MATLAB throws an error and issues a formatted error message.

`validStr = validatestring(str,validStrings,argIndex)` includes the position of the input in your function argument list as part of any generated error messages.

`validStr = validatestring(str,validStrings,funcName)` includes the specified function name in generated error identifiers.

`validStr = validatestring(str,validStrings,funcName,varName)` includes the specified variable name in generated error messages.

`validStr = validatestring(str,validStrings,funcName,varName, argIndex)` includes the specified information in generated error messages or identifiers.

Input Arguments

`str`

String to validate, of type `char`.

`validStrings`

Cell array of allowed strings.

`funcName`

String that specifies the name of the function whose input you are validating. If you specify an empty string, `''`, the `validatestring` function ignores the `funcname` input.

`varName`

String that specifies the name of the input variable. If you specify an empty string, `''`, the `validatstring` function ignores the `varName` input.

`argIndex`

Positive integer that specifies the position of the input argument.

Output Arguments

`validStr`

String that contains the element of `validStrings` that is an unambiguous, case-insensitive match to `str`.

Example — Match `'ball'` with . . .	Return Value	Type of Match
`ball`, `barn`, `bell`	`ball`	Exact match
`balloon`, `barn`	`balloon`	Partial match (leading characters)
`ballo`, `balloo`, `balloon`	`ballo` (shortest match)	Multiple partial matches where each string is a subset of another
`balloon`, `ballet`	Error	Multiple partial matches to unique strings
`barn`, `bell`	Error	No match

Examples

Check whether a string is in a set of valid values.

```
str = 'won';
validStrings = {'wind','wonder','when'};

validStr = validatestring(str,validStrings)
```

Because `str` is a partial match to a unique string, this code returns

```
validStr =
     wonder
```

However, if `str` is a partial match to multiple strings, and the strings are not subsets of each other, `validatestring` throws an error and displays a formatted message.

```
str = 'won';
validStrings = {'wonder','wondrous','wonderful'};

validStr = validatestring(str,validStrings)
```

The error message is

```
Expected argument to match one of these strings:

  wonder, wondrous, wonderful

The input, 'won', matched more than one valid string.
```

Check inputs to a custom function, and include information about the input name and position in generated errors.

```
function a = findArea(shape,ht,wd,units)
   a = 0;
   expectedShapes = {'square','rectangle','triangle'};
   expectedUnits = {'cm','m','in','ft','yds'};

   shapeName = validatestring(shape,expectedShapes,mfilename,'Shape',1)
   unitAbbrev = validatestring(units,expectedUnits,mfilename,'Units',4)
```

When you call the function with valid input strings,

```
myarea = findArea('rect',10,3,'cm')
```

the function stores the inputs in local variables `shapeName` and `unitAbbrev`:

```
shapeName =
     rectangle
```

```
unitAbbrev =
     cm
```

However, if the inputs are not valid, such as

```
myarea = findArea('circle',10,3,'cm')
```

MATLAB displays

```
Error using findArea
Expected argument 1, Shape, to match one of these strings:

  square, rectangle, triangle

The input, 'circle', did not match any of the valid strings.
```

The function name is part of the error identifier, so

```
MException.last.identifier
```

returns

```
MATLAB:findArea:unrecognizedStringChoice
```

See Also

`validateattributes` | `is*` | `isa` | `inputParser`

values

Class: containers.Map
Package: containers

Identify values in `containers.Map` object

Syntax

```
valueSet = values(mapObj)
valueSet = values(mapObj,keySet)
```

Description

`valueSet = values(mapObj)` returns all of the values in `mapObj`.

`valueSet = values(mapObj,keySet)` returns values that correspond to the specified keys.

Input Arguments

`mapObj`

Object of class `containers.Map`.

`keySet`

Cell array that specifies keys in `mapObj`.

Output Arguments

`valueSet`

Cell array containing values from `mapObj`. If you specify `keySet`, the `valueSet` array has the same size and dimensions as `keySet`.

Examples

Get All Values in a Map

Create a map, and view all values in the map:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
mapObj = containers.Map(myKeys,myValues);

valueSet = values(mapObj)
```

This code returns 1-by-3 cell array `valueSet`:

```
valueSet =
    [1]    [2]    [3]
```

Get Selected Values in a Map

View the values corresponding to keys `a` and `c` in `mapObj`, created in the previous example:

```
keySet = {'a','c'};
valueSet = values(mapObj,keySet)
```

This code returns 1-by-2 cell array `valueSet`:

```
valueSet =
    [1]    [3]
```

See Also

`keys` | `isKey` | `containers.Map`

vander

Vandermonde matrix

Syntax

```
A = vander(v)
```

Description

`A = vander(v)` returns the "Vandermonde Matrix" on page 1-8781 such that its columns are powers of the vector v.

Examples

Find the Vandermonde Matrix for Vector Input

Use the colon operator to create vector `v`. Find the Vandermonde matrix for `v`.

```
v = 1:.5:3
A = vander(v)
```

```
v =
    1.0000    1.5000    2.0000    2.5000    3.0000

A =
    1.0000    1.0000    1.0000    1.0000    1.0000
    5.0625    3.3750    2.2500    1.5000    1.0000
   16.0000    8.0000    4.0000    2.0000    1.0000
   39.0625   15.6250    6.2500    2.5000    1.0000
   81.0000   27.0000    9.0000    3.0000    1.0000
```

Find the alternate form of the Vandermonde matrix using `fliplr`.

```
A = fliplr(vander(v))
```

```
A =
    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0000    1.5000    2.2500    3.3750    5.0625
    1.0000    2.0000    4.0000    8.0000   16.0000
```

```
    1.0000    2.5000    6.2500   15.6250   39.0625
    1.0000    3.0000    9.0000   27.0000   81.0000
```

Input Arguments

v — Input

numeric vector

Input, specified as a numeric vector.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Vandermonde Matrix

For input vector $v = \begin{bmatrix} v_1 & v_2 & \dots & v_N \end{bmatrix}$, the Vandermonde matrix is

$$\begin{bmatrix} v_1^{N-1} & \cdots & v_1^1 & v_1^0 \\ v_2^{N-1} & \cdots & v_2^1 & v_2^0 \\ & \iddots & \vdots & \vdots \\ v_N^{N-1} & & v_N^1 & v_N^0 \end{bmatrix}$$

The matrix is described by the formula $A(i,j) = v(i)^{(N-j)}$ such that its columns are powers of the vector v.

An alternate form of the Vandermonde matrix flips the matrix along the vertical axis, as shown. Use `fliplr(vander(v))` to return this form.

$$\begin{bmatrix} v_1^0 & v_1^1 & \cdots & v_1^{N-1} \\ v_2^0 & v_2^1 & \cdots & v_2^{N-1} \\ \vdots & \vdots & \ddots & \\ v_N^0 & v_N^1 & & v_N^{N-1} \end{bmatrix}$$

See Also

`gallery` | `hilb` | `mpower` | `pascal` | `power` | `rosser` | `toeplitz`

Introduced before R2006a

var

Variance

Syntax

```
V = var(A)
V = var( ___ ,w)
V = var( ___ ,dim)
V = var( ___ ,nanflag)
```

Description

`V = var(A)` returns the variance of the elements of `A`along the first array dimension whose size does not equal 1.

- If `A` is a vector of observations, the variance is a scalar.
- If `A` is a matrix whose columns are random variables and whose rows are observations, `V` is a row vector containing the variances corresponding to each column.
- If `A` is a multidimensional array, then `var(A)` treats the values along the first array dimension whose size does not equal 1 as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.
- The variance is normalized by the number of observations`-1` by default.
- If `A` is a scalar, `var(A)` returns `0`. If `A` is a `0`-by-`0` empty array, `var(A)` returns `NaN`.

`V = var( ___ ,w)` specifies a weighting scheme for any of the previous syntaxes. When `w = 0` (default), `V` is normalized by the number of observations`-1`. When `w = 1`, it is normalized by the number of observations. `w` can also be a weight vector containing nonnegative elements. In this case, the length of `w` must equal the length of the dimension over which `var` is operating.

`V = var( ___ ,dim)` returns the variance along the dimension `dim` for any of the previous syntaxes. To maintain the default normalization while specifying the dimension of operation, set `w = 0` in the second argument.

`V = var( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. For example, `var(A,'includenan')` includes all `NaN` values in `A` while `var(A,'omitnan')` ignores them.

Examples

Variance of Matrix

Create a matrix and compute its variance.

```
A = [4 -7 3; 1 4 -2; 10 7 9];
var(A)
```

```
ans =

   21.0000   54.3333   30.3333
```

Variance of Array

Create a 3-D array and compute its variance.

```
A(:,:,1) = [1 3; 8 4];
A(:,:,2) = [3 -4; 1 2];
var(A)
```

```
ans(:,:,1) =

   24.5000    0.5000


ans(:,:,2) =

     2    18
```

Specify Variance Weight Vector

Create a matrix and compute its variance according to a weight vector `w`.

```
A = [5 -4 6; 2 3 9; -1 1 2];
w = [0.5 0.25 0.25];
var(A,w)
```

```
ans =
```

```
    6.1875    9.5000    6.1875
```

Specify Dimension for Variance

Create a matrix and compute its variance along the first dimension.

```
A = [4 -2 1; 9 5 7];
var(A,0,1)
```

```
ans =

   12.5000   24.5000   18.0000
```

Compute the variance of `A` along the second dimension.

```
var(A,0,2)
```

```
ans =

     9
     4
```

Variance Excluding `NaN`

Create a vector and compute its variance, excluding `NaN` values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
V = var(A,'omitnan')
```

```
V =

    5.1970
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

`w` — Weight

`0` (default) | `1` | vector

Weight, specified as one of:

- `0` — normalizes by the number of observations-`1`. If there is only one observation, the weight is 1.
- `1` — normalizes by the number of observations.
- a vector made up of nonnegative scalar weights corresponding to the dimension of `A` along which the variance is calculated.

Data Types: `single` | `double`

`dim` — Dimension to operate along

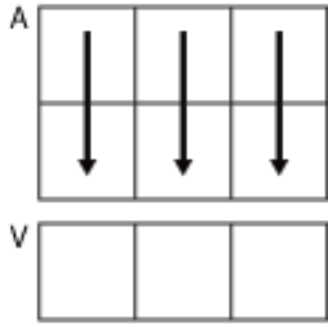
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(V,dim)` is `1`, while the sizes of all other dimensions remain the same.

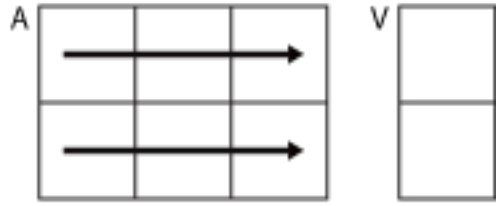
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `var(A,0,1)` returns a row vector containing the variance of the elements in each column.



var(A,0,1)

- If `dim = 2`, then `var(A,0,2)` returns a column vector containing the variance of the elements in each row.



var(A,0,2)

`var` returns an array of zeros the same size as `A` when `dim` is greater than `ndims(A)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
'`includenan`' (default) | '`omitnan`'

`NaN` condition, specified as one of these values:

- '`includenan`' — the variance of input containing `NaN` values is also `NaN`.
- '`omitnan`' — all `NaN` values appearing in either the input array or weight vector are ignored.

Data Types: `char`

More About

Variance

For a random variable vector A made up of N scalar observations, the variance is defined as

$$V = \frac{1}{N-1}\sum_{i=1}^{N} |A_i - \mu|^2$$

where μ is the mean of A,

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i.$$

Some definitions of variance use a normalization factor of N instead of $N-1$, which can be specified by setting `w` to `1`. In either case, the mean is assumed to have the usual normalization factor N.

See Also

`corrcoef` | `cov` | `mean` | `std`

varargin

Variable-length input argument list

Syntax

```
varargin
```

Description

`varargin` is an input variable in a function definition statement that allows the function to accept any number of input arguments. Specify `varargin` using lowercase characters, and include it as the last input argument after any explicitly declared inputs. When the function executes, `varargin` is a 1-by-*N* cell array, where *N* is the number of inputs that the function receives after the explicitly declared inputs.

Examples

Variable Number of Function Inputs

Define a function in a file named `varlist.m` that accepts a variable number of inputs and displays the values of each input.

```
function varlist(varargin)
   fprintf('Number of arguments: %d\n',nargin)
   celldisp(varargin)
```

Call `varlist` with several inputs.

```
varlist(ones(3),'some text',pi)
```

```
Number of arguments: 3

varargin{1} =
     1     1     1
     1     1     1
     1     1     1
```

```
varargin{2} =
some text

varargin{3} =
    3.1416
```

varargin and Declared Inputs

Define a function in a file named `varlist2.m` that expects inputs `X` and `Y`, and accepts a variable number of additional inputs.

```
function varlist2(X,Y,varargin)
   fprintf('Total number of inputs = %d\n',nargin);

   nVarargs = length(varargin);
   fprintf('Inputs in varargin(%d):\n',nVarargs)
   for k = 1:nVarargs
      fprintf('   %d\n', varargin{k})
   end
```

Call `varlist2` with more than two inputs.

```
varlist2(10,20,30,40,50)
```

```
Total number of inputs = 5
Inputs in varargin(3):
   30
   40
   50
```

- “Support Variable Number of Inputs”

More About

- “Argument Checking in Nested Functions”

See Also

`inputname` | `nargin` | `narginchk` | `nargout` | `nargoutchk` | `varargout`

Introduced before R2006a

varargout

Variable-length output argument list

Syntax

```
varargout
```

Description

`varargout` is an output variable in a function definition statement that allows the function to return any number of output arguments. Specify `varargout` using lowercase characters, and include it as the last output argument after any explicitly declared outputs. When the function executes, `varargout` is a 1-by-*N* cell array, where *N* is the number of outputs requested after the explicitly declared outputs.

Examples

Variable Number of Function Outputs

Define a function in a file named `sizeout.m` that returns an output size vector `s` and a variable number of additional scalar values.

```
function [s,varargout] = sizeout(x)
nout = max(nargout,1) - 1;
s = size(x);
for k = 1:nout
   varargout{k} = s(k);
end
```

Output `s` contains the dimensions of the input array `x`. Additional outputs correspond to the individual dimensions within `s`.

Call `sizeout` on a three-dimensional array and request three outputs.

```
[s,rows,cols] = sizeout(rand(4,5,2))
```

```
s =
```

```
     4     5     2

rows =
     4

cols =
     5
```

See Also

varargin | nargin | nargout | nargoutchk | narginchk | inputname

Introduced before R2006a

varfun

Apply function to table variables

Syntax

```
B = varfun(func,A)
B = varfun(func,A,Name,Value)
```

Description

`B = varfun(func,A)` applies the function func separately to each variable of the table `A` and returns the results in the table `B`.

The function `func` must take one input argument and return arrays with the same number of rows each time it is called. The `i`th variable in the output table, `B{:,i}`, is equal to `func(A{:,i})`.

`B = varfun(func,A,Name,Value)` applies the function `func` separately to each variable of the table `A` with additional options specified by one or more Name,Value pair arguments.

For example, you can specify which variables to pass to the function.

Examples

Apply Element-wise Function

Define and apply an element-wise function to the variables of a table to square all the elements.

Define a table containing numeric variables.

```
A = table([0.71;-2.05;-0.35;-0.82;1.57],[0.23;0.12;-0.18;0.23;0.41])
```

```
A =
```

```
    Var1     Var2
    _____    _____

     0.71     0.23
    -2.05     0.12
    -0.35    -0.18
    -0.82     0.23
     1.57     0.41
```

Define the anonymous function to find the square of an input.

```
func = @(x) x.^2;
```

Apply the function to all the variables of table A.

```
B = varfun(func,A)
```

```
B =

    Fun_Var1    Fun_Var2
    ________    ________

    0.5041      0.0529
    4.2025      0.0144
    0.1225      0.0324
    0.6724      0.0529
    2.4649      0.1681
```

The variables of B have names based on the function and the variable names from A.

Apply Function that Returns Scalar From Vector

Compute the mean of each variable in a 5-by-2 table.

Define a table containing numeric variables.

```
A = table([0.71;-2.05;-0.35;-0.82;1.57],[0.23;0.12;-0.18;0.23;0.41])
```

```
A =

    Var1     Var2
    _____    _____

     0.71     0.23
```

```
    -2.05     0.12
    -0.35    -0.18
    -0.82     0.23
     1.57     0.41
```

Define the anonymous function to find the mean of an input.

```
func = @mean;
```

`func` uses an existing MATLAB function to define the operation.

Apply the function to all the variables of table `A`.

```
B = varfun(func,A)
```

```
B =

    mean_Var1    mean_Var2
    _________    _________

    -0.188        0.162
```

`B` is a table containing the average value from each variable. To return a numeric vector instead of a table, you can use `B = varfun(func,A,'OutputFormat','uniform')`.

Apply Function to Groups Within Variables

Compute the group-wise means of variables in a table, `A`, and return them as rows in a table, `B`.

Create a table where one variable defines groups.

```
A = table({'test2';'test1';'test2';'test3';'test1'},...
    [0.71;-2.05;-0.35;-0.82;1.57],[0.23;0.12;-0.18;0.23;0.41])
```

```
A =

     Var1       Var2      Var3
    _______    ______    _____

    'test2'     0.71      0.23
    'test1'    -2.05      0.12
    'test2'    -0.35     -0.18
    'test3'    -0.82      0.23
    'test1'     1.57      0.41
```

Define the anonymous function to find the mean of an input.

```
func = @mean;
```

`func` uses an existing MATLAB function to define the operation.

Apply the function to each group of data defined by `Var1`.

```
B = varfun(func,A,'GroupingVariables','Var1')
```

```
B =

              Var1       GroupCount    mean_Var2    mean_Var3
             _______    __________    _________    _________

    test1    'test1'    2              -0.24        0.265
    test2    'test2'    2               0.18        0.025
    test3    'test3'    1              -0.82         0.23
```

`B` contains row names based on the grouping variables and a variable called `GroupCount` to indicate the number of entries from table `A` in that group.

Input Arguments

`func` — Function

function handle

Function, specified as a function handle. You can define the function in a file or as an anonymous function. If `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

Use the `'OutputFormat','cell'` name-value pair argument, if the function `func` take one input argument and returns arrays with a different numbers of rows each time it is called. Otherwise, `func` must return arrays with the same number of rows.

Example: `func = @(x) x.^2;` conputes the square of each element of an input.

`A` — Input table

table

Input table, specified as a table.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputVariables',2` uses only the second variable in `A` as an input to `func`.

`'InputVariables'` — Variables of `A` to pass to `func`

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector | ...

Variables of `A` to pass to `func`, specified as the comma-separated pair consisting of `'InputVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector, or an anonymous function that returns a logical scalar. If you specify `'InputVariables'` as an anonymous function that returns a logical scalar, `varfun` only passes the variables in `A` where the specified function returns `1` (`true`).

`'GroupingVariables'` — One or more variables in `A` that define groups of rows

positive integer | vector of positive integers | variable name | cell array of variable names | logical vector

One or more variables in `A` that define groups of rows, specified as the comma-separated pair consisting of `'GroupingVariables'` and a positive integer, vector of positive integers, variable name, cell array of variable names, or logical vector.

A grouping variable can be a numeric vector, logical vector, string (or character array), cell array of strings, or a categorical vector. Rows in `A` that have the same grouping variable values belong to the same group. `varfun` applies `func` to each group of rows within each of the variables of `A`, rather than to each entire variable.

The output, `B`, has one row for each group when you specify `'OutputFormat','uniform'` or `'OutputFormat','cell'`. When you specify `'OutputFormat','table'`, the sizes of the outputs from `func` determine how many rows of `B` correspond to each group. When multiple rows of `B` correspond to a group, `varfun` appends a unique identifier to the row names.

`'OutputFormat'` — Format of `B`

`'table'` (default) | `'uniform'` | `'cell'`

Format of B, specified as the comma-separated pair consisting of 'OutputFormat' and either the string 'uniform', 'table', or 'cell'.

'table'	varfun returns a table with one variable for each variable in A (or each variable specified with 'InputVariables'). For grouped computation, B, also contains the grouping variables. 'table' allows you to use a function that returns values of different sizes or data types for the different variables in A. However, for ungrouped computation, func must return arrays with the same number of rows each time it is called. For grouped computation, func must return values with the same number of rows each time it is called for a given group. This is the default output format.
'uniform'	varfun concatenates the values into a vector. func must return a scalar with the same data type each time it is called.
'cell'	varfun returns B as a cell array. 'cell' allows you to use a function that returns values of different sizes or data types.

'ErrorHandler' — Function to call if func fails

function handle

Function to call if func fails, specified as the comma-separated pair consisting of 'ErrorHandler' and a function handle. Define this function so that it rethrows the error or returns valid outputs for function func.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

identifier	Error identifier.
message	Error message text.
index	Index of the variable for which the error occurred.
name	Name of the variable for which the error occurred.

- The set of input arguments to function func at the time of the error.

For example,

```
function [A, B] = errorFunc(S, varargin)
```

```
warning(S.identifier, S.message)
A = NaN; B = NaN;
```

Output Arguments

B — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see `Table Properties`.

More About

- "Anonymous Functions"

See Also

`arrayfun` | `cellfun` | `rowfun` | `structfun`

vectorize

Vectorize expression

Syntax

```
vectorize(s)
vectorize(f)
```

Description

`vectorize(s)` where `s` is a string expression, inserts a `.` before any `^`, `*` or `/` in `s`. The result is a character string.

Note: `vectorize` will not accept inline function objects (`f`) in a future release.

`vectorize(f)` where `f` is an inline function object, vectorizes the formula for `f`. The result is the vectorized version of the function.

See Also

`inline` | `cd` | `dbtype` | `delete` | `dir` | `path` | `what` | `who`

Introduced before R2006a

ver

Version information for MathWorks products

Syntax

```
ver
ver product

product_info = ver(product)
```

Description

`ver` displays:

- A header containing the current MATLAB product family version number, license number, operating system, and version of Java software for the MATLAB product.
- The version numbers for MATLAB and all other installed MathWorks products.

`ver product` displays, in addition to the header information:

- The current version number for product, where product is the name of the folder that contains the `Contents.m` file for the product you are inquiring about.

`product_info = ver(product)` returns product information to the structure array, product_info.

Examples

Version for All Installed Products

Display version information for all installed products. The output shown here is representative. Your results might differ.

```
ver
```

```
-----------------------------------------------------------------------------------------------------
MATLAB Version: 8.2.0.29 (R2013b)
MATLAB License Number: 234567
```

```
Operating System: Microsoft Windows 7 Version 6.1 (Build 7601: Service Pack 1)
Java Version: Java 1.7.0_11-b21 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode
------------------------------------------------------------------------------------------------------
MATLAB                                                Version 8.2       (R2013b)
Simulink                                              Version 8.2       (R2013b)
Control System Toolbox                                Version 9.6       (R2013b)
```

Version for a Particular Product

Display version information for MATLAB and the Control System Toolbox product. The output shown here is representative. Your results might differ.

1. Determine the product name for Control System Toolbox by setting `n` to the name of a function unique to Control System Toolbox, such as `dss`:

```
n = 'dss';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

```
ans=
control
```

2. Specify the value returned in the previous step as an argument to `ver`:

```
ver control
```

```
------------------------------------------------------------------------------------------------------
MATLAB Version: 8.2.0.29 (R2013b)
MATLAB License Number: 234567
Operating System: Microsoft Windows 7 Version 6.1 (Build 7601: Service Pack 1)
Java Version: Java 1.7.0_11-b21 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode
------------------------------------------------------------------------------------------------------
Control System Toolbox                                Version 9.6       (R2013b)
```

Structure Containing Version for the MATLAB family of products

Create a structure containing version information, and then display the structure values. The output shown here is representative. Your results might differ.

```
v = ver;
for k = 1:length(v)
  fprintf('%s\n', v(k).Name);
  fprintf('  Version: %s\n', v(k).Version);
end
```

```
MATLAB
 Version: 8.2

MATLAB Compiler
```

```
 Version: 5.0

My Custom Toolbox
 Version: 1.0
```

Structure Containing Version for a Particular Product

Create a structure containing version information for just the Symbolic Math Toolbox product. The output shown here is representative. Your results might differ.

1. Determine the product name for Symbolic Math Toolbox by setting `n` to the name of a function unique to Symbolic Math Toolbox, such as `sym`:

```
n = 'sym';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

```
ans =

symbolic
```

2. Specify the value returned in the previous step as an argument to `ver`:

```
v = ver('symbolic')
```

```
v =

        Name: 'Symbolic Math Toolbox'
     Version: '5.11'
     Release: '(R2013b)'
        Date: '19-May-2013'
```

Input Arguments

`product` — product-specific information

string

The product or toolbox for which you want to view version information, specified as a string.

To determine the string to use, run the following code, substituting the name of a product function for *toolbxfcn*:

```
n = 'toolbxfcn';
```

```
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

Output Arguments

`product_info` — product name, version, release, and date
structure array

Product name, version, release, and date, returned as a structure array with these fields: `Name`, `Version`, `Release`, and `Date`. If a license is a trial version, the value in the `Version` field is preceded by the letter `T`

More About

- "Information About Your Installation"
- "Create Help Summary Files — Contents.m"

See Also

`computer` | `help` | `license` | `verlessthan` | `version`

Introduced before R2006a

matlab.unittest.Verbosity class

Package: matlab.unittest

Verbosity level enumeration class

Description

The `matlab.unittest.Verbosity` enumeration class provides a means to specify the level of detail related to running tests. A higher value results in a higher level of detail. The enumeration class contains the following members.

Numeric Representation	Enumeration Member Name	Verbosity Description
1	`Terse`	Minimal amount of information
2	`Concise`	Typical amount of information
3	`Detailed`	Supplemental amount of information
4	`Verbose`	Surplus of information

Construction

`matlab.unittest.Verbosity.`*`MemberName`* creates an instance of the verbosity level enumeration class.

Examples

Create Instance of Enumeration Class

```
n = matlab.unittest.Verbosity.Detailed
```

```
n =

    Detailed
```

Display information about the variables.

```
whos n
```

```
  Name      Size            Bytes  Class                          Attributes

  n         1x1               112  matlab.unittest.Verbosity
```

- “Working with Enumerations”

See Also

`matlab.unittest.TestRunner.withTextOutput` | `matlab.unittest.plugins.LoggingPlugin` | `matlab.unittest.plugins.TestRunProgressPlugin`

Introduced in R2014b

verctrl

(To be removed) Source control actions (Windows platforms)

Note: `verctrl` will be removed in a future release. Access source control actions through the context menu instead.

Syntax

```
verctrl('action',{'filename1','filename2',....},0)
result=verctrl('action',{'filename1','filename2',....},0)
verctrl('action','filename',0)
result=verctrl('isdiff','filename',0)
list = verctrl('all_systems')
```

Description

`verctrl('action',{'filename1','filename2',....},0)` performs the source control operation specified by `'action'` for a single file or multiple files. Enter one file as a string; specify multiple files using a cell array of strings. Use the full paths for each file name and include the extensions. Specify `0` as the last argument. Complete the resulting dialog box to execute the operation. Available values for `'action'` are as follows:

action Argument	Purpose
`'add'`	Adds files to the source control system. Files can be open in the Editor or closed when added.
`'checkin'`	Checks files into the source control system, storing the changes and creating a new version.
`'checkout'`	Retrieves files for editing.
`'get'`	Retrieves files for viewing and compiling, but not editing. When you open the files, they are labeled as read-only.

action Argument	Purpose
`'history'`	Displays the history of files.
`'remove'`	Removes files from the source control system. It does not delete the files from disk, but only from the source control system.
`'runscc'`	Starts the source control system. The file name can be an empty string.
`'uncheckout'`	Cancels a previous checkout operation and restores the contents of the selected files to the precheckout version. All changes made to the files since the checkout are lost.

`result=verctrl('action',{'filename1','filename2',....},0)` performs the source control operation specified by `'action'` on a single file or multiple files. The action can be any one of: `'add'`, `'checkin'`, `'checkout'`, `'get'`, `'history'`, or `'undocheckout'`. `result` is a logical `1` (true) when you complete the operation by clicking **OK** in the resulting dialog box, and is a logical `0` (false) when you abort the operation by clicking **Cancel** in the resulting dialog box.

`verctrl('action','filename',0)` performs the source control operation specified by `'action'` for a single file. Use the absolute path for `'filename'`. Specify `0` as the last argument. Complete any resulting dialog boxes to execute the operation. Available values for `'action'` are as follows:

action Argument	Purpose
`'showdiff'`	Displays the differences between a file and the latest checked in version of the file in the source control system.
`'properties'`	Displays the properties of a file.

`result=verctrl('isdiff','filename',0)` compares `filename` with the latest checked in version of the file in the source control system. `result` is a logical `1` (true) when the files are different, and is a logical `0` (false) when the files are identical. Use the full path for `'filename'`. Specify `0` as the last argument.

`list = verctrl('all_systems')` displays in the Command Window a list of all source control systems installed on your computer.

Examples

Check In a File

Check in `D:\file1.ext` to the source control system:

```
result = verctrl('checkin','D:\file1.ext', 0)
```

This opens the Check in file(s) dialog box. Click **OK** to complete the check in. MATLAB displays

```
 result = 1
```

indicating the checkin was successful.

Add Files to the Source Control System

Add `D:\file1.ext` and `D:\file2.ext` to the source control system.

```
verctrl('add',{'D:\file1.ext','D:\file2.ext'}, 0)
```

This opens the Add to source control dialog box. Click **OK** to complete the operation.

Display the Properties of a File

Display the properties of `D:\file1.ext`.

```
verctrl('properties','D:\file1.ext', 0)
```

This opens the source control properties dialog box for your source control system. The function is complete when you close the properties dialog box.

Show Differences for a File

To show the differences between the version of `file1.ext` that you just edited and saved, with the last version in source control, run

```
verctrl('showdiff','D:\file1.ext',0)
```

MATLAB displays differences dialog boxes and results specific to your source control system. After checking in the file, if you run this statement again, MATLAB displays

```
??? The file is identical to latest version under source control.
```

List All Installed Source Control Systems

To view all of the source control systems installed on your computer, type

```
list = verctrl ('all_systems')
```

MATLAB displays all the source control systems currently installed on your computer. For example:

```
list =
'Microsoft Visual SourceSafe'
'ComponentSoftware RCS'
```

More About

- "MSSCCI Source Control Interface"

Introduced before R2006a

verLessThan

Compare toolbox version to specified version string

Syntax

```
verLessThan(toolbox, version)
```

Description

`verLessThan(toolbox, version)` returns logical 1 (`true`) if the version of the toolbox specified by the string `toolbox` is older than the version specified by the string `version`, and logical 0 (`false`) otherwise. Use this function when you want to write code that can run across multiple versions of the MATLAB software, when there are differences in the behavior of the code in the different versions.

The `toolbox` argument is a string enclosed within single quotation marks that contains the name of a MATLAB toolbox folder. The `version` argument is a string enclosed within single quotation marks that contains the version to compare against. This argument must be in the form `major[.minor[.revision]]`, such as `7`, `7.1`, or `7.0.1`. If `toolbox` does not exist, MATLAB generates an error.

To specify `toolbox`, find the folder that holds the `Contents.m` file for the toolbox and use that folder name. To see a list of all toolbox folder names, enter the following statement in the MATLAB Command Window:

```
dir([matlabroot '/toolbox'])
```

Examples

These examples illustrate usage of the `verLessThan` function.

Example 1 – Checking For the Minimum Required Version

```
if verLessThan('simulink', '4.0')
```

```
    error('Simulink 4.0 or higher is required.');
end
```

Example 2 – Choosing Which Code to Run

```
if verLessThan('matlab', '7.0.1')
% -- Put code to run under MATLAB 7.0.0 and earlier here --
else
% -- Put code to run under MATLAB 7.0.1 and later here --
end
```

Example 3 – Looking Up the Folder Name

Find the name of the Data Acquisition Toolbox folder:

```
dir([matlabroot '/toolbox/d*'])

   daq        database   des      distcomp   dotnetbuilder
   dastudio   datafeed   dials    dml        dspblks
```

Use the toolbox folder name, `daq`, to compare the Data Acquisition Toolbox software version that MATLAB is currently running against version number `3`:

```
verLessThan('daq', '3')
ans =
     1
```

See Also

`ver` | `version` | `license` | `ispc` | `isunix` | `ismac` | `dir`

version

Version number for MATLAB and libraries

Syntax

```
version
version -date
version -description
version -release
version -java
v = version('-versionOption')
[v d] = version
```

Description

`version` returns in `ans` the version and release number for the MATLAB software currently running.

`version -date` returns in `ans` the release date for the MATLAB software.

`version -description` returns in `ans` a description of the version. Usually, the description is for special versions, such as beta versions.

`version -release` returns in `ans` the release number for the MATLAB software currently running.

`version -java` returns in `ans` the version of the Oracle JVM software that MATLAB is using.

`v = version('-versionOption')`, where *`versionOption`* is one of the above option strings, is an alternate form of the syntax.

`[v d] = version` returns the version and release number in string `v` and the release date in string `d`. No input arguments are allowed in this syntax.

Examples

Display MATLAB Version

```
version
```

```
ans =

8.2.0.29 (R2013b)
```

Display MATLAB Release

Display the release, prefaced by a descriptor.

```
['Release R' version('-release')]
```

```
ans =

Release R2013b
```

Get Release Version and Date as Separate Strings

```
[v d] = version
```

```
v =

8.2.0.29 (R2013b)


d =

May 19, 2013
```

Display Java Version

```
version -java
```

```
ans =

Java 1.7.0_11-b21 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode
```

More About

- “Check for Software Updates”

See Also

`computer` | `ver` | `verlessthan`

Introduced before R2006a

vertcat

Concatenate arrays vertically

Syntax

```
C = vertcat(A1,...,AN)
```

Description

`C = vertcat(A1,...,AN)` vertically concatenates arrays `A1,...,AN`. All arrays in the argument list must have the same number of columns.

- If the inputs are multidimensional arrays, `vertcat` concatenates N-dimensional arrays along the first dimension. The remaining dimensions must match.
- If the inputs are tables, `vertcat` concatenates by matching variable names. Variable names for all tables must be identical, except for order. Row names, when present, must be unique across tables.

  `vertcat` fills in default row names when some of the inputs have names and some do not. `vertcat` assigns values for each table property (except for `RowNames`) using the first nonempty value for the corresponding property in the tables `A1,...,AN`.

`vertcat` also concatenates character strings. Each string being concatenated must have the same number of characters.

MATLAB calls `C = vertcat(A1, A2, ...)` for the syntax `C = [A1; A2; ...]` when any of the inputs are an object.

Tips

You can concatenate categorical arrays with cell arrays of strings. For more information, see “Combine Categorical Arrays”.

If all the input arrays are ordinal categorical arrays, they must have the same sets of categories including their order. For more information, see “Ordinal Categorical Arrays”.

You can concatenate datetime arrays with cell arrays of strings.

You can concatenate duration arrays and calendar duration arrays. The result is a calendar duration array.

You can concatenate duration or calendar duration arrays with numeric arrays. Prior to concatenation, MATLAB converts the numeric array to an array of equivalent days using the `days` function.

For information on combining unlike integer types, integers with nonintegers, cell arrays with non-cell arrays, or empty matrices with other elements, see "Valid Combinations of Unlike Classes".

Examples

Vertically Concatenate Two Matrices

Create a 5-by-3 matrix, `A`.

```
A = magic(5);
A(:, 4:5) = []
```

```
A =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25
```

Create a 3-by-3 matrix, `B`.

```
B = magic(3)*100
```

```
B =

   800   100   600
   300   500   700
   400   900   200
```

Vertically concatenate `A` and `B`.

```
C = vertcat(A,B)
```

```
C =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25
   800   100   600
   300   500   700
   400   900   200
```

Vertically Concatenate Two Tables

Create a table, A, with three rows and five variables.

```
A = table([5;6;5],['M';'M';'M'],[45;41;40],[45;32;34],{'NY';'CA';'MA'},...
    'VariableNames',{'Age' 'Gender' 'Height' 'Weight' 'Birthplace'},...
    'RowNames',{'Thomas' 'Gordon' 'Percy'})
```

```
A =

              Age    Gender    Height    Weight    Birthplace
              ---    ------    ------    ------    ----------
    Thomas    5      M         45        45        'NY'
    Gordon    6      M         41        32        'CA'
    Percy     5      M         40        34        'MA'
```

Create a table, B, with the same variables as A except for order.

```
B = table(['F';'M';'F'],[6;6;5],{'AZ';'NH';'CO'},[31;42;33],[39;43;40],...
    'VariableNames',{'Gender' 'Age' 'Birthplace' 'Weight' 'Height'})
```

```
B =

    Gender    Age    Birthplace    Weight    Height
    ------    ---    ----------    ------    ------
    F         6      'AZ'          31        39
    M         6      'NH'          42        43
    F         5      'CO'          33        40
```

Vertically concatenate tables A and B.

```
C = vertcat(A,B)
```

```
C =
```

```
           Age    Gender    Height    Weight    Birthplace
           ---    ------    ------    ------    ----------
Thomas     5      M         45        45        'NY'
Gordon     6      M         41        32        'CA'
Percy      5      M         40        34        'MA'
Row4       6      F         39        31        'AZ'
Row5       6      M         43        42        'NH'
Row6       5      F         40        33        'CO'
```

The variables of `C` are in the same order as the variables of `A` and default row names are used for the rows from `B`.

More About

- "Concatenation Methods"

See Also

`horzcat` | `cat`

Introduced before R2006a

vertcat (tscollection)

Vertical concatenation for `tscollection` objects

Syntax

```
tsc = vertcat(tsc1,tsc2,...)
```

Description

`tsc = vertcat(tsc1,tsc2,...)` performs

```
tsc = [tsc1;tsc2;...]
```

This operation appends `tscollection` objects. The time vectors must not overlap. The last time in tsc1 must be earlier than the first time in tsc2. All `tscollection` objects to be concatenated must have the same `timeseries` members.

See Also

`horzcat (tscollection)` | `tscollection`

Introduced before R2006a

vertexAttachments

Class: TriRep

(Will be removed) Return simplices attached to specified vertices

---

Note: `vertexAttachments(TriRep)` will be removed in a future release. Use `vertexAttachments(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
SI = vertexAttachments(TR, VI)
```

Description

`SI = vertexAttachments(TR, VI)` returns the vertex-to-simplex information for the specified vertices `VI`. For 2-D triangulations in MATLAB, the triangles `SI` are arranged in counter-clockwise order around the attached vertex `VI`.

Input Arguments

`TR`	Triangulation representation
`VI`	`VI` is a column vector of indices into the array of points representing the vertex coordinates, `TR.X`. The simplices associated with vertex `i` are the `i`'th entry in the cell array. If `VI` is not specified the vertex-simplex information for the entire triangulation is returned.

Output Arguments

`SI`	Cell array of indices of the simplices attached to a vertex. A cell array is used to store the information because the number of simplices

associated with each vertex can vary. The simplices associated with vertex `i` are in the `i`'th entry in the cell array `SI`.

Definitions

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to compute the vertex-to-triangle relations.

```
load trimesh2d
trep = TriRep(tri, x, y);
```

Find the indices of the tetrahedra attached to the first vertex:

```
Tv = vertexAttachments(trep, 1)
Tv{:}
```

Example 2

Perform a direct query of a 2-D triangulation created using `DelaunayTri`.

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
```

Find the triangles attached to vertex 5:

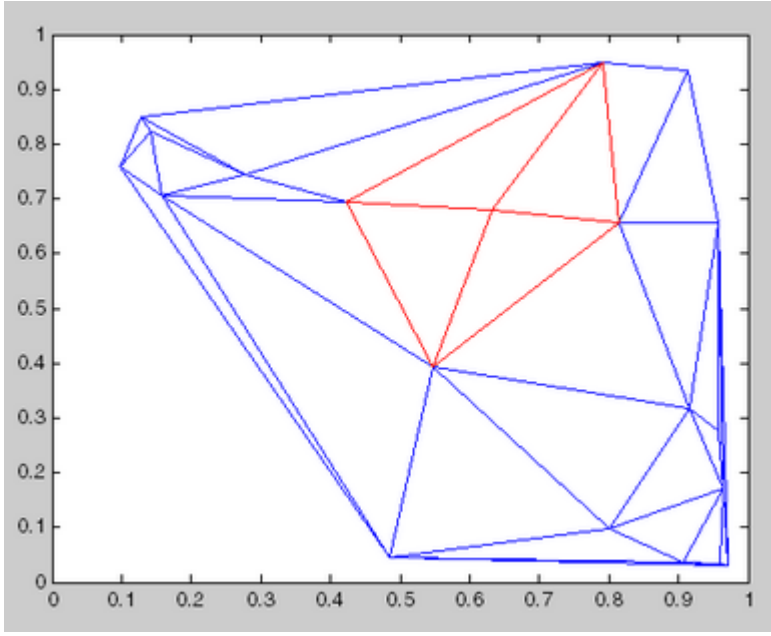
```
t = vertexAttachments(dt,5);
```

Plot the triangulation:

```
triplot(dt);
hold on;
```

Plot the triangles attached to vertex 5 (in red):

```
triplot(dt(t{:},:),x,y,'Color','r');
hold off;
```



See Also

`triangulation` | `delaunayTriangulation`

VideoReader class

Read video files

Description

Use the `VideoReader` function with the `read` method to read video data from a file into the MATLAB workspace.

The file formats that `VideoReader` supports vary by platform, as follows (with no restrictions on file extensions):

All Platforms	AVI, including uncompressed, indexed, grayscale, and Motion JPEG-encoded video (`.avi`) Motion JPEG 2000 (`.mj2`)
All Windows	MPEG-1 (`.mpg`) Windows Media Video (`.wmv`, `.asf`, `.asx`) Any format supported by Microsoft DirectShow®
Windows 7 or later	MPEG-4, including H.264 encoded video (`.mp4`, `.m4v`) Apple QuickTime Movie (`.mov`) Any format supported by Microsoft Media Foundation
Macintosh	Most formats supported by QuickTime Player, including: MPEG-1 (`.mpg`) MPEG-4, including H.264 encoded video (`.mp4`, `.m4v`) Apple QuickTime Movie (`.mov`) 3GPP 3GPP2 AVCHD DV
Linux	Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on http://gstreamer.freedesktop.org/documentation/plugins.html, including Ogg Theora (`.ogg`).

Construction

`obj = VideoReader(filename)` constructs obj to read video data from the file named filename. If it cannot construct the object for any reason, `VideoReader` generates an error.

`obj = VideoReader(filename,Name,Value)` constructs the object with additional options specified by one or more Name,Value pair arguments. `Name` is `'Tag'` or `'UserData'` and `Value` is the corresponding value. You can specify two name and value pair arguments in any order as `Name1,Value1,Name2,Value2`.

Input Arguments

`filename`

String in single quotation marks that specifies the video file to read. The `VideoReader` constructor searches for the file on the MATLAB path.

Name-Value Pair Arguments

Optional comma-separated pairs of Name,Value arguments, where `Name` is `'Tag'` or `'UserData'` and `Value` is the corresponding value. You can specify two name and value pair arguments in any order as `Name1,Value1,Name2,Value2`.

`'Tag'`

String that identifies the object.

Default: `''`

`'UserData'`

Generic field for data of any class that you want to add to the object.

Default: `[]`

Properties

All properties are read-only except `CurrentTime`, `Tag` and `UserData`.

`BitsPerPixel`

Bits per pixel of the video data.

CurrentTime

Timestamp of the video frame to read, specified in seconds from the start of the video file. `CurrentTime` can be a `double` value between zero and the duration of the video.

Duration

Total length of the file in seconds.

FrameRate

Frame rate of the video in frames per second.

Height

Height of the video frame in pixels.

Name

Name of the file associated with the object.

NumberOfFrames

Note: `NumberOfFrames` will be removed in a future release.

Total number of frames in the video stream.

Some files store video at a variable frame rate, including many Windows Media Video files. For these files, `VideoReader` cannot determine the number of frames until you read the last frame. When you construct the object, `VideoReader` returns a warning and does not set the `NumberOfFrames` property.

To count the number of frames in a variable frame rate file, use the `read` method to read the last frame of the file. For example:

```
vidObj = VideoReader('varFrameRateFile.wmv');
lastFrame = read(vidObj, inf);
numFrames = vidObj.NumberOfFrames;
```

Path

String containing the full path to the file associated with the reader.

Tag

String that identifies the object.

Default: `''`

Type

Class name of the object: `'VideoReader'`.

UserData

Generic field for data of any class that you want to add to the object.

Default: `[]`

VideoFormat

String indicating the MATLAB representation of the video format.

Video Format	Value of `VideoFormat`
AVI or MPEG-4 files with RGB24 video	`'RGB24'`
AVI files with indexed video	`'Indexed'`
AVI files with grayscale video	`'Grayscale'`

For Motion JPEG 2000 files, `VideoFormat` is one of the following.

Format of Image Data	Value of `VideoFormat`
Single-band `uint8`	`'Mono8'`
Single-band `int8`	`'Mono8 Signed'`
Single-band `uint16`	`'Mono16'`
Single-band `int16`	`'Mono16 Signed'`
Three-banded `uint8`	`'RGB24'`
Three-banded `int8`	`'RGB24 Signed'`
Three-banded `uint16`	`'RGB48'`
Three-banded `int16`	`'RGB48 Signed'`

`Width`

Width of the video frame in pixels.

Methods

get	Query property values for video reader object
getFileFormats	File formats that `VideoReader` supports
hasFrame	Determine if frame available to read
read	Read video frame data from file
readFrame	Read video frame from video file
set	Set property values for video reader object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct `VideoReader` Object

Construct a `VideoReader` object for the example movie file `xylophone.mp4` and view its properties.

```
xyloObj = VideoReader('xylophone.mp4');
get(xyloObj)
```

```
obj =
```

```
  VideoReader with properties:

   General Properties:
           Name: 'xylophone.mp4'
           Path: 'matlabroot\toolbox\matlab\audiovideo'
       Duration: 4.7000
    CurrentTime: 0
            Tag: ''
       UserData: []

   Video Properties:
          Width: 320
         Height: 240
       FrameRate: 30
    BitsPerPixel: 24
     VideoFormat: 'RGB24'
```

Read and Play Back Movie File

Read and play back the sample movie file, `xylophone.mp4`.

Construct a `VideoReader` object to read data from the sample file. Then, determine the width and height of the video.

```
xyloObj = VideoReader('xylophone.mp4');

vidWidth = xyloObj.Width;
vidHeight = xyloObj.Height;
```

Create a movie structure array, `mov`.

```
mov = struct('cdata',zeros(vidHeight,vidWidth,3,'uint8'),...
    'colormap',[]);
```

Read one frame at a time until the end of the video is reached.

```
k = 1;
while hasFrame(xyloObj)
    mov(k).cdata = readFrame(xyloObj);
    k = k+1;
end
```

Size a figure based on the video's width and height. Then, play back the movie once at the video's frame rate.

```
hf = figure;
set(hf,'position',[150 150 vidWidth vidHeight]);

movie(hf,mov,1,xyloObj.FrameRate);
```

See Also

`mmfileinfo` | `VideoWriter`

How To

- “Read Video Files”

VideoWriter class

Write video files

Description

Use the `VideoWriter` object with the `open`, `writeVideo`, and `close` methods to create video files from figures, still images, or MATLAB movies. `VideoWriter` can create uncompressed AVI and Motion JPEG 2000 compressed AVI files on all platforms, and MPEG-4 files on Windows 7 or later and Mac OS X 10.7 and higher. `VideoWriter` supports files larger than 2 GB, which is the limit for `avifile`.

To set video properties, `VideoWriter` includes predefined *profiles* such as `'Uncompressed AVI'` or `'MPEG-4'`.

Construction

`writerObj = VideoWriter(filename)` constructs a `VideoWriter` object to write video data to an AVI file with Motion JPEG compression.

`writerObj = VideoWriter(filename,profile)` applies a set of properties tailored to a specific file format (such as `'MPEG-4'` or `'Uncompressed AVI'`) to a `VideoWriter` object.

Input Arguments

`filename`

String enclosed in single quotation marks that specifies the name of the file to create.

`VideoWriter` supports these file extensions:

`.avi`	AVI file
`.mj2`	Motion JPEG 2000 file
`.mp4` or `.m4v`	MPEG-4 file (systems with Windows 7 or later, or Mac OS X 10.7 and later)

If you do not specify a valid file extension, `VideoWriter` appends the extension `.avi`, `.mj2` or `.mp4`, depending on the `profile`. If you do not specify a value for `profile`, then `VideoWriter` creates a Motion JPEG compressed AVI file with the extension `.avi`.

`profile`

String enclosed in single quotation marks that describes the type of file to create. Specifying a profile sets default values for video properties such as `VideoCompressionMethod`. Possible values:

`'Archival'`	Motion JPEG 2000 file with lossless compression
`'Motion JPEG AVI'`	Compressed AVI file using Motion JPEG codec
`'Motion JPEG 2000'`	Compressed Motion JPEG 2000 file
`'MPEG-4'`	Compressed MPEG-4 file with H.264 encoding (systems with Windows 7 or later, or Mac OS X 10.7 and later)
`'Uncompressed AVI'`	Uncompressed AVI file with `RGB24` video
`'Indexed AVI'`	Uncompressed AVI file with indexed video
`'Grayscale AVI'`	Uncompressed AVI file with grayscale video

Default: `'Motion JPEG AVI'`

Properties

`ColorChannels`

Number of color channels in each output video frame. (Read-only)

AVI and MPEG-4 files with RGB24 data have three color channels. Indexed and Grayscale AVI files have one color channel. The number of channels for Motion JPEG 2000 files depends on the input data to the `writeVideo` method: one for monochrome image data, three for color data.

`Colormap`

`P`-by-3 numeric matrix that contains color information about the video file. The colormap can have a maximum of 256 entries of type `uint8` or `double`. The entries of the colormap must be integers. Each row of `Colormap` specifies the red, green, and blue

components of a single color. The colormap can be set explicitly before the call to `open`, or using the `colormap` field of a movie frame structure at the time of writing the first frame.

Only applies to objects associated with Indexed AVI files.

After you call `open`, you cannot change the `Colormap` value.

Default: none

`CompressionRatio`

Number greater than 1 that specifies the target ratio between the number of bytes in the input image and the number of bytes in the compressed image. The data is compressed as much as possible, up to the specified target.

Only available for objects associated with Motion JPEG 2000 files. After you call `open`, you cannot change the `CompressionRatio` value. If you previously set `LosslessCompression` to `true`, setting `CompressionRatio` generates an error.

Default: `10`

`Duration`

Scalar value specifying the duration of the file in seconds. (Read-only)

`FileFormat`

String specifying the type of file to write: `'avi'`, `'mp4'`, or `'mj2'`. (Read-only)

`Filename`

String specifying the name of the file. (Read-only)

`FrameCount`

Number of frames written to the video file. (Read-only)

`FrameRate`

Rate of playback for the video in frames per second. After you call `open`, you cannot change the `FrameRate` value.

Default: 30

`Height`

Height of each video frame in pixels. The `writeVideo` method sets values for `Height` and `Width` based on the dimensions of the first frame. (Read-only)

MPEG-4 files require frame dimensions that are divisible by two. If the input frame height for an MPEG-4 file is not an even number, `VideoWriter` pads the frame with a row of black pixels at the bottom.

`LosslessCompression`

Boolean value (logical `true` or `false`) only available for objects associated with Motion JPEG 2000 files. If `true`:

- The `writeVideo` method uses reversible mode so that the decompressed data is identical to the input data.
- `VideoWriter` ignores any specified value for `CompressionRatio`.

After you call `open`, you cannot change the `LosslessCompression` value.

Default: `false` for the `'Motion JPEG 2000'` profile, `true` for the `'Archival'` profile

`MJ2BitDepth`

Number of least significant bits in the input image data, from 1 to 16.

Only available for objects associated with Motion JPEG 2000 files. If you do not specify a value before calling the `open` method, `VideoWriter` sets the bit depth based on the input data type. For example, if the input data to `writeVideo` is an array of `uint8` or `int8` values, `MJ2BitDepth` is `8`.

`Path`

String specifying the fully qualified path. (Read-only)

`Quality`

Integer from 0 through 100. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes.

Only available for objects associated with the `MPEG-4` or `Motion JPEG AVI` profile. After you call `open`, you cannot change the `Quality` value.

Default: 75

`VideoBitsPerPixel`

Number of bits per pixel in each output video frame. (Read-only)

AVI files with truecolor video and MPEG-4 files have 24 bits per pixel (8 bits for each of three color bands).

For Motion JPEG 2000 files, the number of bits per pixel depends on the value of `MJ2BitDepth` and the number of bands of image data. For example, if the input data to `writeVideo` is a three-dimensional array of `uint16` or `int16` values, the default value of `MJ2BitDepth` is `16`, and `VideoBitsPerPixel` is `48` (three times the bit depth).

`VideoCompressionMethod`

String indicating the type of video compression: `'None'`, `'H.264'`, `'Motion JPEG'`, or `'Motion JPEG 2000'`. (Read-only)

`VideoFormat`

String indicating the MATLAB representation of the video format. (Read-only)

Video Format	Value of `VideoFormat`
AVI or MPEG-4 files with RGB24 video	`'RGB24'`
AVI files with indexed video	`'Indexed'`
AVI files with grayscale video	`'Grayscale'`

For Motion JPEG 2000 files, `VideoFormat` depends on the value of `MJ2BitDepth` and the format of the input image data to the `writeVideo` method. For example, if you do not specify the `MJ2BitDepth` property, `VideoWriter` sets the format as shown in this table.

Format of Image Data	Value of `VideoFormat`
Single-band `uint8`	`'Mono8'`
Single-band `int8`	`'Mono8 Signed'`

Format of Image Data	Value of `VideoFormat`
Single-band `uint16`	`'Mono16'`
Single-band `int16`	`'Mono16 Signed'`
Three-banded `uint8`	`'RGB24'`
Three-banded `int8`	`'RGB24 Signed'`
Three-banded `uint16`	`'RGB48'`
Three-banded `int16`	`'RGB48 Signed'`

`Width`

Width of each video frame in pixels. The `writeVideo` method sets values for `Height` and `Width` based on the dimensions of the first frame. (Read-only)

MPEG-4 files require frame dimensions that are divisible by two. If the input frame width for an MPEG-4 file is not an even number, `VideoWriter` pads the frame with a column of black pixels along the right side.

Methods

close	Close file after writing video data
getProfiles	List profiles and file formats supported by `VideoWriter`
open	Open file for writing video data
writeVideo	Write video data to file

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

View and Set Object Properties

Construct a `VideoWriter` object and view its properties.

```
writerObj = VideoWriter('newfile.avi')
```

Set the frame rate to 60 frames per second, using the `FrameRate` property.

```
writerObj.FrameRate = 60;
```

AVI File from Animation

Write a sequence of frames to a compressed AVI file, `peaks.avi`.

Prepare the new file.

```
writerObj = VideoWriter('peaks.avi');
open(writerObj);
```

Generate initial data and set axes and figure properties.

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
set(gcf,'Renderer','zbuffer');
```

Setting the `Renderer` property to `zbuffer` or `Painters` works around limitations of `getframe` with the OpenGL renderer on some Windows systems.

Create a set of frames and write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe;
   writeVideo(writerObj,frame);
end

close(writerObj);
```

See Also

`VideoReader` | `mmfileinfo`

view

Viewpoint specification

Syntax

```
view(az,el)
view([az,el])
view([x,y,z])
view(2)
view(3)
view(ax,...)
[az,el] = view
```

Description

The position of the viewer (the viewpoint) determines the orientation of the axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,el)` and `view([az,el])` set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the *z*-axis as measured in degrees from the negative *y*-axis. Positive values indicate counterclockwise rotation of the viewpoint. `el` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the view direction to the Cartesian coordinates `x`, `y`, and `z`. The magnitude of `(x,y,z)` is ignored.

`view(2)` sets the default two-dimensional view, `az = 0`, `el = 90`.

`view(3)` sets the default three-dimensional view, `az = –37.5`, `el = 30`.

`view(ax,...)` uses axes `ax` instead of the current axes.

`[az,el] = view` returns the current azimuth and elevation.

Examples

View the object from directly overhead.

```
az = 0;
el = 90;
view(az, el);
```

Set the view along the *y*-axis, with the *x*-axis extending horizontally and the *z*-axis extending vertically in the figure.

```
view([0 0]);
```

Rotate the view about the *z*-axis by 180°.

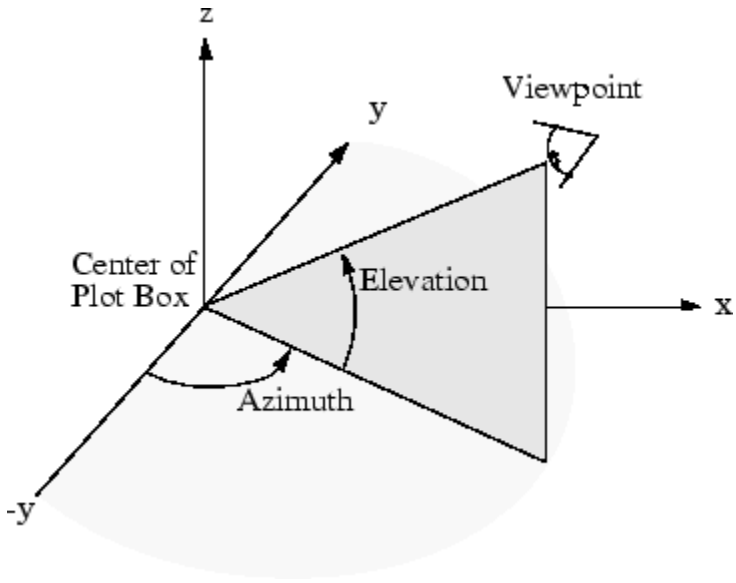
```
az = 180;
el = 90;
view(az, el);
```

More About

Tips

Azimuth is a polar angle in the *x*-*y* plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the *x*-*y* plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.



For more information, see `CameraPosition`, `CameraTarget`, `CameraViewAngle`, and `Projection`.

- "View Overview"

See Also

`hgtransform` | `rotate3d`

Introduced before R2006a

viewmtx

View transformation matrices

Syntax

```
viewmtx
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,phi,xc)
```

Description

`viewmtx` computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

`T = viewmtx(az,el)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `el`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `el` is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,el)
T = view
```

but does not change the current view.

`T = viewmtx(az,el,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

Phi	Description
0 degrees	Orthographic projection
10 degrees	Similar to telephoto lens
25 degrees	Similar to normal lens
60 degrees	Similar to wide-angle lens

`T = viewmtx(az,el,phi,xc)` returns the perspective transformation matrix using `xc` as the target point within the normalized plot cube (i.e., the camera is looking at the point `xc`). `xc` is the target point that is the center of the view. You specify the point as a three-element vector, `xc = [xc,yc,zc]`, in the interval [0,1]. The default value is `xc = [0,0,0]`.

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, `[x,y,z,1]` is the four-dimensional vector corresponding to the three-dimensional point `[x,y,z]`.

Examples

Determine the projected two-dimensional vector corresponding to the three-dimensional point (0.5,0.0,-3.0) using the default view direction. Note that the point is a column vector.

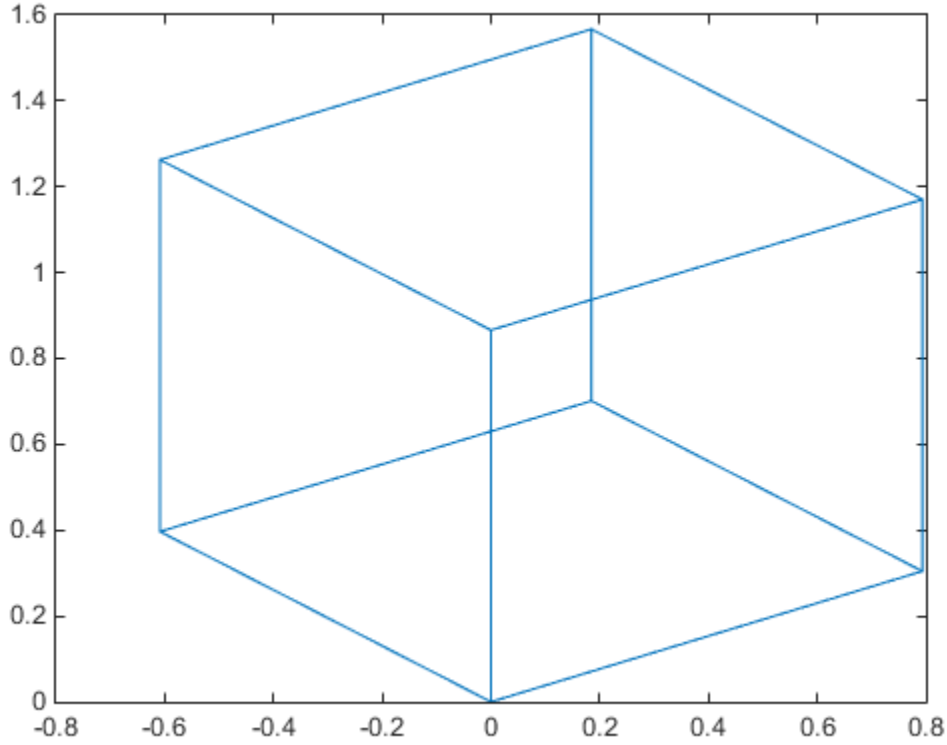
```
A = viewmtx(-37.5,30);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)
x2d =
   0.3967
   -2.4459
```

These vectors trace the edges of a unit cube:

```
x = [0  1  1  0  0  0  1  1  0  0  1  1  1  1  0  0];
y = [0  0  1  1  0  0  0  1  1  0  0  0  1  1  1  1];
z = [0  0  0  0  0  1  1  1  1  1  1  0  0  1  1  0];
```

Transform the points in these vectors to the screen, then plot the object.

```
A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
figure
plot(x2,y2)
```



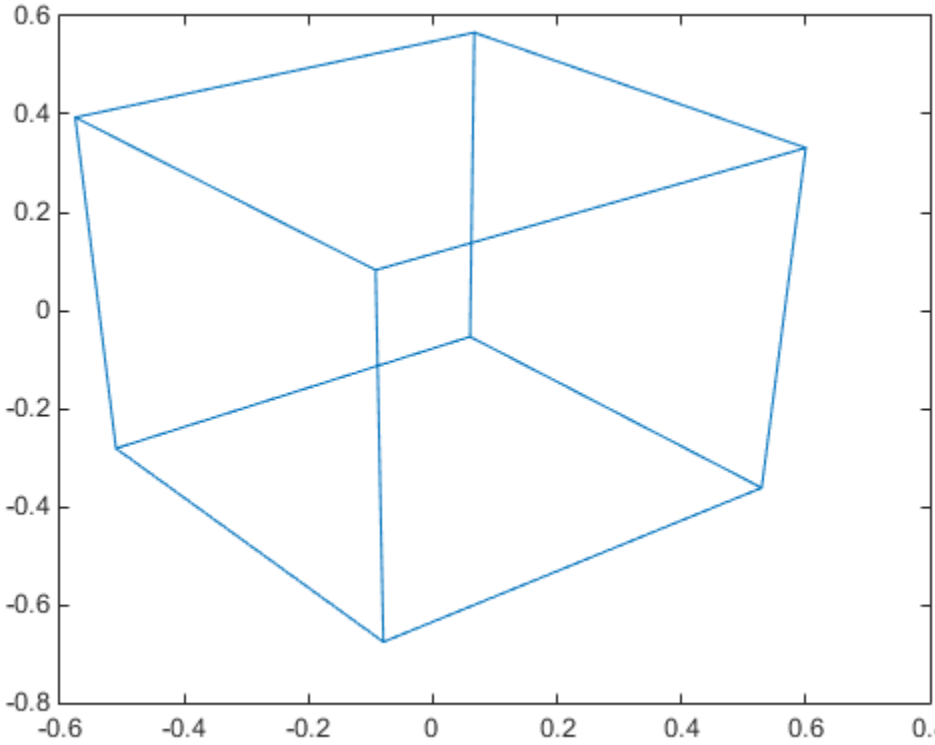
Use a perspective transformation with a 25 degree viewing angle:

```
A = viewmtx(-37.5,30,25);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4)
x2d =
   0.1777
   -1.8858
```

Transform the cube vectors to the screen and plot the object:

```
A = viewmtx(-37.5,30,25);
[m,n] = size(x);
```

```
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)./x2d(4,:);
y2(:) = x2d(2,:)./x2d(4,:);
figure
plot(x2,y2)
```



See Also

`view` | `hgtransform`

Introduced before R2006a

visdiff

Compare two text files, MAT-Files, binary files, Zip files, or folders

Syntax

```
visdiff('filename1', 'filename2')
visdiff('filename1', 'filename2', 'type')
```

Description

`visdiff('filename1', 'filename2')` opens the Comparison Tool and presents the differences between the two files or folders. Either ensure that the two files or folders appear on the MATLAB path, or provide the full path for each file or folder. You can compare files or any combination of folders, zip files, Simulink manifests or models.

If you have Simulink Report Generator™ software, you can select a pair of Simulink models to compare XML text files generated from them. For information, see "Model Comparison".

`visdiff('filename1', 'filename2', 'type')` opens the Comparison Tool and presents the differences between the two files using the specified comparison type. *`type`* can be `'text'` or `'binary'`. If you do not specify `type`, `visdiff` creates the default comparison type for your selected files. The *`type`* option does not apply when comparing folders.

If there are multiple comparison types available for your selections (e.g., text, binary, file list, or XML comparison), you can create a new comparison and choose a different comparison type.

1. In the Comparison Tool, click the New comparison button.

   The dialog box Select Files or Folders for Comparison opens and shows your previous comparison selections in the drop-down lists.

2. Change the comparison type and click **Compare**.

Examples

Specifying Files or Folders to Compare

The `visdiff` function accepts fully qualified file names, relative file names, or names of files on the MATLAB path.

If the files you want to compare appear on the MATLAB path or in the current folder, you can specify the file names without the full path, for example:

```
visdiff('lengthofline.m','lengthofline2.m')
```
or

```
visdiff('lengthofline','lengthofline2')
```

If the files you want to compare are not on the path, either specify the full path to each file, or add the folders to the path.

For example, to specify the fully qualified file names to compare two example files:

```
visdiff(fullfile(matlabroot,'toolbox','matlab','demos','gatlin.mat'), ...
fullfile(matlabroot,'toolbox','matlab','demos','gatlin2.mat'))
```

Specify the full path to files as follows:

```
visdiff('C:\Work\comp\lengthofline.m', 'C:\Work\comp\lengthofline2.m')
```

You can specify paths to files relative to the current folder. For the preceding example, if the current folder is `Work`, then the relative paths are:

```
visdiff('comp\lengthofline.m', 'comp\lengthofline2.m')
```

Compare Two Text Files

To view a comparison of the two example files, `lengthofline.m` and `lengthofline2.m`:

```
visdiff(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','lengthofline.m'), fullfile(matlabroot,'help',...
'techdoc','matlab_env','examples','lengthofline2.m'))
```

For information about using the report features, see “Comparing Text Files”.

Note: If the text files you compare are XML files, you see different results if you have MATLAB Report Generator installed. For details, see “Comparing Files and Folders”.

Compare Two MAT-Files

To compare two example files:

```
visdiff(fullfile(matlabroot,'toolbox','matlab','demos','gatlin.mat'), ...
fullfile(matlabroot,'toolbox','matlab','demos','gatlin2.mat'))
```

For information about the report features, see “Comparing MAT-Files”.

Compare Two Binary Files

The following example code adds a folder containing two MEX-files to the MATLAB path, and then compares the files:

```
addpath([matlabroot '\extern\examples\shrlib'])
visdiff('shrlibsample.mexw32', 'yprime.mexw32')
```

The Comparison Tool opens and indicates that the files are different, but does not provide details about the differences.

For more information on binary comparisons, see “Comparing Binary Files”.

Compare Two Folders or Zip Files

You can perform file list comparisons for any combinations of folders and zip files. To view an example folder comparison and instructions for using the report features, see “Comparing Folders and Zip Files”.

Compare Files and Specify Type

To compare two example text files and specify comparison type as binary:

```
visdiff(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','lengthofline.m'), fullfile(matlabroot,'help',...
'techdoc','matlab_env','examples','lengthofline2.m'), 'binary')
```

If you do not specify type, `visdiff` creates the default comparison type for your selected files, in this case, text comparison. By changing to the binary comparison type you could examine differences such as end-of-line characters.

Similarly, when you compare XML files without specifying type, you get a hierarchical XML comparison report. If instead you want a text or binary comparison, you can specify "text" or "binary" comparison types to see more details. When you compare zip files, the default comparison type is a file list comparison, and you might want to specify a binary comparison instead.

Alternatives

As an alternative to the `visdiff` function, compare files and folders using any of these interactive methods:

- From the Current Folder browser:
  - Select a file or folder. Right-click the file or folder, and select **Compare Against**.
  - For two files or subfolders in the same folder, select the files or folders. Then, right-click, and select **Compare Selected Files/Folders**.
- From the MATLAB desktop, on the **Home** tab, in the **File** section, click **Compare** and then select the files or folders to compare.
- If you have a file open in the Editor, on the **Editor** tab, in the **File** section, click **Compare**. Alternatively, under **Compare**, you can choose a file to compare against, or compare with the autosave version or the version on disk. See “Comparing Files with Autosave Version or Version on Disk”.

More About

- “Comparing Files and Folders”
- “Model Comparison”

volumebounds

Coordinate and color limits for volume data

Syntax

```
lims = volumebounds(X,Y,Z,V)
lims = volumebounds(X,Y,Z,U,V,W)
lims = volumebounds(V)
lims = volumebounds(U,V,W)
```

Description

`lims = volumebounds(X,Y,Z,V)` returns the x, y, z, and color limits of the current axes for scalar volume data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax cmin cmax]
```

You can pass this vector to the `axis` command.

`lims = volumebounds(X,Y,Z,U,V,W)` returns the x, y, and z limits of the current axes for vector volume data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax]
```

`lims = volumebounds(V)` and `lims = volumebounds(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(V)`.

Examples

This example uses `volumebounds` to set the axis and color limits for an `isosurface` generated by the `flow` function.

```
[x y z v] = flow;
```

```
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
daspect([1 1 1])
isocolors(x,y,z,fliplr(v),p)
shading interp
axis(volumebounds(x,y,z,v))
```

See Also

`isosurface` | `streamslice`

Introduced before R2006a

voronoi

Voronoi diagram

Note: Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `voronoi`.

Note: The behavior of `h = voronoi(...)` has changed. The new behavior returns a vector of two chart line handles; one representing the points and the other representing the voronoi edges.

Syntax

```
voronoi(x,y)
voronoi(x,y,TRI)
voronoi(dt)
voronoi(AX,...)
voronoi(...,'LineSpec')
h = voronoi(...)
[vx,vy] = voronoi(...)
```

Description

`voronoi(x,y)` plots the bounded cells of the Voronoi diagram for the points `x`,`y`. Lines-to-infinity are approximated with an arbitrarily distant endpoint.

`voronoi(x,y,TRI)` uses the triangulation `TRI` instead of computing internally.

`voronoi(dt)` uses the Delaunay triangulation `dt` instead of computing it.

`voronoi(AX,...)` plots into `AX` instead of `gca`.

`voronoi(...,'LineSpec')` plots the diagram with color and line style specified.

`h = voronoi(...)` returns `h`, which is a vector of two chart line handles. One represents the points and the other represents the voronoi edges.

`[vx,vy] = voronoi(...)` returns the finite vertices of the Voronoi edges in `vx` and `vy`.

Note For the topology of the Voronoi diagram, i.e., the vertices for each Voronoi cell, use `voronoin`.

```
[v,c] = voronoin([x(:) y(:)])
```

Definitions

Consider a set of coplanar points *P*. For each point P_x in the set *P*, you can draw a boundary enclosing all the intermediate points lying closer to P_x than to other points in the set *P*. Such a boundary is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

Visualization

Use one of these methods to plot a Voronoi diagram:

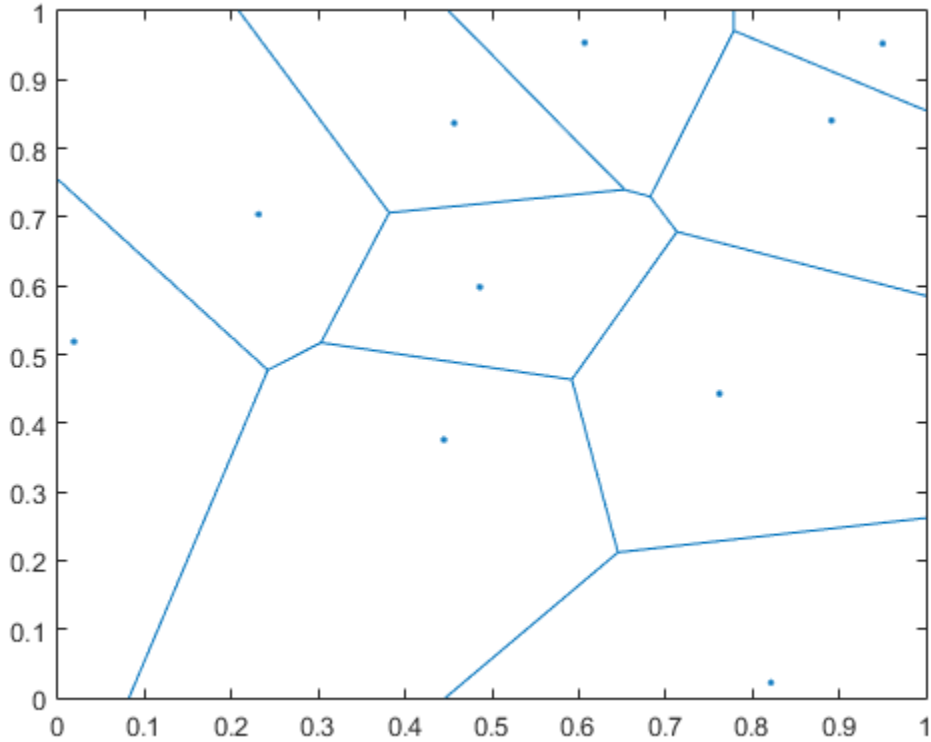
- If you provide no output argument, `voronoi` plots the diagram.
- To gain more control over color, line style, and other figure properties, use the syntax `[vx,vy] = voronoi(...)`. This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the `plot` function.
- To fill the cells with color, use `voronoin` with `n = 2` to get the indices of each cell, and then use `patch` and other plot functions to generate the figure. Note that `patch` does not fill unbounded cells with color.

Examples

Voronoi Diagram Based on Points

This code uses the `voronoi` function to plot the Voronoi diagram for 10 randomly generated points.

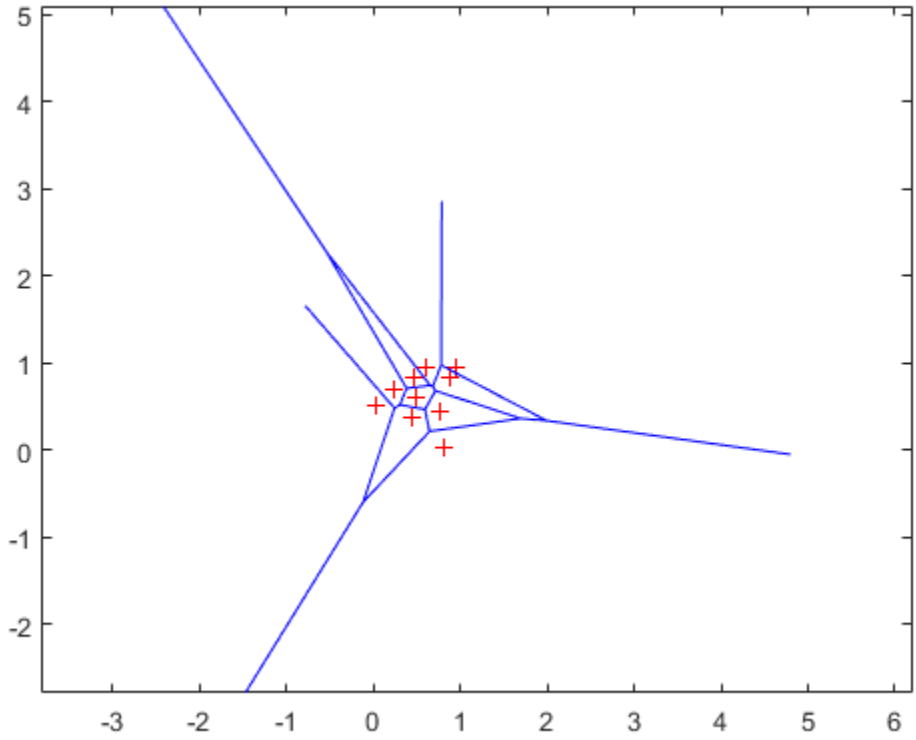
```
x = gallery('uniformdata',[1 10],0);
y = gallery('uniformdata',[1 10],1);
voronoi(x,y)
```



Voronoi Diagram Based on Vertices of Voronoi Edges

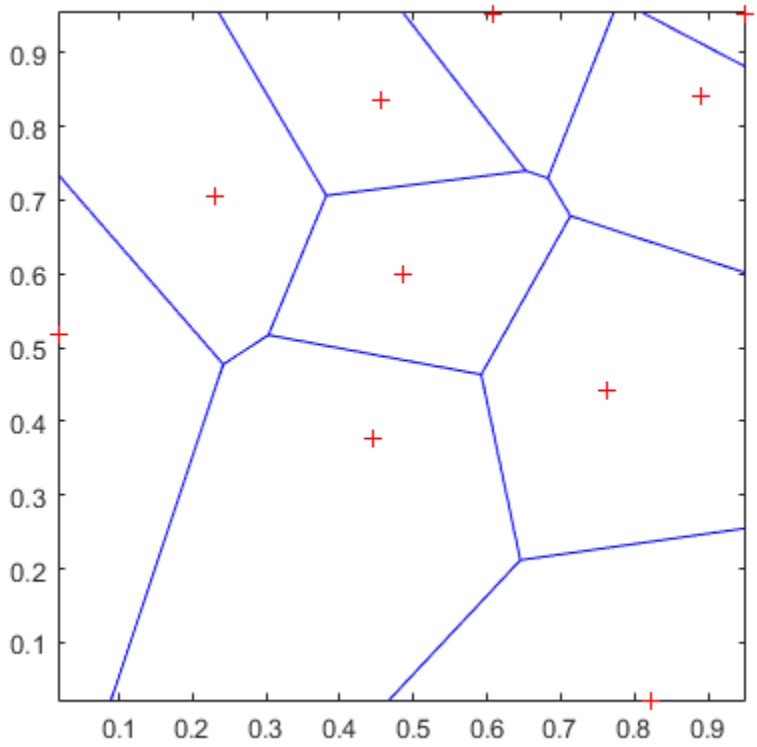
This code uses the vertices of the finite Voronoi edges to plot the Voronoi diagram for the same 10 points used in the previous example.

```
x = gallery('uniformdata',[1 10],0);
y = gallery('uniformdata',[1 10],1);
[vx,vy] = voronoi(x,y);
plot(x,y,'r+',vx,vy,'b-')
axis equal
```



Note that you can add the following code to get the figure shown in the previous example.

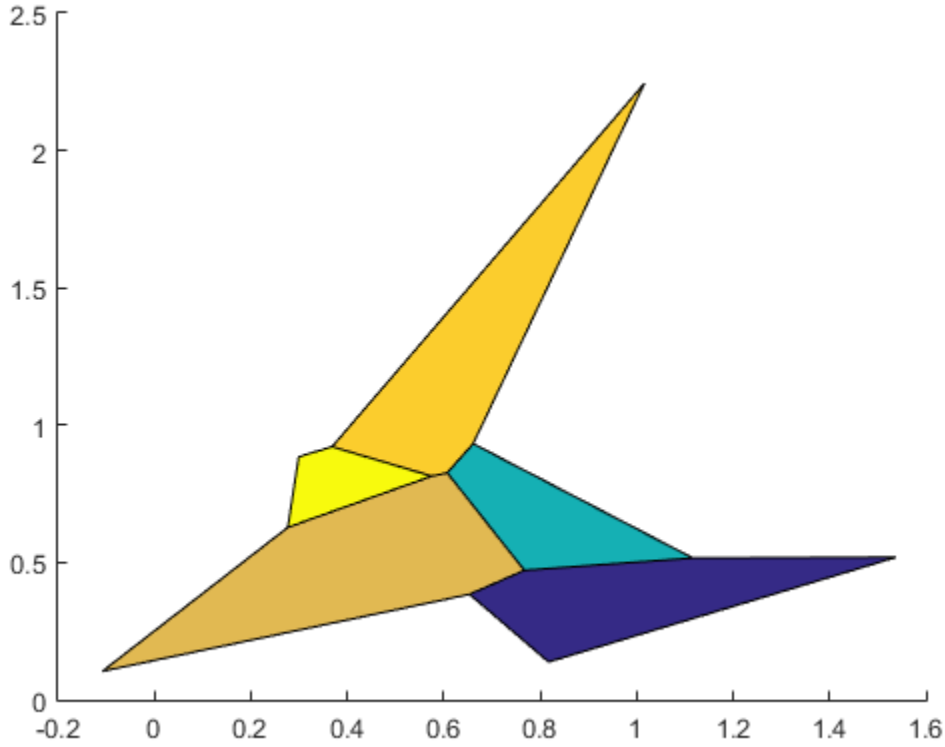
```
xlim([min(x) max(x)])
ylim([min(y) max(y)])
```



Voronoi Diagram with Color

This code uses `voronoin` and `patch` to fill the bounded cells of the same Voronoi diagram with color.

```
x = gallery('uniformdata',[10 2],5);
[v,c] = voronoin(x);
for i = 1:length(c)
if all(c{i}~=1)   % If at least one of the indices is 1,
                  % then it is an open region and we can't
                  % patch that.
patch(v(c{i},1),v(c{i},2),i); % use color i.
end
end
```



See Also

`delaunayTriangulation` | `convhull` | `delaunay` | `LineSpec` | `plot` | `voronoin`

Introduced before R2006a

voronoiDiagram

Class: DelaunayTri

(Will be removed) Voronoi diagram

Note: `voronoiDiagram(DelaunayTri)` will be removed in a future release. Use `voronoiDiagram(delaunayTriangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
[V, R] = voronoiDiagram(DT)
```

Description

`[V, R] = voronoiDiagram(DT)` returns the vertices `V` and regions `R` of the Voronoi diagram of the points `DT.X`. The region `R{i}` is a cell array of indices into `V` that represents the Voronoi vertices bounding the region. The Voronoi region associated with the `i`'th point, `DT.X(i)` is `R{i}`. For 2-D, vertices in `R{i}` are listed in adjacent order, i.e. connecting them will generate a closed polygon (Voronoi diagram). For 3-D the vertices in `R{i}` are listed in ascending order.

The Voronoi regions associated with points that lie on the convex hull of `DT.X` are unbounded. Bounding edges of these regions radiate to infinity. The vertex at infinity is represented by the first vertex in `V`.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

`V`	`numv`-by-`ndim` matrix representing the coordinates of the Voronoi vertices, where `numv` is the number of vertices and `ndim` is the dimension of the space where the points reside.
`R`	Vector cell array of `length(DR.X)`, representing the Voronoi cell associated with each point.

Definitions

The *Voronoi diagram* of a discrete set of points `X` decomposes the space around each point `X(i)` into a region of influence `R{i}`. Locations within the region are closer to point `i` than any other point. The region of influence is called the Voronoi region. The collection of all the Voronoi regions is the Voronoi diagram.

The *convex hull* of a set of points `X` is the smallest convex polygon (or polyhedron in higher dimensions) containing all of the points of `X`.

Examples

Compute the Voronoi Diagram of a set of points:

```
X = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ]
dt = DelaunayTri(X)
[V,R] = voronoiDiagram(dt)
```

See Also

`voronoi` | `delaunayTriangulation` | `voronoin` | `triangulation`

voronoin

N-D Voronoi diagram

Syntax

```
[V,C] = voronoin(X)
[V,C] = voronoin(X,options)
```

Description

`[V,C] = voronoin(X)` returns Voronoi vertices `V` and the Voronoi cells `C` of the Voronoi diagram of `X`. `V` is a `numv`-by-`n` array of the `numv` Voronoi vertices in n-dimensional space, each row corresponds to a Voronoi vertex. `C` is a vector cell array where each element contains the indices into `V` of the vertices of the corresponding Voronoi cell. `X` is an `m`-by-`n` array, representing `m` n-dimensional points, where `n > 1` and `m >= n+1`.

The first row of `V` is a point at infinity. If any index in a cell of the cell array is `1`, then the corresponding Voronoi cell contains the first point in `V`, a point at infinity. This means the Voronoi cell is unbounded.

`voronoin` uses Qhull.

`[V,C] = voronoin(X,options)` specifies a cell array of strings `options` to be used in Qhull. The default options are

- `{'Qbb'}` for 2- and 3-dimensional input
- `{'Qbb','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options are used. If `code` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see http://www.qhull.org.

Visualization

You can plot individual bounded cells of an n-dimensional Voronoi diagram. To do this, use `convhulln` to compute the vertices of the facets that make up the Voronoi cell. Then use `patch` and other plot functions to generate the figure.

Examples

Example 1

Let

```
x = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ]
```

then

```
[V,C] = voronoin(x)
```

```
V =
       Inf       Inf
       0.3833    0.3833
       0.7000   -1.6500
       0.2875    0.0000
      -0.0000    0.2875
      -0.0000   -0.0000
      -0.0500   -0.5250
      -0.0500   -0.0500
      -1.7500    0.7500
      -1.4500    0.6500
C =

    [1x4 double]
    [1x5 double]
    [1x4 double]
    [1x4 double]
    [1x4 double]
    [1x5 double]
    [1x4 double]
```

Use a `for` loop to see the contents of the cell array `C`.

```
for i=1:length(C), disp(C{i}), end
```

```
     4     2     1     3
    10     5     2     1     9
     9     1     3     7
    10     8     7     9
    10     5     6     8
     8     6     4     3     7
     6     4     2     5
```

In particular, the fifth Voronoi cell consists of 4 points: `V(10,:)`, `V(5,:)`, `V(6,:)`, `V(8,:)`.

Example 2

The following example illustrates the `options` input to `voronoin`. The commands

```
X = [-1 -1; 1 -1; 1 1; -1 1];
[V,C] = voronoin(X)
```

return an error message.

```
? qhull input error: can not scale last coordinate. Input is
cocircular
   or cospherical. Use option 'Qz' to add a point at infinity.
```

The error message indicates that you should add the option `'Qz'`. The following command passes the option `'Qz'`, along with the default `'Qbb'`, to `voronoin`.

```
[V,C] = voronoin(X,{'Qbb','Qz'})
V =

   Inf   Inf
     0     0


C =

    [1x2 double]
    [1x2 double]
    [1x2 double]
    [1x2 double]
```

More About

Algorithms

`voronoin` is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/.

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483.

See Also

`delaunayTriangulation` | `convhull` | `convhulln` | `delaunay` | `delaunayn` | `voronoi`

Introduced before R2006a

waitbar

Open or update wait bar dialog box

Syntax

```
h = waitbar(x,'message')
waitbar(x,'message','CreateCancelBtn','button_callback')
waitbar(x,'message',property_name,property_value,...)
waitbar(x)
waitbar(x,h)
waitbar(x,h,'updated message')
```

Description

A wait bar is a figure that displays what percentage of a calculation is complete as the calculation proceeds by progressively filling a bar with red from left to right.

`h = waitbar(x,'message')` displays a wait bar of fractional length `x`. The wait bar figure displays until the code that controls it closes it or the use clicks its Close Window button. Its (figure) handle is returned in `h`. The argument `x` must be between `0` and `1`.

Note Wait bars are not modal figures (their WindowStyle is `'normal'`). They often appear to be modal because the computational loops within which they are called prevent interaction with the Command Window until they terminate. For more information, see WindowStyle in Figure Properties.

`waitbar(x,'message','CreateCancelBtn','button_callback')` specifying `CreateCancelBtn` adds a **Cancel** button to the figure that executes the MATLAB commands specified in `button_callback` when the user clicks the **Cancel** button or the **Close Figure** button. `waitbar` sets both the **Cancel** button callback and the figure CloseRequestFcn to the string specified in `button_callback`.

`waitbar(x,'message',property_name,property_value,...)` optional arguments `property_name` and `property_value` enable you to set Figure Properties for the `waitbar`.

`waitbar(x)` subsequent calls to `waitbar(x)` extend the length of the bar to the new position `x`. Successive values of `x` normally increase. If they decrease, the wait bar runs in reverse.

`waitbar(x,h)` extends the length of the bar in the wait bar `h` to the new position `x`.

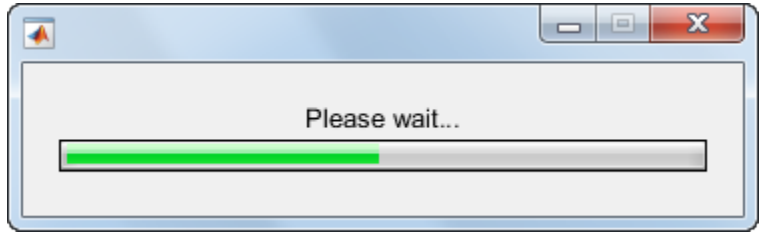
`waitbar(x,h,'updated message')` updates the message text in the `waitbar` figure, in addition to setting the fractional length to `x`.

Examples

Example 1 — Basic Wait Bar

Typically, you call `waitbar` repeatedly inside a `for` loop that performs a lengthy computation. For example:

```
h = waitbar(0,'Please wait...');
steps = 1000;
for step = 1:steps
    % computations take place here
    waitbar(step / steps)
end
close(h)
```



Example 2 — Wait Bar with Dynamic Text and Cancel Button

Adding a **Cancel** button allows user to abort the computation. Clicking it sets a logical flag in the figure's application data (appdata). The function tests for that value within the main loop and exits the loop as soon as the flag has been set. The example iteratively approximates the value of π. At each step, the current value is encoded as a string and

displayed in the wait bar's message field. When the function finishes, it destroys the wait bar and returns the current estimate of π and the number of steps it ran.

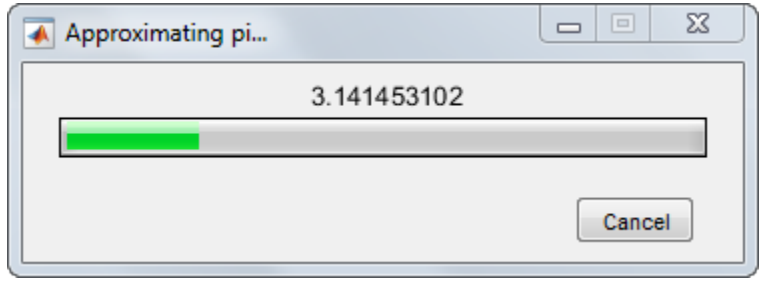
Copy the following function to a code file and save it as `approxpi.m`. Execute it as follows, allowing it to run for 10,000 iterations.

```
[estimated_pi steps] = approxpi(10000)
```

You can click **Cancel** or close the window to abort the computation and return the current estimate of π.

```
function [valueofpi step] = approxpi(steps)
% Converge on pi in steps iterations, displaying waitbar.
% User can click Cancel or close button to exit the loop.
% Ten thousand steps yields error of about 0.001 percent.

h = waitbar(0,'1','Name','Approximating pi...',...
            'CreateCancelBtn',...
            'setappdata(gcbf,''canceling'',1)');
setappdata(h,'canceling',0)
% Approximate as pi^2/8 = 1 + 1/9 + 1/25 + 1/49 + ...
pisqover8 = 1;
denom = 3;
valueofpi = sqrt(8 * pisqover8);
for step = 1:steps
    % Check for Cancel button press
    if getappdata(h,'canceling')
        break
    end
    % Report current estimate in the waitbar's message field
    waitbar(step/steps,h,sprintf('%12.9f',valueofpi))
    % Update the estimate
    pisqover8 = pisqover8 + 1 / (denom * denom);
    denom = denom + 2;
    valueofpi = sqrt(8 * pisqover8);
end
delete(h)       % DELETE the waitbar; don't try to CLOSE it.
```

The function sets the figure Name property to describe what is being computed. In the `for` loop, calling `waitbar` sets the fractional progress indicator and displays intermediate results. the code `waitbar(i/steps,h,sprintf('%12.9f',valueofpi))` sets the wait bar's `message` variable to a string representation of the current estimate of *pi*. Naturally, the extra computation involved makes iterations last longer than they need to, but such feedback can be helpful to users.



Note: You should call `delete` to remove a wait bar when you give it a CloseRequestFcn, as in the preceding code; calling `close` does not close it, and makes its Cancel and Close Window buttons unresponsive. This happens because the figure's CloseRequestFcn callback recursively calls itself. In such a situation you must forcibly remove the wait bar, for example like this:

```
set(groot,'ShowHiddenHandles','on')
delete(get(groot,'Children'))
```

However, as issuing these commands will delete all open figures—not just the wait bar—it is best never to use `close` in a CloseRequestFcn to close a window.

See Also

`close` | `delete` | `dialog` | `msgbox` | `getappdata` | `setappdata`

Introduced before R2006a

waitfor

Block execution and wait for condition

Syntax

```
waitfor(h)
waitfor(h,PropertyName)
waitfor(h,PropertyName,PropertyValue)
```

Description

`waitfor(h)` blocks the caller from executing statements until the graphics object identified by `h` closes (is deleted). When object `h` no longer exists, `waitfor` returns, enabling execution to resume. If the object does not exist, `waitfor` returns immediately without processing any events.

`waitfor(h,PropertyName)` blocks the caller from executing until the value of `PropertyName` (any property of the graphics object `h`) changes or `h` closes (is deleted). If `PropertyName` is not a valid property for the object, MATLAB returns an error.

`waitfor(h,PropertyName,PropertyValue)` blocks the caller from executing until the value of `PropertyName` for the graphics object `h` changes to the specific value `PropertyValue` or `h` closes (is deleted). If the value of `PropertyName` is already `PropertyValue`, `waitfor` returns immediately without processing any events.

Here are some important characteristics of the `waitfor` function:

- The `waitfor` function prevents its caller from continuing, but callbacks that respond to various user actions (for example, pressing a mouse button) can still run.
- The `waitfor` function also blocks Simulink models from executing, but callbacks do still execute.
- The `waitfor` function can block nested function calls. For example, a callback that executes while the `waitfor` function is running can call `waitfor`.
- If a callback function of a UI component is currently executing the `waitfor` function, then that callback can be interrupted regardless of that component's Interruptible property value.

- If you type **Ctrl+C** in the Command Window while the `waitfor` function is executing, the executing program terminates. To avoid terminating, the program can call the `waitfor` function within a `try/catch` block that handles the exception that typing **Ctrl+C** generates.

Examples

Create a plot and pause execution of the rest of the statements until you close the figure window:

```
f = warndlg('This is a warning.', 'A Warning Dialog');
disp('This prints immediately');
drawnow     % Necessary to print the message
waitfor(f);
disp('This prints after you close the warning dialog');
```

Suspend execution until name of figure changes:

```
f = figure('Name', datestr(now));
h = uicontrol('String','Change Name',...
              'Position',[20 20 100 30],...
'Callback', 'set(gcbf, ''Name'', datestr(now))');
disp('This prints immediately');
drawnow     % Necessary to print the message
waitfor(f, 'Name');
disp('This prints after clicking the push button');
```

Display text object and wait for user to edit it:

```
figure;
textH = text(.5, .5, 'Edit me and click away');
set(textH,'Editing','on', 'BackgroundColor',[1 1 1]);
disp('This prints immediately.');
drawnow
waitfor(textH,'Editing','off');
set(textH,'BackgroundColor',[1 1 0]);
disp('This prints after text editing is complete.');
```

If you close the figure while `waitfor` is executing, an error occurs because the code attempts to access handles of objects that no longer exist. You can handle the error by enclosing code starting with the call to `waitfor` in a try/catch block, as follows:

```
figure;
```

```
textH = text(.5, .5, 'Edit me and click away');
set(textH,'Editing','on', 'BackgroundColor',[1 1 1]);
disp('This prints immediately.');
drawnow
% Use try/catch block to handle errors,
% such as deleting figure
try
    waitfor(textH,'Editing','off');
    set(textH,'BackgroundColor',[1 1 0]);
    disp('This prints after text editing is complete.');
catch ME
    disp('This prints if figure is deleted:')
    disp(ME.message)
  % You can place other code to respond to the error here
end
```

The `ME` variable is a MATLAB Exception object that you can use to determine the type of error that occurred. For more information, see " Respond to an Exception".

See Also

`drawnow` | `pause` | `keyboard` | `uiresume` | `uiwait` | `waitforbuttonpress`

Introduced before R2006a

waitforbuttonpress

Wait for key press or mouse-button click

Note: The behavior of the `waitforbuttonpress` function changed in R2014b. The figure that is current when you call the `waitforbuttonpress` function is the only area in which users can press a key or click a mouse button to resume program execution.

Syntax

```
k = waitforbuttonpress
```

Description

`k = waitforbuttonpress` blocks the caller's execution stream until the function detects that the user has clicked a mouse button or pressed a key while the figure window is active. The figure that is current when you call the `waitforbuttonpress` function is the only area in which users can press a key or click a mouse button to resume program execution. The return argument, `k`, can have these values:

- `0` if it detects a mouse button click
- `1` if it detects a key press

If a `WindowButtonDownFcn` callback is defined for the figure, its callback is executed before `waitforbuttonpress` returns a value.

Only keys that generate characters cause the function to return. Pressing any of the following keys by itself does nothing: **Ctrl**, **Shift**, **Alt**, **Caps_lock**, **Num_lock**, **Scroll_lock**.

These figure properties provide additional information about the user's interaction with the window: `CurrentCharacter`, `SelectionType`, and `CurrentPoint`.

You can interrupt the `waitforbuttonpress` function by typing **Ctrl**+C, but it returns an error unless your code calls it from within a `try/catch` block. The

`waitforbuttonpress` function errors if the user closes the figure by clicking the **X** (close box) unless your code calls the `waitforbuttonpress` function within a `try/catch` block.

Examples

These statements display text in the Command Window when the user either clicks a mouse button or types a key in the figure window:

```
f = figure;
w = waitforbuttonpress;
if w == 0
    disp('Button click')
else
    disp('Key press')
end
```

See Also

`dragrect` | `ginput` | `rbbox` | `waitfor`

Introduced before R2006a

warndlg

Create warning dialog box

Syntax

```
h = warndlg
h = warndlg(warningstring)
h = warndlg(warningstring,dlgname)
h = warndlg(warningstring,dlgname,createmode)
```

Description

`h = warndlg` displays a dialog box named `Warning Dialog` containing the string, `This is the default warning string`. The `warndlg` function returns the handle of the dialog box in `h`. The warning dialog box disappears after the user clicks **OK**.

`h = warndlg(warningstring)` displays a dialog box with the title `Warning Dialog` containing the string specified by `warningstring`. The `warningstring` argument can be any valid string format – cell arrays are preferred.

To use multiple lines in your warning, define `warningstring` using either of the following:

- `sprintf` with `newline` characters separating the lines

  ```
  warndlg(sprintf('Message line 1 \n Message line 2'))
  ```

- Cell arrays of strings

  ```
  warndlg({'Message line 1';'Message line 2'})
  ```

`h = warndlg(warningstring,dlgname)` displays a dialog box with title `dlgname`.

`h = warndlg(warningstring,dlgname,createmode)` specifies whether the warning dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `warningstring` and `dlgname`. The *`createmode`* argument can be a string or a structure.

If *createmode* is a string, it must be one of the values shown in the following table.

createmode Value	Description
`modal`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a modal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.
`nonmodal` (default)	Creates a new nonmodal warning dialog box with the specified parameters. Existing warning dialog boxes with the same title are not deleted.
`replace`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a nonmodal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

Note: A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

Modal dialogs (created using `errordlg`, `msgbox`, or `warndlg`) replace any existing dialogs created with these functions that also have the same name.

For more information about modal dialog boxes, see WindowStyle in Figure Properties.

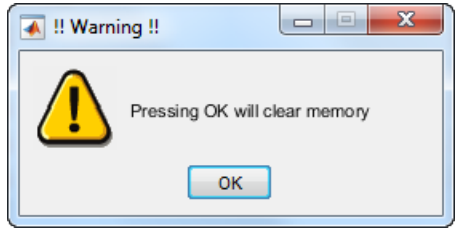
If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings `'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

Examples

The statement

```
warndlg('Pressing OK will clear memory','!! Warning !!')
```

displays this dialog box:



See Also

`dialog` | `errordlg` | `helpdlg` | `inputdlg` | `listdlg` | `msgbox` | `questdlg` | `figure` | `uiwait` | `uiresume` | `warning`

Introduced before R2006a

warning

Display warning message

Syntax

```
warning(msg)
warning(msg,A1,...,An)
warning(msgID, ___ )

warning(state)
warning(state,msgID)
warning

warnStruct = warning
warning(warnStruct)

warning(state,mode)
warnStruct = warning(state,mode)
```

Description

`warning(msg)` displays the warning message and sets the warning state for the`lastwarn` function. If `msg` is an empty string (`''`), `warning` resets the warning state for `lastwarn`, but does not display any text.

`warning(msg,A1,...,An)` displays a message string that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `msg` is converted to one of the values `A1,...,An`.

`warning(msgID, ___ )` attaches a warning identifier to the warning message. You can include any of the input arguments in the previous syntaxes. The identifier enables you to distinguish warnings and to control what happens when MATLAB encounters the warnings.

`warning(state)` enables, disables, or displays the state of all warnings.

`warning(state,msgID)` acts on the state of a specified warning.

`warning` displays the state of all of the warnings. It is equivalent to `warning('query')`.

`warnStruct = warning` returns a structure or array of structures that contains information about which warnings are enabled and disabled. `warnStruct` includes an `identifier` field with a `msgID` or `'all'`, and a `state` field that indicates the state of the corresponding warning.

`warning(warnStruct)` sets the current warning settings as indicated in the structure array, `warnStruct`.

`warning(state,mode)` controls whether MATLAB displays the stack trace or additional information about the warning.

`warnStruct = warning(state,mode)` returns a structure with an `identifier` field containing the `mode` and a `state` field containing the current state of `mode`. If you pass the output structure, `warnStruct`, into the `warning` function, you set the state of the `mode`, not which warnings are enabled or disabled.

Examples

Issue Warning Message

Generate a warning that displays a message.

```
n = 7;
if ~ischar(n)
   warning('Input must be a string')
end
```

```
Warning: Input must be a string
```

Include information about `n` in the warning message.

```
if ~ischar(n)
   warning('Input must be a string, not a %s',class(n))
end
```

```
Warning: Input must be a string, not a double
```

Attach a message identifier to the warning message.

```
if ~ischar(n)
```

```
   warning('MyComponent:incorrectType',...
       'Input must be a string, not a %s',class(n))
end
```

```
Warning: Input must be a string, not a double
```

Set and Query Warning State

Disable all warnings.

```
warning('off')
```

Query the warnings.

```
warning
```

```
All warnings have the state 'off'.
```

Enable all warnings, disable the singular matrix warning, and query all warnings.

```
warning('on')
warning('off','MATLAB:singularMatrix')
warning
```

```
The default warning state is 'on'. Warnings not set to the default are

State  Warning Identifier

    off  MATLAB:singularMatrix
```

Re-enable the singular matrix warning.

```
warning('on','MATLAB:singularMatrix')
```

Save and Restore Warning Settings

Enable all warnings, and then disable the singular matrix warning.

```
warning('on')
warning('off','MATLAB:singularMatrix')
```

Save the current warning settings.

```
s = warning
```

```
s =
```

```
2x1 struct array with fields:

    identifier
    state
```

Examine the two structures.

```
s(1)
```

```
ans =

    identifier: 'all'
         state: 'on'
```

```
s(2)
```

```
ans =

    identifier: 'MATLAB:singularMatrix'
         state: 'off'
```

All warnings are enabled except for `'MATLAB:singularMatrix'`.

Disable and query all warnings.

```
warning('off')
warning('query')
```

```
All warnings have the state 'off'.
```

Restore the saved warning state structure, and query the state.

```
warning(s)
warning('query')
```

```
The default warning state is 'on'. Warnings not set to the default are

State  Warning Identifier

    off  MATLAB:singularMatrix
```

Control Warning Verbosity

Ensure `verbose` and `backtrace` settings are the default values.

```
warning('off','verbose')
warning('on','backtrace')
```

Turn on all warnings, and remove a folder that does not exist on the MATLAB path.

```
warning('on')
rmpath('nosuchfolder')
```

```
Warning: "nosuchfolder" not found in path.
> In rmpath at 57
```

Enable verbosity to display an extended warning message.

```
warning('on','verbose')
rmpath('nosuchfolder')
```

```
Warning: "nosuchfolder" not found in path.
(Type "warning off MATLAB:rmpath:DirNotFound" to suppress this warning.)

> In rmpath at 57
```

Disable display of the stack trace.

```
warning('off','backtrace')
rmpath('nosuchfolder')
```

```
Warning: "nosuchfolder" not found in path.
(Type "warning off MATLAB:rmpath:DirNotFound" to suppress this warning.)
```

Temporarily Disable Warning

Compute a singular matrix.

```
A = eye(2);
B = [3 6; 4 8];
C = B\A;
```

```
Warning: Matrix is singular to working precision.
```

Find the warning ID, save the current warning state, and disable the specific warning

```
[msgStr,msgId] = lastwarn;
warnStruct = warning('off',msgId);
C = B\A;
```

Restore previous warning state.

```
warning(warnStruct);
C = B\A;
```

```
Warning: Matrix is singular to working precision.
```

- “Issue Warnings and Errors”
- “Suppress Warnings”
- “Restore Warnings”
- “Change How Warnings Display”

Input Arguments

`msg` — Information about cause of warning

`string`

Information about the cause of the warning and how you might correct it, specified as a string. To format the string, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see “Formatting Strings”.

Note: You must specify more than one input argument with `warning` if you want MATLAB to convert special characters (such as `\t`, `\n`, `%s`, and `%d`) in the warning message string.

Example: `'Input must be a string.'`

`A1,...,An` — Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array. This input argument provides the values that correspond to and replace the conversion specifiers in `msg`.

`msgID` — Identifier for warning

string | `'all'` | `'last'`

Identifier for the warning, specified as a string, `'all'`, or `'last'`. Use the warning identifier to help identify the source of the warning or to control a selected subset of the warnings in your program.

The message identifier includes a `component` and `mneumonic`. The identifier must always contain a colon and follows this simple format: `component:mnemonic`. The `component` and `mneumonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No whitespace characters can appear anywhere in msgID. For more information, see "Message Identifiers".

When you set the state of a warning, the `msgID` can have a value of `'all'` or `'last'`. Use `'all'` to set the state of all warnings, and use `'last'` to set the state of the last issued warning.

Example: `'MATLAB:singularMatrix'`

Example: `'MATLAB:narginchk:notEnoughInputs'`

`state` — Warning control indicator

`'on' | 'off' | 'query'`

Warning control indicator specified as `'on'`, `'off'`, or `'query'`. Use `'on'` or `'off'` to control whether MATLAB issues a warning. Use `'query'` to query the current state of the warning.

`warnStruct` — Warning settings

structure | array of structures

Warning settings, specified as a structure or array of structures that contains information about which warnings are enabled and which are disabled. `warnStruct` includes an `identifier` field with a `msgID` or `'all'`, and `state` field indicating the state of the corresponding warning.

`mode` — Verbosity and sack trace display settings

`'backtrace' | 'verbose'`

Verbosity and the stack trace display of settings, specified by `'backtrace'` or `'verbose'`. By default, the state of verbosity is set to `'off'` and the state of stack trace display is set to `'on'`.

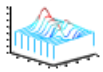
See Also

`dbstop` | `disp` | `error` | `errordlg` | `lasterror` | `lastwarn` | `sprintf` | `warndlg`

Introduced before R2006a

waterfall

Waterfall plot



Syntax

```
waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)
waterfall(axes_handles,...)
h = waterfall(...)
```

Description

The `waterfall` function draws a mesh similar to the `meshz` function, but it does not generate lines from the columns of the matrices. This produces a “waterfall” effect.

`waterfall(Z)` creates a waterfall plot using `x = 1:size(Z,2)` and `y = 1:size(Z,1)`. `Z` determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in `X`, `Y`, and `Z`. `Z` also determines the color, so color is proportional to the surface height. If `X` and `Y` are vectors, `X` corresponds to the columns of `Z`, and `Y` corresponds to the rows, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(Z)`. `X` and `Y` are vectors or matrices that define the *x*- and *y*-coordinates of the plot. `Z` is a matrix that defines the *z*-coordinates of the plot (i.e., height above a plane). If `C` is omitted, color is proportional to `Z`.

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of `C`, which must be the same size as `Z`.

MATLAB performs a linear transformation on `C` to obtain colors from the current colormap.

`waterfall(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

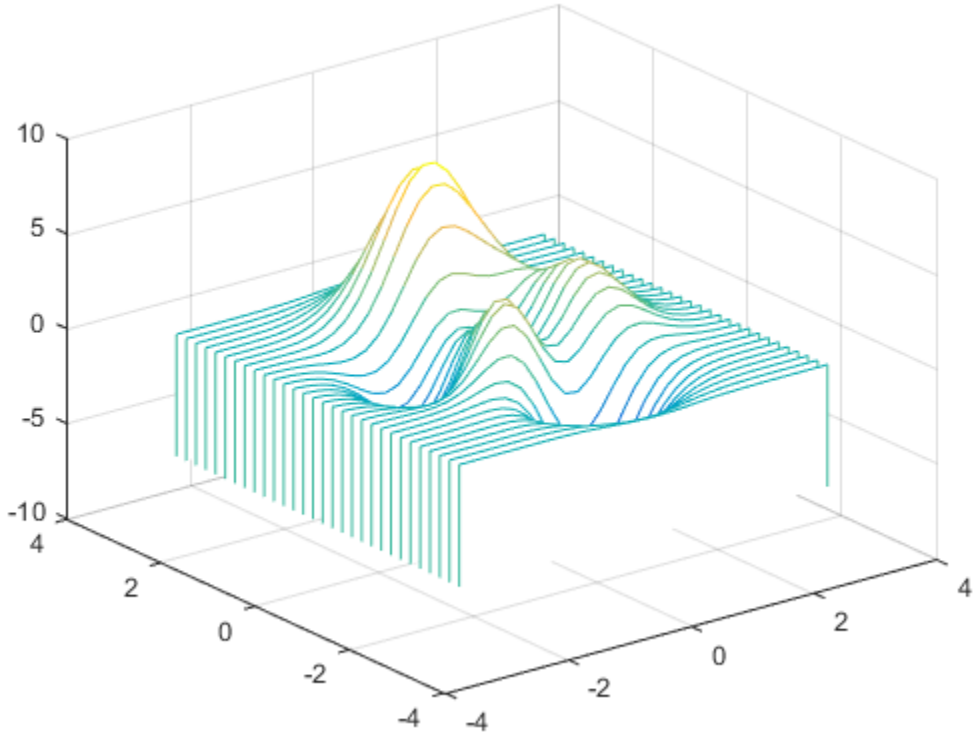
`h = waterfall(...)` returns the handle of the patch graphics object used to draw the plot.

Examples

Create Waterfall Plot

Create a waterfall plot of the peaks function.

```
figure
[X,Y,Z] = peaks(30);
waterfall(X,Y,Z)
```



More About

Tips

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

Algorithms

The range of `X`, `Y`, and `Z`, or the current setting of the axes `XLim`, `YLim`, and `ZLim` properties, determines the range of the axes (also set by `axis`). The range of `C`, or the current setting of the axes `CLim` property, determines the color scaling (also set by `caxis`).

The `CData` property for the patch graphics objects specifies the color at every point along the edge of the patch, which determines the color of the lines.

The waterfall plot looks like a mesh surface; however, it is a patch graphics object. To create a surface plot similar to `waterfall`, use the `meshz` function and set the `MeshStyle` property of the surface to `'Row'`. For a discussion of parametric surfaces and related color properties, see `surf`.

See Also

`axis` | `caxis` | `meshz` | `ribbon` | `surf`

Introduced before R2006a

wavfinfo

Information about WAVE (`.wav`) sound file

Note: `wavfinfo` will be removed in a future release. Use `audioinfo` instead.

Syntax

```
[m d] = wavfinfo(filename)
```

Description

`[m d] = wavfinfo(filename)` returns information about the contents of the WAVE sound file specified by the string *`filename`*. Enclose the *`filename`* input in single quotes.

`m` is the string `'Sound (WAV) file'`, if *`filename`* is a WAVE file. Otherwise, it contains an empty string (`''`).

`d` is a string that reports the number of samples in the file and the number of channels of audio data. If *`filename`* is not a WAVE file, it contains the string `'Not a WAVE file'`.

See Also

`audioplayer` | `audiorecorder` | `audioread` | `audiowrite`

Introduced before R2006a

wavplay

Play recorded sound on PC-based audio output device

Note: `wavplay` has been removed. Use `audioplayer` instead.

Syntax

```
wavplay(y,Fs)
wavplay(y,Fs,mode)
```

Description

`wavplay(y,Fs)` plays the audio signal stored in the vector `y` on a PC-based audio output device. `Fs` is the integer sample rate in Hz (samples per second). The default value for `Fs` is 11025 Hz. `wavplay` supports only 1- or 2-channel (mono or stereo) audio signals. To play in stereo, `y` must be a two-column matrix.

`wavplay(y,Fs,mode)` specifies how `wavplay` interacts with the command line. The string `mode` is one of the following:

- `'sync'` (default): You do not have access to the command line until the sound has finished playing (a blocking device call).
- `'async'`: You have immediate access to the command line as soon as the sound begins to play on the audio output device (a nonblocking device call). If you call `wavplay` again in `async` mode while the audio is playing, `wavplay` blocks access to the command line until the original playback completes.

The audio signal `y` can be one of four data types. The number of bits used to quantize and play back each sample depends on the data type.

Data Types for wavplay

Data Type	Quantization
Double-precision (default value)	16 bits/sample

Data Type	Quantization
Single-precision	16 bits/sample
16-bit signed integer	16 bits/sample
8-bit unsigned integer	8 bits/sample

Examples

The MAT-files `gong.mat` and `chirp.mat` both contain an audio signal `y` and a sampling frequency `Fs`. Load and play the gong and the chirp audio signals. Change the names of these signals in between `load` commands and play them sequentially using the `'sync'` option for `wavplay`.

```
load chirp;
y1 = y; Fs1 = Fs;
load gong;
wavplay(y1,Fs1,'sync') % The chirp signal finishes before the
wavplay(y,Fs)          % gong signal begins playing.
```

More About

Tips

The `wavplay` function is for use only with 32-bit Microsoft Windows operating systems. To play audio data on other platforms, use `audioplayer`.

See Also

`audioplayer` | `audioinfo` | `audioread` | `audiowrite`

Introduced before R2006a

wavread

Read WAVE (`.wav`) sound file

Note: `wavread` will be removed in a future release. Use `audioread` instead.

Syntax

```
y = wavread(filename)
[y, Fs] = wavread(filename)
[y, Fs, nbits] = wavread(filename)
[y, Fs, nbits, opts] = wavread(filename)
[ ___ ] = wavread(filename, N)
[ ___ ] = wavread(filename, [N1 N2])
[ ___ ] = wavread( ___ , fmt)
siz = wavread(filename,'size')
```

Description

`y = wavread(filename)` loads a WAVE file specified by the string *`filename`*, returning the sampled data in *`y`*. If *`filename`* does not include an extension, `wavread` appends `.wav`.

`[y, Fs] = wavread(filename)` returns the sample rate (*`Fs`*) in Hertz used to encode the data in the file.

`[y, Fs, nbits] = wavread(filename)` returns the number of bits per sample (*`nbits`*).

`[y, Fs, nbits, opts] = wavread(filename)` returns a structure *`opts`* of additional information contained in the WAV file. The content of this structure differs from file to file. Typical structure fields include *`opts`*`.fmt` (audio format information) and *`opts`*`.info` (text that describes the title, author, etc.).

`[ ___ ] = wavread(filename, N)` returns only the first *`N`* samples from each channel in the file.

`[ ___ ] = wavread(filename, [N1 N2])` returns only samples *N1* through *N2* from each channel in the file.

`[ ___ ] = wavread( ___ , fmt)` specifies the data format of *y* used to represent samples read from the file. *fmt* can be either of the following values, or a partial match (case-insensitive):

`'double'`	Double-precision normalized samples (default).
`'native'`	Samples in the native data type found in the file.

siz `= wavread(`*filename*`,'size')` returns the size of the audio data contained in *filename* instead of the actual audio data, returning the vector *siz* `= [`*samples channels*`]`.

Output Scaling

The range of values in *y* depends on the data format *fmt* specified. Some examples of output scaling based on typical bit-widths found in a WAV file are given below for both `'double'` and `'native'` formats.

Native Formats

Number of Bits	MATLAB Data Type	Data Range
8	`uint8` (unsigned integer)	0 <= *y* <= 255
16	`int16` (signed integer)	-32768 <= *y* <= +32767
24	`int32` (signed integer)	-2^23 <= *y* <= 2^23-1
32	`single` (floating point)	-1.0 <= *y* < +1.0

Double Formats

Number of Bits	MATLAB Data Type	Data Range
N<32	`double`	-1.0 <= *y* < +1.0
N=32	`double`	-1.0 <= *y* <= +1.0 Note: Values in *y* might exceed -1.0 or +1.0 for the case of N=32 bit data samples stored in the WAV file.

`wavread` supports multi-channel data, with up to 32 bits per sample.

`wavread` supports Pulse-code Modulation (PCM) data format only.

Examples

Create a WAV file from the example file `handel.mat`, and read portions of the file back into MATLAB.

```
% Create WAV file in current folder.
load handel.mat

hfile = 'handel.wav';
wavwrite(y, Fs, hfile)
clear y Fs

% Read the data back into MATLAB, and listen to audio.
[y, Fs, nbits, readinfo] = wavread(hfile);
sound(y, Fs);

% Pause before next read and playback operation.
duration = numel(y) / Fs;
pause(duration + 2)

% Read and play only the first 2 seconds.
nsamples = 2 * Fs;
[y2, Fs] = wavread(hfile, nsamples);
sound(y2, Fs);
pause(4)

% Read and play the middle third of the file.
sizeinfo = wavread(hfile, 'size');

tot_samples = sizeinfo(1);
startpos = tot_samples / 3;
endpos = 2 * startpos;

[y3, Fs] = wavread(hfile, [startpos endpos]);
sound(y3, Fs);
```

See Also

`mmfileinfo` | `audioinfo` | `audiowrite` | `audioread` | `audioplayer` | `audiorecorder` | `sound`

Introduced before R2006a

wavrecord

Record sound using PC-based audio input device

Note: `wavrecord` has been removed. Use `audiorecorder` instead.

Syntax

```
y = wavrecord(n,Fs)
y = wavrecord(___,ch)
y = wavrecord(___,'dtype')
```

Description

`y = wavrecord(n,Fs)` records `n` samples of an audio signal, sampled at a rate of `Fs` Hz (samples per second). The default value for `Fs` is 11025 Hz.

`y = wavrecord(___,ch)` uses `ch` number of input channels from the audio device. `ch` can be either 1 or 2, for mono or stereo, respectively. The default value for `ch` is 1.

`y = wavrecord(___,'dtype')` uses the data type specified by the string `'dtype'` to record the sound. The following table lists the string values for `'dtype'` along with the corresponding bits per sample and acceptable data range for `y`.

dtype	Bits/sample	y Data Range
`'double'`	16	–1.0 <= y < +1.0
`'single'`	16	–1.0 <= y < +1.0
`'int16'`	16	–32768 <= y <= +32767
`'uint8'`	8	0 <= y <= 255

Examples

Record 5 seconds of 16-bit audio sampled at 11025 Hz. Play back the recorded sound using `wavplay`. Speak into your audio device (or produce your audio signal) while the `wavrecord` command runs.

```
Fs = 11025;
y = wavrecord(5*Fs,Fs,'int16');
wavplay(y,Fs);
```

More About

Tips

Standard sampling rates for PC-based audio hardware are 8000, 11025, 22050, and 44100 samples per second. Stereo signals are returned as two-column matrices. The first column of a stereo audio matrix corresponds to the left input channel, while the second column corresponds to the right input channel.

The `wavrecord` function is for use only with 32-bit Microsoft Windows operating systems. To record audio data from audio input devices on other platforms, use `audiorecorder`.

See Also

`audiorecorder` | `audioinfo` | `audioread` | `audiowrite`

Introduced before R2006a

wavwrite

Write WAVE (`.wav`) sound file

Note: `wavwrite` will be removed in a future release. Use `audiowrite` instead.

Syntax

```
wavwrite(y,filename)
wavwrite(y,Fs,filename)
wavwrite(y,Fs,N,filename)
```

Description

`wavwrite(y,filename)` writes the data stored in the variable *y* to a WAVE file called *filename*. The *filename* input is a string enclosed in single quotes. The data has a sample rate of 8000 Hz and is assumed to be 16-bit. Each column of the data represents a separate channel. Therefore, stereo data should be specified as a matrix with two columns.

`wavwrite(y,Fs,filename)` writes the data stored in the variable *y* to a WAVE file called *filename*. The data has a sample rate of *Fs* Hz and is assumed to be 16-bit.

`wavwrite(y,Fs,N,filename)` writes the data stored in the variable *y* to a WAVE file called *filename*. The data has a sample rate of *Fs* Hz and is *N*-bit, where *N* is 8, 16, 24, or 32.

Input Data Ranges

The range of values in *y* depends on the number of bits specified by *N* and the data type of *y*. The following tables list the valid input ranges based on the value of *N* and the data type of *y*.

If *y* contains integer data:

N Bits	y Data Type	y Data Range	Output Format
8	`uint8`	0 <= *y* <= 255	`uint8`
16	`int16`	–32768 <= *y* <= +32767	`int16`
24	`int32`	–2^23 <= *y* <= 2^23 – 1	`int32`

If `y` contains floating-point data:

N Bits	y Data Type	y Data Range	Output Format
8	`single` or `double`	–1.0 <= *y* < +1.0	`uint8`
16	`single` or `double`	–1.0 <= *y* < +1.0	`int16`
24	`single` or `double`	–1.0 <= *y* < +1.0	`int32`
32	`single` or `double`	–1.0 <= *y* <= +1.0	`single`

For floating point data where *N* < 32, amplitude values are clipped to the range –1.0 <= *y* < +1.0.

Note 8-, 16-, and 24-bit files are type 1 integer pulse code modulation (PCM). 32-bit files are written as type 3 normalized floating point.

See Also

`mmfileinfo` | `audioinfo` | `audioread` | `audiowrite` | `audioplayer` | `audiorecorder` | `sound`

Introduced before R2006a

web

Open Web page or file in browser

Syntax

```
web
web(url)
web(url,opt)
web(url,opt1,...,optN)

stat = web( ___ )
[stat,h] = web( ___ )
[stat,h,url] = web( ___ )
```

Description

`web` opens an empty MATLAB Web browser.

`web(url)` opens the page specified by `url` in the MATLAB Web browser. If multiple browsers are open, the page displays in the one that was most recently used.

`web(url,opt)` opens the page using the specified browser option, such as `'-new'` to create a new browser instance or `'-browser'` to use the system browser.

`web(url,opt1,...,optN)` opens the page using one or more browser options.

`stat = web( ___ )` returns the status of the operation: `0` if successful, `1` or `2` if unsuccessful. You can include any of the input arguments in previous syntaxes.

`[stat,h] = web( ___ )` returns a handle to a MATLAB Web browser that allows you to close it using the command `close(h)`. If you do not specify any inputs to the `web` function, such as `[stat,h] = web`, then the handle corresponds to the most recently used MATLAB Web browser.

`[stat,h,url] = web( ___ )` returns the URL of the current page in the MATLAB Web browser.

Examples

Web Page in MATLAB Web Browser

Open the MathWorks Web site home page.

```
url = 'http://www.mathworks.com';
web(url)
```

Open the page in a new instance of the browser that does not include a toolbar.

```
web(url,'-new','-notoolbar')
```

File in MATLAB Web Browser

View an HTML file that resides on your system.

Create an HTML file by publishing an example program file. Copy the program file to the current folder so that the code can run during the publishing process.

```
program = fullfile(matlabroot,'help','techdoc',...
          'matlab_env','examples','fourier_demo2.m');
copyfile(program);
htmlFile = publish('fourier_demo2.m');
```

View a file by specifying the file name.

```
web(htmlFile)
```

Alternatively, you can use the `file:///` URL scheme, as long as you include the full path. The `publish` function returns the path in the `htmlFile` output.

```
url = ['file:///',htmlFile];
web(url)
```

Web Page in System Browser

Open the MathWorks Web site home page in the system browser.

```
url = 'http://www.mathworks.com';
web(url,'-browser')
```

Email from System Browser

Send email from your system browser's default mail application using the `mailto:` URL scheme.

To run this example, replace the value for `email` with a valid email address.

```
email = 'myaddress@provider.ext';
url = ['mailto:',email];
web(url)
```

Handle to MATLAB Web Browser

Open the MathWorks Web site home page, and then close the browser using its handle.

```
url = 'http://www.mathworks.com';
[stat,h] = web(url);
```

Close the browser window.

```
close(h)
```

Text Displayed in MATLAB Web Browser

View formatted text using the `text://` URL scheme.

```
web('text://<html><h1>Hello World</h1></html>')
```

Input Arguments

`url` — Web page address or file location

string

Web page address or file location, specified as a string. File locations can include an absolute or relative path.

If `url` corresponds to a file in the installed product documentation, then the page displays in the MATLAB Help browser instead of the Web browser.

Example: `'http:\\www.mathworks.com'`

Example: `'myfolder/myfile.html'`

`opt` — Browser option

`'-browser' | '-new' | '-noaddressbox' | '-notoolbar'`

Browser option, specified as one of the following strings. Options can appear in any order.

`'-browser'`	Opens the page in a system browser window instead of the MATLAB Web browser. On Microsoft Windows and Apple Macintosh platforms, the operating system determines the system Web browser. On other systems, the default is the Mozilla® Firefox® browser, but you can change the default using MATLAB Web preferences.
`'-new'`	Opens the page in a new MATLAB Web browser window. Does not apply to the system browser.
`'-noaddressbox'`	Opens the page in a browser that does not display the address box. Only applies to new instances of the MATLAB Web browser.
`'-notoolbar'`	Opens the page in a browser that does not display a toolbar or address box. Only applies to new instances of the MATLAB Web browser.

Example: `'-new','-noaddressbox'`

Output Arguments

`stat` — Browser status

`0 | 1 | 2`

Browser status, returned as an integer with one of these values:

`0`	Found and launched system browser.
`1`	Could not find system browser.
`2`	Found, but could not launch system browser.

`h` — Handle to most recent MATLAB Web browser

scalar

Handle to the most recent MATLAB Web browser, returned as a scalar instance of the associated Java class.

If you do not request the handle when you open the page, be aware that this handle might not correspond to your most recent use of the `web` function. Other MATLAB functionality also uses the `web` function, such as links to external sites from the Help browser.

`url` — Current page address

string

Current page address in the most recent MATLAB Web browser, returned as a string.

More About

Tips

- If you plan to deploy an application that calls the `web` function using the MATLAB Compiler product, then use the `'-browser'` option.
- If you are displaying Japanese streaming text in the MATLAB Web browser, specify a header that includes the `charset` attribute. For example:

```
web(['text://<html><head><meta http-equiv="content-type" ' ...
     'content="text/html;charset=utf-8"></head><body>TEXT</body></html>'])
```

- "Web Browsers and MATLAB"
- "Specify Proxy Server Settings for Connecting to the Internet"
- "Specify the System Browser for Linux Platforms"

See Also

`urlread` | `urlwrite`

Introduced before R2006a

weboptions

Specify parameters for RESTful web service

Syntax

```
options = weboptions
options = weboptions(Name,Value)
```

Description

`options = weboptions` creates a default `weboptions` object to specify parameters for a request to a web service. A `weboptions` object can be an optional input argument to the `webread`, `websave`, and `webwrite` functions.

`options = weboptions(Name,Value)` creates a `weboptions` object with the named parameters set to the specified values.

Examples

Default `weboptions` Object

Create a default `weboptions` object and display the default values for its properties.

```
options = weboptions
```

```
options =

  weboptions with properties:

    CharacterEncoding: 'auto'
            UserAgent: 'MATLAB 8.5.0.184244 (R2015a)'
              Timeout: 5
             Username: ''
             Password: ''
              KeyName: ''
             KeyValue: ''
          ContentType: 'auto'
        ContentReader: []
```

```
           MediaType: 'application/x-www-form-urlencoded'
       RequestMethod: 'auto'
```

User Name and Password in `weboptions` Object

Set a user name and password in a `weboptions` object. When given this `weboptions` object, `webread` can provide the user name and password to a web service that requires authentication.

```
options = weboptions('Username','jdoe','Password','mypassword')
```

```
options =

  weboptions with properties:

    CharacterEncoding: 'auto'
            UserAgent: 'MATLAB 8.5.0.184244 (R2015a)'
              Timeout: 5
             Username: 'jdoe'
             Password: '**********'
              KeyName: ''
             KeyValue: ''
          ContentType: 'auto'
        ContentReader: []
            MediaType: 'application/x-www-form-urlencoded'
        RequestMethod: 'auto'
```

The password is obscured when you display the `weboptions` object. However, the object stores the password as plain text. You can retrieve the password from the `weboptions.Password` property.

```
options.Password
```

```
ans =

mypassword
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `weboptions('Timeout',60)` creates a `weboptions` object that sets the timeout connection duration to 60 seconds.

`'CharacterEncoding'` — Character encoding
`'auto'` (default) | string

Character encoding, specified as a string. Common encodings include `'US-ASCII'`, `'UTF-8'`, `'latin1'`, `'Shift_JS'`, and `'ISO-8859-1'`.

`'UserAgent'` — User agent identification
`['MATLAB ' version]` (default) | string

User agent identification, specified as a string indicating the client user agent.

`'Timeout'` — Timeout connection duration
5 (default) | positive numeric scalar

Timeout connection duration in seconds, specified as a positive numeric scalar. Set to `Inf` to disable timeouts.

`'Username'` — User identifier
`''` (default) | string

User identifier, specified as a string for basic HTTP authentication (no encryption).

`'Password'` — User authentication password
`''` (default) | string

User authentication password, specified as a string for basic HTTP authentication (no encryption). If you display a `weboptions` object with the `Password` property set, the value is displayed as a string containing '*'. However, the object stores the value of the `Password` property as plain text.

`'KeyName'` — Name of key
`''` (default) | string

Name of a key, specified as a string. `KeyName` is an additional name to add to the HTTP request header. For example, `KeyName` can be a web service API key name.

Example: `weboptions('KeyName','duration','KeyValue',7)` creates a `weboptions` object that contains a key name, `duration`, defined by a web service.

`'KeyValue'` — Value of key

`''` (default) | string | numeric | logical

Value of a key, specified as a string, numeric, or logical value to add to the HTTP request header. `KeyValue` is the value of a key specified by `KeyName`.

Example: `weboptions('KeyName','duration','KeyValue',7)` creates a `weboptions` object that contains a key value, `7`, paired with a key name, `duration`.

`'ContentType'` — Content type

`'auto'` (default) | string

Content type, specified as a string. You can define the `ContentType` property of a `weboptions` object, and pass the object as an input argument to `webread`. Then `webread` returns data as that type of content. The table lists the valid content types.

`'auto'`	Output type automatically determined based on content type (default).

`'text'`	String for content types:
	
	`text/plain`
	`text/html`
	`text/xml`
	`application/xml`
	`application/javascript`
	`application/x-javascript`
	`application/x-www-form-urlencoded`
	
	If a web service returns a MATLAB file with a `.m` extension, the function returns its content as a string.

`'image'`	Numeric or logical matrix for `image/format` content. If the first output argument is an indexed image, the second output argument is the colormap, and the third output argument is the alpha channel.
`'audio'`	Numeric matrix for `audio/format` content with numeric scalar sampling rate as a second output argument.
`'binary'`	`uint8` column vector for binary content (that is, content not to be treated as type `char`).
`'table'`	Scalar table object for spreadsheet and CSV (`text/csv`) content.

`'json'`	`char`, numeric, logical, structure, or cell array, for `application/json` content.
`'xmldom'`	Java Document Object Model (DOM) node for `text/xml` or `application/xml` content. If not specified, the function returns XML content as a string.
`'raw'`	`char` column vector for `'text'`, `'xmldom'`, and `'json'` content. The function returns any other content type as a `uint8` column vector.

Example: `weboptions('ContentType','text')` creates a `weboptions` object that instructs `webread` to return text, JSON, or XML content as a character vector.

`'ContentReader'` — Content reader

`[ ]` (default) | function handle

Content reader, specified as a function handle. You can create a `weboptions` object with `ContentReader` specified, and pass the object as an input argument to `webread`. Then `webread` downloads data from a web service and reads the data with the function specified by the function handle. `webread` ignores `ContentType` when `ContentReader` is specified.

Example: `weboptions('ContentReader',@readtable)` creates a `weboptions` object that instructs `webread` to use `readtable` to read content as a table.

`'MediaType'` — Media type

`'application/x-www-form-urlencoded'` (default) | string

Media type, specified as a string. See Internet Media Types for a complete list of media types.

Example: `weboptions('MediaType','application/json')` creates a `weboptions` object that instructs `webwrite` to encode string data as JSON to post it to a web service.

`'RequestMethod'` — HTTP request method

`'auto'` (default) | `'get'` | `'post'`

HTTP request method, specified as `'auto'`, `'get'`, or `'post'`. `RequestMethod` can be set to `'get'` to indicate the HTTP GET method, `'post'` to indicate the HTTP POST method, or `'auto'`.

- `webread` and `websave` use the HTTP GET method when the `RequestMethod` property of an input `weboptions` object is set to `'auto'`.

- `webwrite` uses the HTTP POST method when the `RequestMethod` property of an input `weboptions` object is set to `'auto'`.

Example: `weboptions('RequestMethod','post')` creates a `weboptions` object that instructs `webread`, `websave`, or `webwrite` to use the HTTP POST method of a web service.

See Also

`webread` | `websave` | `webwrite`

Introduced in R2014b

webread

Read content from RESTful web service

Syntax

```
data = webread(url)
data = webread(url,QueryName,QueryValue)
data = webread( ___ ,options)

[data,colormap,alpha] = webread( ___ )

[data,Fs] = webread( ___ )
```

Description

`data = webread(url)` reads content from the web service specified by `url` and returns the content in `data`.

The web service provides a "RESTful API" on page 1-8916 that returns data in an internet media type format such as JSON, XML, image, or text.

`data = webread(url,QueryName,QueryValue)` appends additional web service query parameters to `url`, as specified by one or more pairs of `Name,Value` arguments.

`data = webread( ___ ,options)` adds other HTTP request options, specified by the `weboptions` object `options`. You can use this syntax with any of the input arguments of the previous syntaxes.

To read content with a function, specify the `ContentReader` property of `options` as a handle to the function. `webread` downloads data from a web service and reads the data with the specified function:

- If you specify a handle to a function that returns multiple output arguments, `webread` returns all output arguments.
- If you specify a handle to a function that returns no output argument (such as `@implay` for video files), `webread` returns no output argument.

`webread` supports HTTP GET and POST methods. To send an HTTP POST request, specify the `RequestMethod` property of `options` as `'post'`. Many web services provide both GET and POST methods to request data.

`[data,colormap,alpha] = webread( ___ )` reads an image from the web service specified by `url` and returns the image in `data`. You can use the previous syntaxes to return the image only. Use this syntax to return the colormap and alpha channels associated with the image.

`[data,Fs] = webread( ___ )` reads audio data from the web service specified by `url` and returns the audio data in `data`. You can use the previous syntaxes to return the audio data only. Use this syntax to return the sample rate of the audio data in hertz.

Examples

Read Image from Website

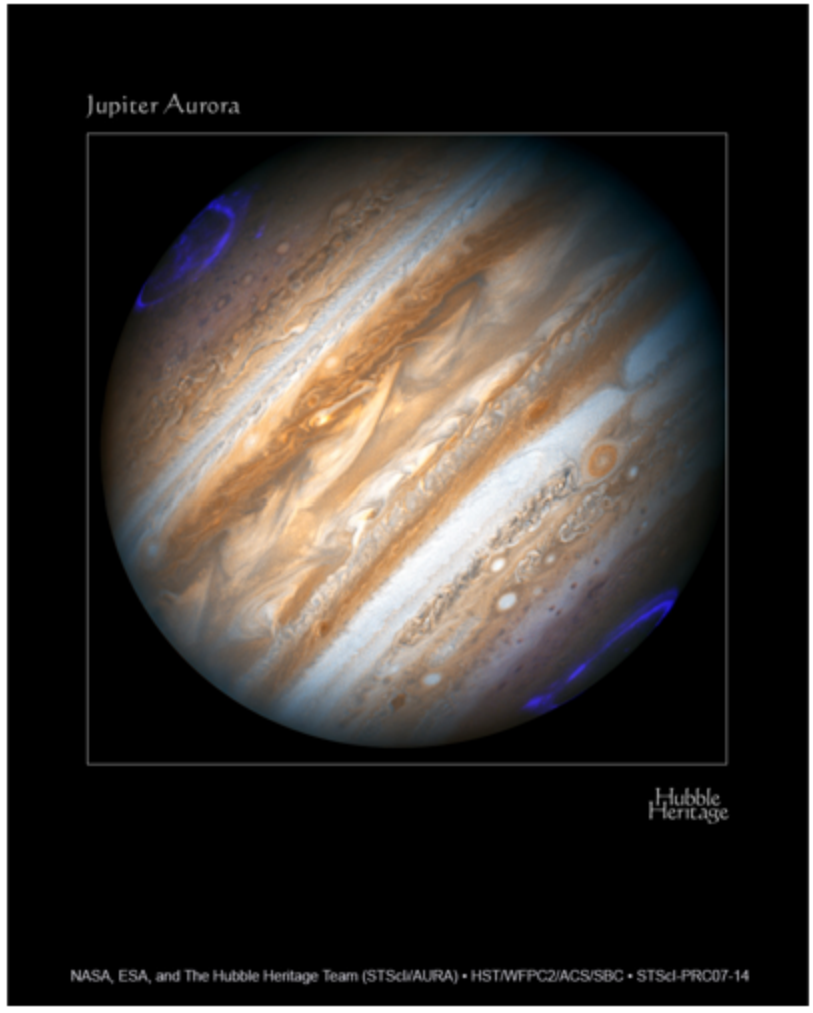
Read an image of Jupiter from the Hubble Heritage website.

```
url = 'http://heritage.stsci.edu/2007/14/images/p0714aa.jpg';
rgb = webread(url);
whos rgb
```

```
  Name          Size                 Bytes  Class    Attributes

  rgb        1000x800x3            2400000  uint8
```

Resize and display the image.

```
rgb = imresize(rgb,0.6);
imshow(rgb)
```


Jupiter Aurora
Hubble Heritage
NASA, ESA, and The Hubble Heritage Team (STScI/AURA) • HST/WFPC2/ACS/SBC • STScI-PRC07-14

Jupiter image courtesy of NASA, ESA, and the Hubble Heritage Team (STScI/AURA). (See Hubble Heritage Information Center for terms of use.)

Read Data from Web Service API

Read data from the World Bank Climate Data API. This API returns data in JSON format.

```
api = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/';
url = [api 'country/cru/tas/year/USA'];
S = webread(url)
```

```
S =

112x1 struct array with fields:

    year
    data
```

This API returns data as a JSON object. `webread` converts the JSON object to a structure array. Each structure contains the year and the average temperature in the USA for that year, in degrees Celsius.

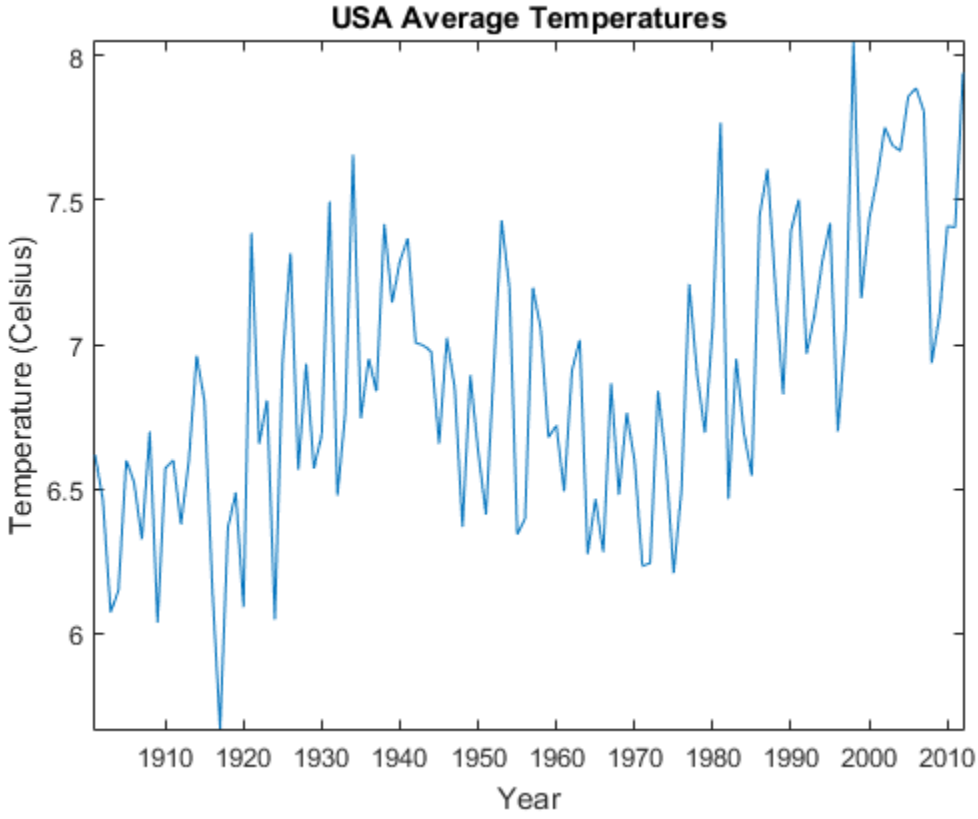
Display the temperature for the first year.

```
S(1)
```

```
ans =

    year: 1901
    data: 6.6187
```

Plot the average temperatures. Concatenate `S.year` and `S.data` into arrays and plot them.

```
year = [S.year];
data = [S.data];
plot(year,data)
xlabel('Year');
ylabel('Temperature (Celsius)');
title('USA Average Temperatures')
axis tight
```



API and data courtesy of the World Bank: Climate Data API. (See World Bank: Climate Data API for more information about the API, and World Bank: Terms of Use for terms of use.)

Specify Web Service Query Parameters

Specify web service query parameters with `webread`. Search the File Exchange for files uploaded within the past 7 days that contain the word *Simulink*. Use `QueryName,QueryValue` pairs to specify the `term` and `duration` query parameters. The File Exchange web service defines the `term` and `duration` query parameters, not the `webread` function. `webread` appends web service query parameters to the URL.

```
url = 'http://www.mathworks.com/matlabcentral/fileexchange/';
data = webread(url,'term','simulink','duration',7);
```

`webread` returns the HTML for the search result page as a character array.

Specify Additional Query Parameters

Specify additional web service query parameters with a `weboptions` object. Read data from the World Bank Climate Data API, but do not convert the JSON object to a structure array. Create a `weboptions` object and set its `ContentType` to `'text'`. The `webread` function converts the JSON object to a character array. Display the beginning of the character array.

```
api = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/';
url = [api 'country/cru/tas/year/USA'];
options = weboptions('ContentType','text');
data = webread(url,options);
data(1:62)
```

```
ans =

[{"year":1901,"data":6.6187487},{"year":1902,"data":6.4643273}
```

API and data courtesy of the World Bank: Climate Data API. (See World Bank: Climate Data API for more information about the API, and World Bank: Terms of Use for terms of use.)

Read Data with POST Request

Send an HTTP POST request to search File Exchange for files uploaded within the past 7 days that contain the word *Simulink*.

```
url = 'http://www.mathworks.com/matlabcentral/fileexchange/';
options = weboptions('RequestMethod','post');
data = webread(url,'term','simulink','duration',7,options);
```

Many web services provide a POST method for requesting data in addition to GET.

- “Download Data from Web Service”
- “Convert Data from Web Service”

Input Arguments

`url` — URL to web service

string

URL to a web service, specified as a string. The web service implements a RESTful interface. See “RESTful API” on page 1-8916 for more information.

Example: `webread('http://www.mathworks.com/matlabcentral')` reads the web page and returns its HTML as a string.

`QueryName,QueryValue` — Web service query parameters

`Name,Value` pairs

Web service query parameters, specified as pairs of `Name,Value` arguments. `QueryName` must be a string that specifies the name of a query parameter. `QueryValue` must be a string, numeric, or logical value that specifies the value of the query parameter. Numeric and logical values can be in arrays. The web service defines these `Name,Value` pairs.

Example: `webread('http://www.mathworks.com/matlabcentral/fileexchange/','term','webread','duration',7)` retrieves a list of files uploaded to the File Exchange within the past 7 days that contain the word `webread`. The File Exchange web service defines the `term` and `duration` query parameters.

`options` — Additional HTTP request options

`weboptions` object

Additional HTTP request options, specified as a `weboptions` object. See `weboptions` for all request options that are `weboptions` properties.

Output Arguments

`data` — Content from web service

scalar | array | structure | table

Content read from a web service, returned as a scalar, array, structure, or table.

You can define the `ContentType` property of a `weboptions` object, and pass the object as an input argument to `webread`. Then `webread` returns `data` as that type of content. The table lists the valid content types you can specify in a `weboptions` object.

`'auto'`	Output type automatically determined based on content type (default).
`'text'`	String for content types:

	`text/plain`
	`text/html`
	`text/xml`
	`application/xml`
		`application/javascript`
		`application/x-javascript`
		`application/x-www-form-urlencoded`

If a web service returns a MATLAB file with a `.m` extension, the function returns its content as a string.

`'image'`	Numeric or logical matrix for `image/format` content. If the first output argument is an indexed image, the second output argument is the colormap, and the third output argument is the alpha channel.
`'audio'`	Numeric matrix for `audio/format` content with numeric scalar sampling rate as a second output argument.
`'binary'`	`uint8` column vector for binary content (that is, content not to be treated as type `char`).
`'table'`	Scalar table object for spreadsheet and CSV (`text/csv`) content.
`'json'`	`char`, numeric, logical, structure, or cell array, for `application/json` content.
`'xmldom'`	Java Document Object Model (DOM) node for `text/xml` or `application/xml` content. If not specified, the function returns XML content as a string.
`'raw'`	`char` column vector for `'text'`, `'xmldom'`, and `'json'` content. The function returns any other content type as a `uint8` column vector.

`colormap` — Colormap associated with indexed image

numeric array

Colormap associated with an indexed image, returned as a numeric array.

`alpha` — Alpha channels associated with indexed image

numeric array

Alpha channels associated with an indexed image, returned as a numeric array.

`Fs` — Sample rate of audio data in hertz

positive numeric scalar

Sample rate of audio data in hertz, returned as a positive numeric scalar.

More About

RESTful API

REST means *representational state transfer*, a common architectural style for web services. RESTful APIs or interfaces provide standard HTTP methods such as GET, PUT, POST, or DELETE.

Tips

- This function supports only nonauthenticated and basic authentication types for use with your proxy server. To enable proxy support, run MATLAB with the Java JVM. To specify proxy server settings, see "Specify Proxy Server Settings for Connecting to the Internet".
- For HTTPS connections, this function verifies that the certificate domain matches the host name of the web service. This function does not verify the certificate chain.
- For HTTP POST requests, this function supports only the `application/x-www-form-urlencoded` media type. To send a POST request with content of any other internet media type, use `webwrite`.

Algorithms

JSON supports fewer data types than MATLAB. `webread` converts JSON data types to the MATLAB data types shown in the table.

JSON Data Type	MATLAB Data Type
Number	`double`
String	`char` array
Boolean	`logical`
Array, when all elements are of the same data type	Array
Array, when elements are of different data types	Cell array
Object. (In JSON, *object* means an unordered set of name-value pairs.)	Scalar structure
Array of objects, when all objects have the same set of field names	Structure array

JSON Data Type	MATLAB Data Type
Array of objects, when objects have different field names	Cell array of scalar structures

`webread` converts JSON object field names to MATLAB structure field names.

- Representational State Transfer
- JavaScript Object Notation (JSON)

See Also

`weboptions` | `websave` | `webwrite`

Introduced in R2014b

websave

Save content from RESTful web service to file

Syntax

```
outfilename = websave(filename,url)
outfilename = websave(filename,url,QueryName,QueryValue)
outfilename = websave( ___ ,options)
```

Description

`outfilename = websave(filename,url)` saves content from the web service specified by `url` to the file, `filename`. The full path to `filename` is returned as `outfilename`.

The web service provides a "RESTful API" on page 1-8921 that returns data in an internet media type format such as JSON, XML, image, or text.

`outfilename = websave(filename,url,QueryName,QueryValue)` appends additional web service query parameters to `url`, as specified by one or more pairs of `Name,Value` arguments.

`outfilename = websave( ___ ,options)` adds other HTTP request options, specified by the `weboptions` object `options`. You can use this syntax with any of the input arguments of the previous syntaxes.

`websave` supports HTTP GET and POST methods. To send an HTTP POST request, specify the `RequestMethod` property of `options` as `'post'`. Many web services provide both GET and POST methods to request data.

Examples

Save Image from Website

Save an image of Jupiter from the Hubble Heritage website.

```
url = 'http://heritage.stsci.edu/2007/14/images/p0714aa.jpg';
filename = 'jupiter_aurora.jpg';
outfilename = websave(filename,url)

outfilename =

C:\Libraries\Documents\jupiter_aurora.jpg
```

`websave` saves the image as a JPEG file, as specified by the Hubble web service, even when you give `filename` a different extension. (Jupiter image courtesy of NASA, ESA, and the Hubble Heritage Team (STScI/AURA). See Hubble Heritage Information Center for terms of use.)

Save Search Results to File

Save the results of a search of the File Exchange to an HTML file. Search File Exchange for files uploaded within the past 7 days that contain the word *Simulink*.

```
url = 'http://www.mathworks.com/matlabcentral/fileexchange/';
filename = 'simulink_search.html';
outfilename = websave(filename,url,'term','simulink','duration',7)

outfilename =

C:\Libraries\Documents\simulink_search.html
```

The File Exchange web service defines the `term` and `duration` query parameters, not `websave`.

Display the HTML file in a web browser.

```
web(outfilename)
```

Save Data to File

Save sunspot data from the National Geophysical Data Center (NGDC) to an ASCII file. Use a `weboptions` object to set the timeout value to `Inf` so that the connection does not time out.

```
api = 'http://www.ngdc.noaa.gov/stp/space-weather/';
url = [api 'solar-data/solar-indices/sunspot-numbers/' ...
       'american/lists/list_aavso-arssn_yearly.txt'];
filename = 'sunspots_annual.txt';
options = weboptions('Timeout',Inf);
```

```
outfilename = websave(filename,url,options)

outfilename =

C:\Libraries\Documents\sunspots_annual.txt
```

Aggregated data and web service courtesy of the NGDC. Sunspot data courtesy of the American Association of Variable Star Observers (AAVSO), originally published in AAVSO Sunspot Counts: 1943-2013, AAVSO Solar Section (R. Howe, Chair). (See NGDC Privacy Policy, Disclaimer, and Copyright for NGDC terms of use, and AAVSO Solar Section for AAVSO terms of use.)

Save Data with POST Request

Send an HTTP POST request to search File Exchange for files uploaded within the past 7 days that contain the word *Simulink*. Save the results of the search to an HTML file.

```
url = 'http://www.mathworks.com/matlabcentral/fileexchange/';
filename = 'simulink_search.html';
options = weboptions('RequestMethod','post');
outfilename = websave(filename,url,'term','simulink','duration',7,options);
```

Many web services provide a POST method for requesting data in addition to GET.

Input Arguments

`filename` — Name of file to save content to

string

Name of file to save content to, specified as a string. `websave` saves the content as is. `websave` ignores `options.ContentType` and `options.ContentReader` even if these properties are set.

Example: `websave('matlabcentral.html','http://www.mathworks.com/matlabcentral')` reads the web page and saves its HTML to the file `matlabcentral.html`.

`url` — URL to web service

string

URL to a web service, specified as a string. The web service implements a RESTful interface. See “RESTful API” on page 1-8921 for more information.

`QueryName,QueryValue` — Web service query parameters

`Name,Value` pairs

Web service query parameters, specified as pairs of `Name,Value` arguments. `QueryName` must be a string that specifies the name of a query parameter. `QueryValue` must be a string, numeric, or logical value that specifies the value of the query parameter. Numeric and logical values can be in arrays. The web service defines these `Name,Value` pairs.

Example: `websave('webread_search.html','http://www.mathworks.com/matlabcentral/fileexchange/','term','simulink','duration',7)` retrieves a list of files uploaded to the File Exchange within the past 7 days that contain the word *`simulink`* and saves the HTML to file.

`options` — Additional HTTP request options

`weboptions` object

Additional HTTP request options, specified as a `weboptions` object. See `weboptions` for all request options that are `weboptions` properties.

More About

RESTful API

REST means *representational state transfer*, a common architectural style for web services. RESTful APIs or interfaces provide standard HTTP methods such as GET, PUT, POST, or DELETE.

Tips

- This function supports only nonauthenticated and basic authentication types for use with your proxy server. To enable proxy support, run MATLAB with the Java JVM. To specify proxy server settings, see “Specify Proxy Server Settings for Connecting to the Internet”.
- For HTTPS connections, this function verifies that the certificate domain matches the host name of the web service. This function does not verify the certificate chain.
- For HTTP POST requests, this function supports only the `application/x-www-form-urlencoded` media type. To send a POST request with content of any other internet media type, use `webwrite`.
- Representational State Transfer

- JavaScript Object Notation (JSON)

See Also

`weboptions` | `webread` | `webwrite`

Introduced in R2014b

webwrite

Write data to RESTful web service

Syntax

```
response = webwrite(url,PostName,PostValue)
response = webwrite(url,data)
response = webwrite( ___ ,options)
```

Description

`response = webwrite(url,PostName,PostValue)` writes content to the web service specified by `url` and returns the response in `response`. The input arguments `PostName,PostValue` specify the content as one or more pairs of `Name,Value` arguments. `webwrite` form-encodes `PostName,PostValue` in the body of an HTTP POST request to the web service. The web service defines `response`.

The web service provides a "RESTful API" on page 1-8927 that returns data in an internet media type format such as JSON, XML, image, or text.

`response = webwrite(url,data)` writes content to the web service specified by `url` and returns the response in `response`. The input argument `data` specifies the content as a form-encoded string. `webwrite` puts `data` in the body of an HTTP POST request to the web service. The web service defines `response`.

`response = webwrite( ___ ,options)` adds other HTTP request options, specified by the `weboptions` object `options`. You can use this syntax with any of the input arguments of the previous syntaxes.

To write content as an internet media type other than a form-encoded string (`'application/x-www-form-urlencoded'`), specify the `MediaType` property of `options`.

To request data with an HTTP POST request and read the response with a function, specify the `ContentReader` property of `options` as a handle to the function. If you specify a handle to a function that returns multiple output arguments, `webwrite` returns all output arguments.

Examples

Write Data to Web Service

Write a number to the ThingSpeak server. (To run this code, create a ThingSpeak account. Use the Write API key and Channel ID from your ThingSpeak account. The default field name is `'field1'`.)

```
thingSpeakURL = 'http://api.thingspeak.com/';
thingSpeakWriteURL = [thingSpeakURL 'update'];
writeApiKey = 'Your Write API Key';
fieldName = 'field1';
fieldValue = 42;
response = webwrite(thingSpeakWriteURL,'api_key',writeApiKey,fieldName,fieldValue)
```

`response` is `1` if this call to `webwrite` is the first update to your ThingSpeak channel.

Read back the number you wrote to your channel. ThingSpeak provides a different URL to get the last entry to your channel. Your Channel ID is part of the URL.

```
channelID = num2str(Your Channel ID);
thingSpeakReadURL = [thingSpeakURL 'channels/' channelID '/fields/' fieldName '/last'];
data = webread(thingSpeakReadURL,'api_key',writeApiKey)
```

```
data =

42
```

Write Form-Encoded String

Write a number to the ThingSpeak server. Encode your Write API Key and the number as a form-encoded string. (To run this code, create a ThingSpeak account. Use the Write API key and Channel ID from your ThingSpeak account. The default field name is `'field1'`.)

```
thingSpeakURL = 'http://api.thingspeak.com/';
thingSpeakWriteURL = [thingSpeakURL 'update'];
writeApiKey = 'Your Write API Key';
data = 42;
data = num2str(data);
data = ['api_key=',writeApiKey,'&field1=',data];
response = webwrite(thingSpeakWriteURL,data)
```

`response` is `1` if this call to `webwrite` is the first update to your ThingSpeak channel.

Read back the number you wrote to your channel. ThingSpeak provides a different URL to get the last entry to your channel. Your Channel ID is part of the URL.

```
channelID = num2str(Your Channel ID);
thingSpeakReadURL = [thingSpeakURL 'channels/' channelID '/fields/field1/last'];
data = webread(thingSpeakReadURL,'api_key',writeApiKey)
```

```
data =

42
```

Write JSON Object

Write a number to the ThingSpeak server. Create a structure whose fields are your Write API Key and the number. `webwrite` converts a structure to a JSON object when you use a `weboptions` object to specify the media type as `'application/json'`. (To run this code, create a ThingSpeak account. Use the Write API key and Channel ID from your ThingSpeak account. The default field name is `'field1'`.)

```
thingSpeakURL = 'http://api.thingspeak.com/update.json';
writeApiKey = 'Your Write API Key';
data = 42;
data = struct('api_key',writeApiKey,'field1',data);
options = weboptions('MediaType','application/json');
response = webwrite(thingSpeakURL,data,options)
```

```
response =

    channel_id: Your Channel ID
        field1: 42
        field2: []
        field3: []
        field4: []
        field5: []
        field6: []
        field7: []
        field8: []
    created_at: '2014-11-14T20:08:14Z'
      entry_id: 1
        status: []
      latitude: []
     longitude: []
     elevation: []
      location: []
```

As a response `webwrite` receives a JSON object that contains the number you wrote to your ThingSpeak channel. `webwrite` converts the JSON object and returns it as a structure in `response`.

Input Arguments

`url` — URL to web service

string

URL to a web service, specified as a string. The web service implements a RESTful interface. See "RESTful API" on page 1-8927 for more information.

`PostName,PostValue` — Web service post parameters

`Name,Value` pairs

Web service post parameters, specified as pairs of `Name,Value` arguments. `PostName` must be a string that specifies the name of a post parameter. `PostValue` must be a string, numeric, or logical value that specifies the value of the post parameter. Numeric and logical values can be in arrays. The web service defines these `Name,Value` pairs. `webwrite` encodes the `Name,Value` pairs as a form-encoded string in the body of an HTTP POST request and sets the content type to `application/x-www-form-urlencoded` by default.

Example: `webwrite('http://www.mathworks.com/matlabcentral/fileexchange/','term','webwrite','duration',7)` retrieves a list of files uploaded to the File Exchange within the past 7 days that contain the word `webwrite`. The File Exchange web service defines the `term` and `duration` parameters.

`data` — Data to write to web service

string

Data to write to a web service, specified as a form-encoded string. The web service implements a RESTful interface. `webwrite` puts `data` in the body of an HTTP POST request and sets the media type to `application/x-www-form-urlencoded` by default.

If `data` is not form-encoded, set the `MediaType` property of the `options` input argument to a different media type. See Internet Media Types for a complete list of media types.

Example: `webwrite('http://www.mathworks.com/matlabcentral/fileexchange/','term=webwrite&duration=7')` retrieves a list of files uploaded

to the File Exchange within the past 7 days that contain the word `webwrite`. The File Exchange web service defines the `term` and `duration` parameters.

`options` — Additional HTTP request options

`weboptions` object

Additional HTTP request options, specified as a `weboptions` object. See `weboptions` for all request options that are `weboptions` properties.

Output Arguments

`response` — Response from web service

scalar | array | structure | table

Response from a web service, returned as a scalar, array, structure, or table.

More About

RESTful API

REST means *representational state transfer*, a common architectural style for web services. RESTful APIs or interfaces provide standard HTTP methods such as GET, PUT, POST, or DELETE.

Tips

- `webwrite` writes `PostName,PostValue` input arguments as form-encoded strings. If you also specify the `options` input argument, then its `MediaType` property must be `'application/x-www-form-urlencoded'`.
- This function supports only nonauthenticated and basic authentication types for use with your proxy server. To enable proxy support, run MATLAB with the Java JVM. To specify proxy server settings, see “Specify Proxy Server Settings for Connecting to the Internet”.
- For HTTPS connections, this function verifies that the certificate domain matches the host name of the web service. This function does not verify the certificate chain.

- Representational State Transfer
- JavaScript Object Notation (JSON)

See Also

`weboptions` | `webread` | `websave`

Introduced in R2015a

week

Week number

Syntax

```
w = week(t)
w = week(t,weekType)
```

Description

`w = week(t)` returns the week-of-year numbers of the datetime values in `t`. The `m` output is a `double` array the same size as `t` and contains integer values from 1 to 53.

`w = week(t,weekType)` returns the type of week number specified by `weekType`.

Examples

Find Week of Year Numbers of Dates

```
t = datetime(2013,05,31):calmonths(3):datetime(2014,06,15)
```

```
t =

   31-May-2013    31-Aug-2013    30-Nov-2013    28-Feb-2014    31-May-2014
```

```
w = week(t)
```

```
w =

    22    35    48     9    22
```

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`weekType` — Type of week values

`'weekofyear'` (default) | `'weekofmonth'`

Type of week values, specified as either `'weekofyear'` or `'weekofmonth'`.

- If `weekType` is `'weekofyear'`, then `month` returns the week-of-year number. Week 1 is the week in which January 1 falls, even if fewer than 4 days of that week fall in the same year.
- If `weekType` is `'weekofmonth'`, then `month` returns the week-of-month number. Week 1 in a month is defined as the week in which the first day of the month falls, even if fewer than 4 days of that week fall in the same month.

See Also

`day` | `month` | `quarter` | `year`

Introduced in R2014b

weekday

Day of week

Syntax

```
DayNumber = weekday(D)
[DayNumber,DayName] = weekday(D)
[DayNumber,DayName] = weekday(D,DayForm)
[DayNumber,DayName] = weekday(D,language)
[DayNumber,DayName] = weekday(D,DayForm,language)
```

Description

`DayNumber = weekday(D)` returns a number representing the day of the week for each element in `D`.

`[DayNumber,DayName] = weekday(D)` additionally returns abbreviated English names for the day of the week, in `DayName`.

`[DayNumber,DayName] = weekday(D,DayForm)` returns the name for the day of the week in the format specified by `DayForm`, in US English.

`[DayNumber,DayName] = weekday(D,language)` returns the abbreviated name for the day of the week in the language of the locale specified in `language`.

`[DayNumber,DayName] = weekday(D,DayForm,language)` returns the name for the day of the week in the specified format and in the language of the specified locale. You can specify `DayForm` and `language` in either order.

Examples

Return Day of Week of a Date String

Determine the day of the week of December 21, 2012.

```
D = '21-Dec-2012';
[DayNumber,DayName] = weekday(D)
```

```
DayNumber =

     6


DayName =

Fri
```

December 21, 2012 falls on a Friday.

Return Full Day Names of Multiple Date Numbers

Return the full name of the day of the week for a vector of serial date numbers.

```
D = [734999;735015];
DayForm = 'long';
[DayNumber,DayName] = weekday(D,DayForm)
```

```
DayNumber =

     5
     7


DayName =

Thursday
Saturday
```

Return Full Day Names in Local Language

Return day names in U.S. English using the `language` input argument.

```
D = 728647;
DayForm = 'long';
language = 'en_US';
[DayNumber, DayName] = weekday(D,DayForm,language)
```

```
DayNumber =

     2


DayName =
```

```
Monday
```

In U.S. English, the name of the day of the week is `Monday`.

Return day names in the language of the current locale.

```
language = 'local';
[DayNumber, DayName] = weekday(D,DayForm,language)
```

```
DayNumber =

     2


DayName =

Lundi
```

In a French locale, the name of the day of the week is `Lundi`.

Return Day of Week of a Date String in Custom Format

Determine the day of the week for a date specified in the format `mmm.dd.yyyy`. Call `datenum` inside of `weekday` to specify the format of the input date string.

```
[DayNumber,DayName] = weekday(datenum('Dec.21.2012','mmmm.dd.yyyy'))
```

```
DayNumber =

     6


DayName =

Fri
```

Input Arguments

`D` — Serial date numbers or date strings

vector | matrix | string | cell array of strings | character array

Serial date numbers or date strings. Date numbers can be specified as a vector or matrix. Date strings can be specified as a single string, a cell array of strings, or a character array. If `D` is a cell array of strings, it must be 1-by-`n` or `n`-by-1.

If D is a single date string, a cell array of date strings, or a character array of date strings, the date strings can be in one of the following formats.

Date String Format	Example
dd-mmm-yyyy	01-Mar-2000
mm/dd/yyyy	03/01/2000
yyyy-mm-dd	2000-03-01

For date strings in other formats, first convert them to serial date numbers using the datenum function, before passing them to weekday.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char | cell

DayForm — Format of output day names

'short' (default) | 'long'

Format of the output day names, specified as one of the following strings.

DayForm	Format of DayName Names	Example
'short'	Abbreviated name	Mon
'long'	Full name	Monday

language — Output language of day names

'en_US' (default) | 'local'

Output language of day names in DayName, specified as one of the following strings.

language	Description
'en_US'	US English
'local'	Language of the current locale

Output Arguments

DayNumber — Value representing day of week

array of integers in the range [1,7]

Value representing the day of the week, returned as an array of integers in the range `[1,7]`, where `1` represents Sunday, and `7` represents Saturday.

- If input `D` is a numeric array, then the size of `DayNumber` is equivalent to the size of `D`.
- If input `D` is a cell array of strings, then `DayNumber` is an `m`-by-1 vector, where `m` is equivalent to the length of `D`.

`DayName` — Name of day of week

character array

Name of the day of the week, returned as a character array. The content of `DayName` depends on `DayForm`.

- If `DayForm` is `'short'`, then `DayName` contains an abbreviated name (for example, `Tues`).
- If `DayForm` is `'long'`, then `DayName` contains the full name of the weekday (for example, `Tuesday`).

`DayName` is `m`-by-`n`, where `m` is the number of serial date numbers or date strings in `D`.

See Also

`datenum` | `datevec` | `eomday`

Introduced before R2006a

what

List MATLAB files in folder

Syntax

```
what
what folderName
what className
what packageName
s = what('folderName')
```

Description

`what` lists the path for the current folder, and lists all files and folders relevant to MATLAB found in the current folder. Files listed are M, MAT, MEX, MDL, SLX, and P-files. Folders listed are all class and package folders.

`what folderName` lists path, file, and folder information for *`folderName`*.

`what className` lists path, file, and folder information for method folder `@`*`className`*. For example, `what cfit` lists the MATLAB files and folders in `toolbox/curvefit/curvefit/@cfit`.

`what packageName` lists path, file, and folder information for package folder `+`*`packageName`*. For example, `what commsrc` lists the MATLAB files and folders in `toolbox/comm/comm/+commsrc`.

`s = what('folderName')` returns the results in a structure array with the fields shown in the following table.

Field	Description
`path`	Path to folder
`m`	Cell array of MATLAB program file names
`mat`	Cell array of MAT-file names

Field	Description
`mex`	Cell array of MEX-file names
`mdl`	Cell array of MDL-file names
`slx`	Cell array of SLX-file names
`p`	Cell array of P-file names
`classes`	Cell array of class folders
`packages`	Cell array of package folders

Examples

List the MATLAB files and folders in `C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo\+audiovideo`, where *`Rnnnn`* represents the folder for the MATLAB release, for example, R2012b:

```
what audiovideo

MATLAB Code files in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo\+audiovideo

FileFormatInfo

Packages in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo\+audiovideo

internal  writer


MATLAB Code files in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

Contents            aufinfo             mmcompinfo          wavplay
audiodevinfo        auread              mmfileinfo          wavread
audioinfo           auwrite             movie2avi           wavrecord
audioplayerreg      avgate              mu2lin              wavwrite
audioread           avifinfo            prefspanel
audiorecorderreg    aviinfo             sound
audiouniquename     aviread             soundsc
audiowrite          lin2mu              wavfinfo

MAT-files in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

chirp         handel          splat
gong          laughter        train

Classes in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

VideoReader    audioplayer    avifile
VideoWriter    audiorecorder  mmreader

Packages in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

audiovideo
```

Obtain a structure array containing the file and folder names in `toolbox/matlab/codetools` that are relevant to MATLAB, where *Rnnnn* represents the folder for the MATLAB release, for example, R2012a:

```
s = what('codetools')

s =

        path: 'C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\codetools'
           m: {77x1 cell}
         mat: {0x1 cell}
         mex: {0x1 cell}
         mdl: {0x1 cell}
         slx: {0x1 cell}
           p: {0x1 cell}
     classes: {2x1 cell}
    packages: {3x1 cell}
```

Find the supporting files for one of the packages in the Communications System Toolbox™ product:

```
p1 = what('comm');
p1.packages
ans =

    'gpu'
    'internal'


ans =

    'internal'.
.
.
p2 = what('commsrc');
p2.m
ans =
    'abstractJitter.m'
    'abstractPulse.m'
    'combinedjitter.m'
    'diracjitter.m'
    'periodicjitter.m'
    'randomjitter.m'
```

See Also

`dir` | `exist` | `lookfor` | `ls` | `which` | `who`

Introduced before R2006a

whatsnew

Release Notes

Note: `whatsnew` will be removed in a future release.

Syntax

```
whatsnew
```

Description

`whatsnew` displays the MATLAB Release Notes in the Help browser, presenting information about new features, problems from previous releases that have been fixed in the current release, and compatibility issues.

See Also

`help` | `version`

Introduced before R2006a

which

Locate functions and files

Syntax

```
which item
which fun1 in fun2

which ___ -all

str = which(item)
str = which(fun1,'in',fun2)

str = which( ___ ,'-all')
```

Description

`which item` displays the full path for `item`.

- If `item` is a MATLAB function in an M or P file, or a Simulink model in an SLX or MDL file, then `which` displays the full path for the corresponding file. `item` must be on the MATLAB path.
- If `item` is a method in a loaded Java class, then `which` displays the package, class, and method name for that method.
- If `item` is a workspace variable, then `which` displays a message identifying `item` as a variable.
- If `item` is a file name including the extension, and it is in the current working folder or on the MATLAB path. then `which` displays the full path of `item`.

If `item` is an overloaded function or method, then `which item` returns only the path of the first function or method found.

`which fun1 in fun2` displays the path to function `fun1` that is called by file `fun2`. Use this syntax to determine whether a local function is being called instead of a function on the path. This syntax does not locate nested functions.

`which ___ -all` displays the paths to all items on the MATLAB path with the requested name. Such items include methods of instantiated classes. You can use `-all` with the input arguments of any of the previous syntaxes.

`str = which(item)` returns the full path for `item` in the string, `str`.

`str = which(fun1,'in',fun2)` returns the path to function `fun1` that is called by file `fun2`. Use this syntax to determine whether a local function is being called instead of a function on the path. This syntax does not locate nested functions.

`str = which(___,'-all')` returns the results of `which` in the string or cell array of strings, `str`. You can use this syntax with any of the input arguments in the previous syntax group.

Examples

Locate MATLAB Function

Locate the `pinv` function.

```
which pinv
```

```
matlabroot\toolbox\matlab\matfun\pinv.m
```

`pinv` is in the `matfun` folder of MATLAB.

You also can use function syntax to return the path to a string, `str`. When using the function form of `which`, enclose all input strings in single quotes.

```
str = which('pinv');
```

Locate Method in a Loaded Java Class

Create an instance of the Java class. This loads the class into MATLAB.

```
myDate = java.util.Date;
```

Locate the `setMonth` method.

```
which setMonth
```

```
setMonth is a Java method  % java.util.Date method
```

Locate Private Function

Find the `orthog` function in a private folder.

```
which private/orthog
```

```
matlabroot\toolbox\matlab\elmat\private\orthog.m  % Private to elmat
```

MATLAB displays the path for `orthog.m` in the `/private` subfolder of `toolbox/matlab/elmat`.

Determine if Local Function is Called

Determine which `parseargs` function is called by `area.m`.

```
which parseargs in area
```

```
matlabroot\toolbox\matlab\specgraph\area.m (parseargs)  % Subfunction of area
```

You also can use function syntax to return the path to a string, `str`. When using the function form of `which`, enclose all input strings in single quotes.

```
str = which('parseargs','in','area');
```

Locate Function Invoked with Given Input Arguments

Suppose you have a `matlab.io.MatFile` object that corresponds to the example MAT-file `'topography.mat'`:

```
matObj = matfile('topography.mat');
```

Display the path of the implementation of `who` that is invoked when called with the input argument `(matObj)`.

```
which who(matObj)
```

```
matlabroot\toolbox\matlab\iofun\+matlab\+io\MatFile.m  % matlab.io.MatFile method
```

Store the result to a string.

```
str = which('who(matObj)')
```

```
str =
matlabroot\toolbox\matlab\iofun\+matlab\+io\MatFile.m
```

If you do not specify the input argument `(matObj)`, then `which` returns only the path of the first function or method found.

```
which who
```

```
built-in (matlabroot\\toolbox\matlab\general\who)
```

Locate All Items with Given Name

Display the paths to all items on the MATLAB path with the name `fopen`.

```
which fopen -all
```

```
built-in (matlabroot\toolbox\matlab\iofun\fopen)
matlabroot\toolbox\matlab\iofun\@serial\fopen.m            % serial method
matlabroot\toolbox\shared\instrument\@icinterface\fopen.m  % icinterface method
matlabroot\toolbox\instrument\instrument\@i2c\fopen.m      % i2c method
```

Return Path Names as Strings

Return the results of `which` in the string `str`.

You must use the function form of `which`, enclosing all arguments in parentheses and single quotes.

```
str = which('private/stradd','-all');
whos str
```

```
  Name      Size            Bytes  Class    Attributes

  str       2x1               650  cell
```

Input Arguments

`item` — Function or file to locate

string

Function or file to locate, specified as a string. When using the function form of `which`, enclose all input strings in single quotes. `item` can be in one of the following forms.

Form of the `item` Input	Path to Display
`fun`	Display full path for `fun`, which can be a MATLAB function, Simulink model, workspace variable, method

Form of the `item` Input	Path to Display
	in a loaded Java class, or file name that includes the file extension. To display the path for a file that has no file extension, type `which file.` (The period following the filename is required). Use `exist` to check for the existence of files anywhere else.
`private/fun`	Limit the search to private functions named `fun`. For example, `which private/orthog` or `which('private/orthog')` displays the path for `orthog.m` in the `/private` subfolder of the parent folder.
`fun(a1,...,an)`	Display the path to the implementation of function `fun` which would be invoked if called with the input arguments `a1,...,an`. Use this syntax to query overloaded functions. See the Example, "Locate Function Invoked with Given Input Arguments" on page 1-8943.

`fun1` — Function to locate

string

Function to locate, specified as a string. `fun1` can be the name of a function, or it can be in the form `fun(a1,...,an)`. For more information about the form, `fun(a1,...,an)`, see "Locate Function Invoked with Given Input Arguments" on page 1-8943.

When using the function form of `which`, enclose all input strings in single quotes, for example, `which('myfun1','in','myfun2')`.

`fun2` — Calling file

string

Calling file, specified as a string. `fun2` can be the name of a file, or it can be in the form `fun(a1,...,an)`. For more information about the form, `fun(a1,...,an)`, see "Locate Function Invoked with Given Input Arguments" on page 1-8943.

When using the function form of `which`, enclose all input strings in single quotes, for example, `which('myfun1','in','myfun2')`.

Output Arguments

`str` — Function or file location

string | cell array of strings

Function or file location, returned as a string, or returned as a cell array of strings if you use `'-all'`.

- If `item` is a workspace variable, then `str` is the string `'variable'`.
- If `str` is a cell array of strings, then each row of `str` identifies a function. The functions are in order of precedence, unless they are shadowed. Among shadowed functions, you should not rely on the order of the functions in `str`. To determine if a function is shadowed, call `which` using command syntax. For example, `which test -all` displays the paths to all functions with the name `test`. Shadowed functions are indicated by the comment, `% Shadowed`.

Limitations

- When the class is not loaded, `which` only finds methods if they are defined in separate files in an @-folder and are not in any packages.

More About

Tips

- For more information about how MATLAB uses scope and precedence when calling a function, see “Function Precedence Order”.

See Also

`dir` | `doc` | `exist` | `fileparts` | `lookfor` | `mfilename` | `path` | `type` | `what` | `who`

Introduced before R2006a

while

Repeat execution of statements while condition is true

Syntax

```
while expression
    statements
end
```

Description

`while` *`expression`*`,` *`statements`*`, end` evaluates an expression, and repeats the execution of a group of statements in a loop while the expression is true. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

Examples

Repeat Statements Until Expression Is False

Use a `while` loop to calculate `factorial(10)`.

```
n = 10;
f = n;
while n > 1
    n = n-1;
    f = f*n;
end
disp(['n! = ' num2str(f)])
```

```
n! = 3628800
```

Skip to Next Loop Iteration

Count the number of lines of code in the file `magic.m`. Skip blank lines and comments using a `continue` statement. `continue` skips the remaining instructions in the `while` loop and begins the next iteration.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
count
fclose(fid);
```

```
count =

    31
```

Exit Loop Before Expression Is False

Sum a sequence of random numbers until the next random number is greater than an upper limit. Then, exit the loop using a `break` statement.

```
limit = 0.8;
s = 0;

while 1
    tmp = rand;
    if tmp > limit
        break
    end
    s = s + tmp;
end
```

More About

Expression

An expression can include relational operators (such as < or ==) and logical operators (such as &&, ||, or ~). Use the logical operators `and` and `or` to create compound expressions. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Within the conditional expression of a `while...end` block, logical operators `&` and `|` behave as short-circuit operators. This behavior is the same as `&&` and `||`, respectively. Since `&&` and `||` consistently short-circuit in conditional expressions and statements, it is good practice to use `&&` and `||` instead of `&` and `|` within the expression. For example,

```
x = 42;
while exist('myfunction.m','file') && (myfunction(x) >= pi)
    disp('Expressions are true')
    break
end
```

The first part of the expression evaluates to false. Therefore, MATLAB does not need to evaluate the second part of the expression, which would result in an undefined function error.

Tips

- If you inadvertently create an infinite loop (that is, a loop that never ends on its own), stop execution of the loop by pressing **Ctrl+C**.
- If the conditional expression evaluates to a matrix, MATLAB evaluates the statements only if all elements in the matrix are true (nonzero). To execute statements if any element is true, wrap the expression in the `any` function.
- To programmatically exit the loop, use a `break` statement. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.
- When nesting a number of `while` statements, each `while` statement requires an `end` keyword.
- “Relational Operators”

See Also

`break` | `continue` | `end` | `for` | `if` | `Logical Operators: Short Circuit` | `return` | `switch`

Introduced before R2006a

whitebg

Change axes background color

Syntax

```
whitebg
whitebg(fig)
whitebg(ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig)
```

Description

`whitebg` complements the colors in the current figure.

`whitebg(fig)` complements colors in all figures specified in the vector `fig`.

`whitebg(ColorSpec)` and `whitebg(fig, ColorSpec)` change the color of the axes, which are children of the figure, to the color specified by `ColorSpec`. Without a figure specification, `whitebg` or `whitebg(ColorSpec)` affects the current figure and the root's default properties so subsequent plots and new figures use the new colors.

`whitebg(fig, ColorSpec)` sets the default axes background color of the figures in the vector `fig` to the color specified by `ColorSpec`. Other axes properties and the figure background color can change as well so that graphs maintain adequate contrast. `ColorSpec` can be a 1-by-3 RGB color or a color string such as `'white'` or `'w'`.

`whitebg(fig)` complements the colors of the objects in the specified figures. This syntax is typically used to toggle between black and white axes background colors, and is where `whitebg` gets its name. Include the root window handle `(0)` in `fig` to affect the default properties for new windows or for `clf reset`.

Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

More About

Tips

`whitebg` works best in cases where all the axes in the figure have the same background color.

`whitebg` changes the colors of the figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. `whitebg` sets the default properties on the root such that all subsequent figures use the new background color.

See Also

`ColorSpec` | `colordef`

Introduced before R2006a

who

List variables in workspace

Syntax

```
who
who(variables)
who(location)
who(variables,location)
c = who(variables,location)
```

Description

`who` lists in alphabetical order all variables in the currently active workspace.

`who(variables)` lists only the specified variables.

`who(location)` lists variables in the specified location: `'global'` for the global workspace, or `'-file'` for a MAT-file. For MAT-files, you must also include the file name as an input.

`who(variables,location)` lists the specified variables in the specified location. The `location` input can appear before or after `variables`.

`c = who(variables,location)` stores the names of the variables in cell array `c`. Specifying `variables` and `location` is optional.

Input Arguments

`variables`

Strings that specify the variables to list. Use one of these forms:

`var1, var2, ...`	List the specified variables. Use the `'*'` wildcard to match patterns. For example, `who('A*')` lists all variables that start with `A`.

`'-regexp', expressions`	List variables whose names match the specified regular expressions.

Default: `'*'` (all variables)

`location`

String that indicates whether to list variables from the global workspace or from a file:

`'global'`	Global workspace.
`'-file', filename`	MAT-file. The *`filename`* input can include the full, relative, or partial path.

Default: `''` (current workspace)

Output Arguments

`c`

Cell array of strings that correspond to each variable name.

Examples

Display information about variables in the current workspace whose names start with the letter `a`:

```
who a*
```

Show variables stored in MAT-file `durer.mat`:

```
who -file durer
```

This code returns:

```
Your variables are:

X        caption  map
```

Store the variable names from `durer.mat` in cell array `durerInfo`:

```
durerInfo = who('-file', 'durer');
```

Display the contents of cell array `durerInfo`:

```
for k=1:length(durerInfo)
   disp(durerInfo{k})
end
```

This code returns:

```
  X
  caption
  map
```

Suppose that a file `mydata.mat` contains variables with names that start with `java` and end with `Array`. Display information about those variables:

```
 whos -file mydata -regexp \<java.*Array\>
```

Call `who` within a nested function (`get_date`):

```
function who_demo
date_time = datestr(now);

[str pos] = textscan(date_time, '%s%s%s', ...
                     1, 'delimiter', '- :');
get_date(str);

   function get_date(d)
      day = d{1};
      mon = d{2};
      year = d{3};
      who
   end

end
```

When you run `who_demo`, the `who` function displays the variables by function workspace (although the name of the function does not appear in the output):

```
Your variables are:

d          mon        ans        pos
day        year       date_time  str
```

Alternatives

To view the variables in the workspace, use the Workspace browser. To view the contents of MAT-files, use the Details Panel of the Current Folder browser..

More About

Tips

- The `who` function displays the variable list unless you specify an output argument.
- When used within a nested function, the `who` function lists the variables in the workspaces of that function and all functions containing that function, grouped by workspace. This applies whether you call `who` from your function code or from the MATLAB debugger.

See Also

`assignin` | `clear` | `dir` | `evalin` | `exist` | `inmem` | `load` | `save` | `what` | `whos` | `workspace`

Introduced before R2006a

who

Class: matlab.io.MatFile
Package: matlab.io

Names of variables in MAT-file

Syntax

```
varlist = who(matObj)
varlist = who(matObj,variables)
```

Description

`varlist = who(matObj)` lists alphabetically all variables in the MAT-file associated with `matObj`. Optionally, returns the list in cell array `varlist`.

`varlist = who(matObj,variables)` lists the specified variables.

Input Arguments

`matObj`

Object created by the `matfile` function.

`variables`

Names of variables in the MAT-file corresponding to `matObj`. Use one of these forms:

`var1,...,varN`	Comma-separated list of variable name strings. Optionally, match patterns with the `'*'` wildcard, such as `who(matobj,'A*')`.
`'-regexp',expressions`	Regular expression strings that describe variable names.

Output Arguments

`varlist`

Cell array of strings that correspond to each variable name.

Examples

Display a list of variables in the example file `topography.mat`:

```
matObj = matfile('topography.mat');
who(matObj)
```

This code returns:

```
Your variables are:

topo        topolegend  topomap1    topomap2
```

See Also

`whos` | `matfile`

whos

List variables in workspace, with sizes and types

Syntax

```
whos
whos(variables)
whos(location)
whos(variables,location)

S = whos( ___ )
```

Description

`whos` displays in alphabetical order all variables in the currently active workspace, with information about their sizes and types.

`whos(variables)` displays only the specified variables.

`whos(location)` displays variables in the specified location.

`whos(variables,location)` displays the specified variables in the specified location. The `location` and `variables` inputs can appear in either order.

`S = whos( ___ )` stores information about the variables in structure array, `S`, using no input arguments, or any of the input arguments from the previous syntaxes.

Examples

Display Information About Workspace Variables

Display information about variables in the current workspace whose names start with the letter `a`.

```
whos('a*')
```

Display Variables Stored in a MAT-File

Display all variables stored in the sample MAT-file, `durer.mat`.

```
whos('-file','durer.mat')
```

```
  Name           Size                Bytes  Class     Attributes

  X            648x509            2638656  double
  caption        2x28                 112  char
  map          128x3                 3072  double
```

Show only variables in the MAT-file with names that end with `ion`.

```
whos('-file','durer.mat','-regexp','ion$')
```

```
  Name          Size             Bytes  Class    Attributes

  caption       2x28               112  char
```

Store Variable Information in a Structure Array

Store information about the variables in `durer.mat` in structure array `S`.

```
S = whos('-file','durer.mat');
```

Display the contents of `S`.

```
for k = 1:length(S)
   disp(['  ' S(k).name ...
         '  ' mat2str(S(k).size) ...
         '  ' S(k).class]);
end
```

```
  X  [648 509]  double
  caption  [2 28]  char
  map  [128 3]  double
```

Create Function to Display Variable Attribute Information

In the Editor, create a function that creates variables with various persistent, global, sparse, and complex attributes, and then displays information about them.

```
function show_attributes
persistent p;
global g;
p = 1;
```

```
g = 2;
s = sparse(eye(5));
c = [4+5i 9-3i 7+6i];
whos
```

When you call the function, `show_attributes` displays the attributes.

```
show_attributes
```

```
Name      Size            Bytes  Class     Attributes

c         1x3                48  double    complex
g         1x1                 8  double    global
p         1x1                 8  double    persistent
s         5x5               128  double    sparse
```

Call whos Within a Nested or Anonymous Function

Create a function, `whos_demo`, that contains a nested function, `get_date`. Call `whos` within the nested function.

```
function whos_demo
date_time = datestr(now);

C = strsplit(date_time,{'-',''});
get_date(C);

   function get_date(d)
      day = d{1};
      mon = d{2};
      year = d{3};
      whos
   end

end
```

When you run the `whos_demo` function, `whos` displays the variables of the nested `get_date` function, and all functions that contain it, grouped by function workspace. This grouping applies whether you call `whos` from your function code or from the MATLAB debugger.

```
whos_demo
```

```
  Name           Size            Bytes  Class           Attributes
```

```
---- get_date ------------------------------------------------
d               1x3               372  cell
day             1x2                 4  char
mon             1x3                 6  char
year            1x13               26  char

---- whos_demo -----------------------------------------------
C               1x3               372  cell
ans             0x0                 0  (unassigned)
date_time       1x20               40  char
```

When called within an anonymous function, variables in the anonymous function also display in a group, titled with the function signature

Input Arguments

`variables` — Variables to display

strings

Variables to display, specified as one or more strings in one of the following forms.

Form of `Variables` Input	Variable Names
`var1 ... varN`	List the named variables, specified as individual strings. Use the `'*'` wildcard to match patterns. For example, `whos('A*')` lists all variables in the workspace that start with `A`.
`'-regexp',`*`expr1 ... exprN`*	List only the variables that match the regular expressions, specified as strings. For example, `whos('-regexp','^Mon','^Tues')` lists only the variables in the workspace that begin with `Mon` or `Tues`.

`location` — Location of variables

`'global' | '-file',`*`filename`*

Location of variables, specified as one of the following strings.

Value of `location`	Location of Variables
`'global'`	Global workspace.
`'-file',filename`	MAT-file. The *`filename`* input can include the full, relative, or partial path. For example, `whos('-file','myFile.mat')` lists all variables in the MAT-file named `myFile.mat`. `whos('-file',filename)` does not return the sizes of any MATLAB objects in file *`filename`*.

Output Arguments

S — Information about variables

nested structure array

Information about variables, returned as a nested structure array that contains a scalar `struct` for each variable. Each scalar `struct` contains these fields.

Field	Description
`name`	Name of the variable.
`size`	Dimensions of the variable array.
`bytes`	Number of bytes allocated for the variable array.
`class`	Class of the variable. If the variable has no value, class is `'(unassigned)'`.
`global`	True if the variable is global; otherwise, false.
`sparse`	True if the variable is sparse; otherwise, false.
`complex`	True if the variable is complex; otherwise, false.
`nesting`	Structure with these fields: • `function` — Name of the nested or outer function that defines the variable. • `level` — Nesting level of that function.
`persistent`	True if the variable is persistent; otherwise, false.

More About

Tips

- You also can view the contents of MAT-files using the Details Panel of the Current Folder browser.

Algorithms

`whos` returns the number of bytes each variable occupies in the workspace, not in a MAT-file. Version 7 MAT-files and later are compressed, so the number of bytes in the file is typically fewer than the number of bytes required in the workspace.

- "What Is the MATLAB Workspace?"

See Also

`clear` | `exist` | `what` | `who`

Introduced before R2006a

whos

Class: matlab.io.MatFile
Package: matlab.io

Names, sizes, and types of variables in MAT-file

Syntax

```
details = whos(matObj)
details = whos(matObj,variables)
```

Description

`details = whos(matObj)` returns information about all variables in the MAT-file associated with `matObj`.

`details = whos(matObj,variables)` returns information about the specified variables.

Input Arguments

`matObj`

Object created by the `matfile` function.

`variables`

Names of variables in the MAT-file corresponding to `matObj`. Use one of these forms:

`var1,...,varN`	Comma-separated list of variable name strings. Optionally, match patterns with the `'*'` wildcard, such as `whos(matobj,'A*')`.
`'-regexp',expressions`	Regular expression strings that describe variable names.

Output Arguments

`details`

Structure array with these fields (identical to the structure returned by the `whos` function):

`name`	Variable name
`size`	Dimensions of the variable
`bytes`	Number of bytes allocated for the array when you load the entire variable
`class`	Class (data type) of the variable
`global`	Whether the variable is global (`true` or `false`)
`sparse`	Whether the variable is sparse
`complex`	Whether the variable is complex
`nesting`	Structure with these fields: • `function` — Name of the nested or outer function that defines the variable • `level` — Nesting level
`persistent`	Whether the variable is persistent

Examples

Display a list of variables in the example file `topography.mat`:

```
matObj = matfile('topography.mat');
whos(matObj)
```

This code returns:

```
  Name              Size              Bytes  Class     Attributes

  topo            180x360             518400  double
  topolegend        1x3                   24  double
  topomap1         64x3                 1536  double
  topomap2        128x3                 3072  double
```

Without loading any data, find the size and number of dimensions of the variable `topo` in `topography.mat`:

```
matObj = matfile('topography.mat');
info = whos(matObj,'topo');
sizeX = info.size
nDimsX = length(sizeX)
```

This code returns:

```
sizeX =
   180   360

nDimsX =
     2
```

See Also

`size` | `matfile`

width

Number of table variables

Syntax

```
W = width(T)
```

Description

`W = width(T)` returns the number of variables in table `T`.

`width(T)` is equivalent to `size(T,2)`.

Examples

Number of Variables in Table

Create a table, `T`.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
T =

                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38     71        176       124          93
    Johnson     43     69        163       109          77
    Williams    38     64        131       125          83
    Jones       40     67        133       117          75
    Brown       49     64        119       122          80
```

Find the number of variables in table `T`.

```
W = width(T)
```

```
W =

     4
```

`T` contains `4` variables; `width` does not count the row names.

The variable `BloodPressure` counts as one variable even though it contains two columns.

Input Arguments

`T` — Input table

table

Input table, specified as a table.

Variables in a table can have multiple columns, but `width(T)` only counts the number of variables.

See Also

`height` | `numel` | `size`

wilkinson

Wilkinson's eigenvalue test matrix

Syntax

```
W = wilkinson(n)
```

Description

`W = wilkinson(n)` returns one of J. H. Wilkinson's eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Examples

```
wilkinson(7)

ans =

    3    1    0    0    0    0    0
    1    2    1    0    0    0    0
    0    1    1    1    0    0    0
    0    0    1    0    1    0    0
    0    0    0    1    1    1    0
    0    0    0    0    1    2    1
    0    0    0    0    0    1    3
```

The most frequently used case is `wilkinson(21)`. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

See Also

`eig` | `gallery` | `pascal`

Introduced before R2006a

winopen

Open file in appropriate application (Windows)

Syntax

```
winopen(fileName)
```

Description

`winopen(fileName)` opens `fileName` in the associated Microsoft Windows application. The application is associated with the extension in `fileName` in the Windows operating system. `filename` is a string enclosed in single quotes. `winopen` uses a Windows shell command, and performs the same action as double-clicking the file in the Windows Explorer program. That is, `winopen` calls the application associated the file extension to open the file. Use an absolute or relative path for `fileName`.

Examples

Open the file `thesis.doc`, located in the current folder, in the Microsoft Word program:

```
winopen('thesis.doc')
```

Open `myresults.html` in the system Web browser:

```
winopen('D:/myfiles/myresults.html')
```

On Microsoft Windows platforms, open the current folder in the Windows Explorer tool:

```
winopen(cd)
```

To open a file on the MATLAB path, use `winopen` with `which`. For example, to open the `meshgrid` function in the Editor, use:

```
winopen(which('meshgrid'))
```

See Also

`dos` | `open` | `web`

Introduced before R2006a

winqueryreg

Item from Windows registry

Syntax

```
valnames = winqueryreg('name', 'rootkey', 'subkey')
value = winqueryreg('rootkey', 'subkey', 'valname')
value = winqueryreg('rootkey', 'subkey')
```

Description

`valnames = winqueryreg('name', 'rootkey', 'subkey')` returns all value names in `rootkey\subkey` of Microsoft Windows operating system registry to a cell array of strings. The first argument is the literal quoted string, `'name'`.

`value = winqueryreg('rootkey', 'subkey', 'valname')` returns the value for value name `valname` in `rootkey\subkey`.

If the value retrieved from the registry is a string, `winqueryreg` returns a string. If the value is a 32-bit integer, `winqueryreg` returns the value as an integer of the MATLAB software type `int32`.

`value = winqueryreg('rootkey', 'subkey')` returns a value in `rootkey\subkey` that has no value name property.

Note The literal **`name`** argument and the `rootkey` argument are case-sensitive. The `subkey` and `valname` arguments are not.

Examples

Example 1

Get the value of `CLSID` for the MATLAB sample Microsoft COM control `mwsampctrl.2`:

```
winqueryreg 'HKEY_CLASSES_ROOT' 'mwsamp.mwsampctrl.2\clsid'

ans =
   {5771A80A-2294-4CAC-A75B-157DCDDD3653}
```

Example 2

Get a list in variable `mousechar` for registry subkey `Mouse`, which is under subkey `Control Panel`, which is under root key `HKEY_CURRENT_USER`.

```
mousechar = winqueryreg('name', 'HKEY_CURRENT_USER', ...
   'control panel\mouse');
```

For each name in the `mousechar` list, get its value from the registry and then display the name and its value:

```
for k=1:length(mousechar)
   setting = winqueryreg('HKEY_CURRENT_USER', ...
      'control panel\mouse', mousechar{k});
   str = sprintf('%s = %s', mousechar{k}, num2str(setting));
   disp(str)
   end

ActiveWindowTracking = 0
DoubleClickHeight = 4
DoubleClickSpeed = 830
DoubleClickWidth = 4
MouseSpeed = 1
MouseThreshold1 = 6
MouseThreshold2 = 10
SnapToDefaultButton = 0
SwapMouseButtons = 0
```

More About

Tips

This function works only for the following registry value types:

- strings (REG_SZ)
- expanded strings (REG_EXPAND_SZ)
- 32-bit integer (REG_DWORD)

Introduced before R2006a

workspace

Open Workspace browser to manage workspace

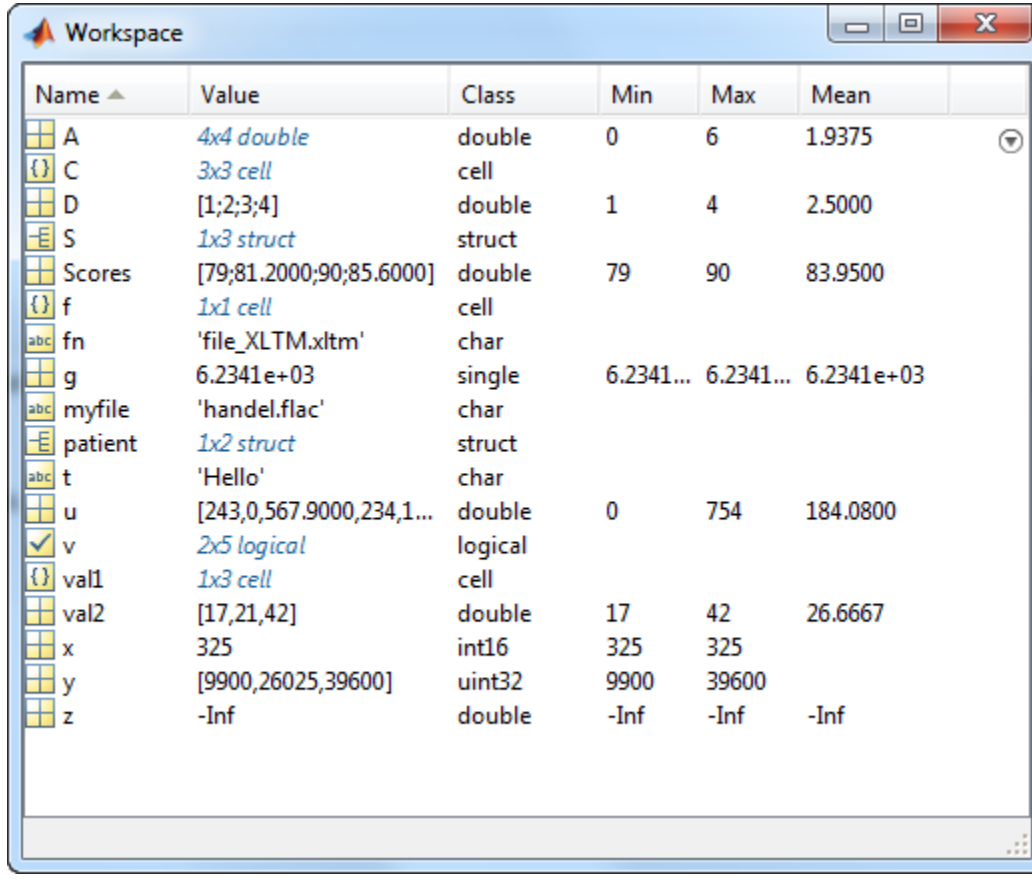
Syntax

```
workspace
```

Description

`workspace` displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the workspace in MATLAB. It provides a graphical representation of the `whos` display, and allows you to perform the equivalent of the `clear`, `load`, `open`, and `save` functions.

The Workspace browser also displays and automatically updates statistical calculations for each variable, which you can choose to show or hide.



You can edit a value directly in the Workspace browser for small numeric and character arrays. To see and edit a graphical representation of larger variables and for other classes, double-click the variable in the Workspace browser. The variable displays in the Variables editor, where you can view the full contents and make changes.

More About

- "What Is the MATLAB Workspace?"

See Also

`openvar` | `who`

Introduced before R2006a

write

Write data to remote host over TCP/IP interface

Syntax

```
write(t,data)
```

Description

`write(t,data)` sends the N-dimensional matrix of data from `tcpclient` object `t` connected to the remote host. The function waits until the specified values are written to the remote host.

Examples

Read and Write uint8 Data

Create a TCP/IP object called `t`, connecting to a TCP/IP echo server, with Port of `7`. This requires you to have an `echotcpip` server running on Port 7.

```
t = tcpclient('localhost', 7)
```

```
t =

   tcpclient with properties:

          Address: 'local host'
             Port: 7
          Timeout: 10
   BytesAvailable: 0
```

The `write` function synchronously writes data to the remote host connected to the `tcpclient` object. First specify the data, then write the data. The function waits until the specified number of values is written to the remote host.

Assign 10 bytes of `uint8` data to the variable `data`.

```
data = uint8(1:10)
```

```
data =

  1    2    3    4    5    6    7    8    9    10
```

Check the data.

```
whos data
```

```
Name      Size      Bytes      Class      Attributes

data      1x10         10      uint8
```

Write data to the echo server.

```
write(t, data)
```

Check that the data was written using the `BytesAvailable` property.

```
t.BytesAvailable
```

```
ans =

    10
```

Read the data from the server.

```
read(t)
```

```
ans =

  1    2    3    4    5    6    7    8    9    10
```

Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Input Arguments

`data` — Data to write to the remote host

1xN matrix of numeric data

Data to write to the remote host, specified as a 1xN matrix of numeric data.

Example: `write(t, data)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

writetable

Write table to file

Syntax

```
writetable(T)
writetable(T,filename)
writetable( ___ ,Name,Value)
```

Description

`writetable(T)` writes the table, `T`, to a comma delimited text file. The file name is the workspace variable name of the table, appended with the extension `.txt`. If `writetable` cannot construct the file name from the input table name, then it writes to the file `table.txt`.

Each column of each variable in `T` becomes a column in the output file. The variable names of `T` become column headings in the first line of the file.

`writetable(T,filename)` writes table `T` to a file with the name and extension specified by `filename`.

`writetable` determines the file format based on the specified extension. The extension must be one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, or `.xlsx` for Excel spreadsheet files

`writetable( ___ ,Name,Value)` writes the table to a file with additional options specified by one or more Name,Value pair arguments and can include any of the input arguments in previous syntaxes.

For example, you can specify whether to write the variable names as column headings in the output file.

Examples

Write Table to Text File

Create a table.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T =

    Var1      Var2      Var3    Var4
    ____    _________    ____    _____

    M       45    45     'NY'    true
    F       41    32     'CA'    false
    M       40    34     'MA'    false
```

Write the table, `T`, to a comma delimited text file.

```
writetable(T)
```

`writetable` outputs a text file named `T.txt`.

Display the contents of the file.

```
type 'T.txt'
```

```
Var1,Var2_1,Var2_2,Var3,Var4
M,45,45,NY,1
F,41,32,CA,0
M,40,34,MA,0
```

`writetable` appends a unique suffix to the variable name, `Var2`, above the two columns of corresponding data.

Write Table to Space Delimited Text File

Create a table.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
```

```
    {'NY';'CA';'MA'},[true;false;false])
```

```
T =

    Var1      Var2      Var3    Var4
    ____    _________    ____    _____

    M       45    45     'NY'    true
    F       41    32     'CA'    false
    M       40    34     'MA'    false
```

Write the table to a space delimited text file named `myData.txt`.

```
writetable(T,'myData.txt','Delimiter',' ')
```

Display the contents of the file.

```
type 'myData.txt'
```

```
Var1 Var2_1 Var2_2 Var3 Var4
M 45 45 NY 1
F 41 32 CA 0
M 40 34 MA 0
```

`writetable` appends a unique suffix to the variable name, `Var2`, above the two columns of corresponding data.

Write Table to Text File Including Row Names

Create a table.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T = table(Age,Height,Weight,BloodPressure,...
    'RowNames',LastName)
```

```
T =
```

```
              Age    Height    Weight    BloodPressure
              ___    ______    ______    _____________

    Smith     38     71        176       124      93
    Johnson   43     69        163       109      77
    Williams  38     64        131       125      83
    Jones     40     67        133       117      75
    Brown     49     64        119       122      80
```

Write the table, `T`, to a comma delimited text file, called `myPatientData.dat`.

```
writetable(T,'myPatientData.dat','WriteRowNames',true)
```

Display the contents of the file.

```
type 'myPatientData.dat'
```

```
Row,Age,Height,Weight,BloodPressure_1,BloodPressure_2
Smith,38,71,176,124,93
Johnson,43,69,163,109,77
Williams,38,64,131,125,83
Jones,40,67,133,117,75
Brown,49,64,119,122,80
```

The first column, which contains the row names, has the column heading, `Row`. This is the first dimension name for the table from the property `T.Properties.DimensionNames`.

Write Quoted Strings to CSV File

Create a table.

```
T = table(['M';'F';'M'],[45;41;36],...
    {'New York, NY';'San Diego, CA';'Boston, MA'},[true;false;false])
```

```
T =

    Var1    Var2         Var3          Var4
    ____    ____    _______________    _____

    M       45      'New York, NY'     true
    F       41      'San Diego, CA'    false
    M       36      'Boston, MA'       false
```

Write the table to a comma-separated text file named `myData.csv`. Enclose strings in double quotation marks using the `'QuoteStrings'` name-value pair argument, to ensure that the commas in the third column are not treated as delimiters.

```
writetable(T,'myData.csv','Delimiter',',','QuoteStrings',true)
```

View the contents of the file.

```
type 'myData.csv'
```

```
Var1,Var2,Var3,Var4
"M",45,"New York, NY",1
"F",41,"San Diego, CA",0
"M",36,"Boston, MA",0
```

Write Table to Specific Sheet and Range in Spreadsheet

Create a table.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T =

    Var1          Var2          Var3    Var4
    ____    ________________    ____    _____

    M       45          45      'NY'    true
    F       41          32      'CA'    false
    M       40          34      'MA'    false
```

Write the table to a spreadsheet named `myData.xls`. Include the data on the second sheet in the 5-by-5 region with corners at `B2` and `F6`.

```
writetable(T,'myData.xls','Sheet',2,'Range','B2:F6')
```

Excel fills the row of the spreadsheet from `B6` to `F6` with `#N/A` since the range specified is larger than the size of the input table `T`.

Input Arguments

T — Input table

table

Input table, specified as a table.

`filename` — File name
string

File name, specified as a string. To write to a specific folder, specify the full path name. Otherwise, `writetable` writes to a file in the current folder. If `filename` includes the file's extension, then `writetable` determines the file format from the extension. Otherwise, `writetable` creates a comma separated text file and appends the extension `.txt`. Alternatively, you can specify `filename` without the file's extension, and then include the `'FileType'` name-value pair arguments to indicate the type of file.

If `filename` does not exist, then `writetable` creates the file. If `filename` is the name of an existing text file, then `writetable` overwrites the file. If `filename` is the name of an existing spreadsheet file, then `writetable` writes a table to the specified location, but does not overwrite any values outside that range.

Example: `'myData.xls'`

Example: `'C:\test\myData.txt'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'WriteVariableNames',false` indicates that the variable names should not be included as the first row of the output file.

`'FileType'` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and the string `'text'` or `'spreadsheet'`.

The `'FileType'` name-value pair must be used with the `filename` input argument. You do not need to specify the `'FileType'` name-value pair argument if the file type can be determined from an extension in the filename input argument.

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, or `.xlsx` for Excel spreadsheet files

Example: `writetable(T,'mySpreadsheet','FileType','spreadsheet')`

`'WriteVariableNames'` — Indicator for writing variable names as column headings

`true` (default) | `false` | `1` | `0`

Indicator for writing variable names as column headings, specified as the comma-separated pair consisting of `'WriteVariableNames'` and either `true`, `false`, `1`, or `0`.

`true`	`writetable` includes variable names as the column headings of the output. This is the default behavior. If both the `'WriteVariablesNames'` and `'WriteRowNames'` logical indicators are `true`, then `writetable` uses the first dimension name from the property `T.Properties.DimensionNames` as the column heading for the first column of the output.
`false`	`writetable` does not include variable names in the output.

`'WriteRowNames'` — Indicator for writing row names in first column

`false` (default) | `true` | `0` | `1`

Indicator for writing row names in first column, specified as the comma-separated pair consisting of `'WriteRowNames'` and either `false`, `true`, `0`, or `1`.

`true`	`writetable` includes the row names from `T` as the first column of the output. If both the `'WriteVariablesNames'` and `'WriteRowNames'` logical indicators are `true`, then `writetable` uses the first dimension name from the property `T.Properties.DimensionNames` as the column heading for the first column of the output.
`false`	`writetable` does not include the row names from `T` in the output. This is the default behavior.

`'Delimiter'` — Field delimiter character

string

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and one of the following strings:

`','`	Comma. This is the default behavior.
`'comma'`	
`' '`	Space
`'space'`	
`'\t'`	Tab
`'tab'`	
`';'`	Semicolon
`'semi'`	
`'|'`	Vertical bar
`'bar'`	

You can use the `'Delimiter'` name-value pair only for delimited text files.

Example: `'Delimiter','space'`

`'QuoteStrings'` — Indicator for writing quoted strings

`false` (default) | `true` | `0` | `1`

Indicator for writing quoted strings, specified as the comma-separated pair consisting of `'QuoteStrings'` and either `false`, `true`, `0`, or `1`. If `'QuoteStrings'` is `true`, then `writetable` encloses strings in double quotation marks, and replaces any double-quote characters that appear as part of a string with two double-quote characters. For an example, see "Write Quoted Strings to CSV File" on page 1-8984.

You can use the `'QuoteStrings'` name-value pair only with delimited text files.

`'Sheet'` — Worksheet to write to

string containing worksheet name | positive integer indicating worksheet index

Worksheet to write to, specified as the comma-separated pair consisting of `'Sheet'` and a string containing the worksheet name or a positive integer indicating the worksheet index. The worksheet name string cannot contain a colon (`:`). To determine the names of sheets in a spreadsheet file, use `[status,sheets] = xlsfinfo(filename)`.

If the sheet does not exist, then `writetable` adds a new sheet at the end of the worksheet collection. If the sheet is an index larger than the number of worksheets, then

`writetable` appends empty sheets until the number of worksheets in the workbook equals the sheet index. In either case, `writetable` generates a warning indicating that it has added a new worksheet.

You can use the `'Sheet'` name-value pair only with spreadsheet files.

Example: `'Sheet',2`

`'Range'` — Rectangular portion of worksheet to write to

string

Rectangular portion of worksheet to write to, specified as the comma-separated pair consisting of `'Range'` and a string in one of the following forms.

Form of the Value of `Range`	Description
`'Corner1'`	*`Corner1`* specifies the first cell of the region to write. `writetable` writes table `T` beginning at this cell. **Example:** `'Range','D2'`
`'Corner1:Corner2'`	*`Corner1`* and *`Corner2`* are two opposing corners that define the region to write. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `'Range'` name-value pair argument is not case sensitive, and uses Excel A1 reference style (see Excel help). If the range you specify is larger than the size of the input table `T`, Excel software fills the remainder of the region with `#N/A`. If the range specified is smaller than the size of the input table `T`, then `writetable` writes only the subset that fits into the range. **Example:** `'Range','D2:H4'`

The `'Range'` name-value pair can only be used with Excel files.

Limitations

- On Linux systems, `writetable` writes only to text files and does not write to spreadsheet files.

More About

Tips

- In the cases below, `writetable` creates a file that does not represent T exactly. Then, if you use `readtable` to read that file and create a new table, the result might not have the same format or contents as the original table. Save `T` as a MAT-file if you need to import it again as a table with the same data and organization.
  - `writetable` outputs numeric variables using `long g` format, and categorical or character variables as unquoted strings in text files.
  - For variables that have more than one column, `writetable` appends a unique identifier to the variable name to use as the column headings.
  - For output variables that have more than two dimensions, `writetable` outputs these variables as two dimensional where the trailing dimensions are collapsed. For example, `writetable` outputs a 4-by-3-by-2 variable as if its size were 4-by-6.
  - For variables with a `cell` data type, `writetable` outputs the contents of each cell as a single row, in multiple fields. If the contents are other than numeric, logical, character, or categorical, then `writetable` outputs a single empty field.

Algorithms

Excel converts `Inf` values to `65535`. MATLAB converts `NaN` values to empty cells.

See Also

`readtable` | `table`

Introduced in R2013b

write

Class: Tiff

Write entire image

Syntax

```
write(tiffobj,imageData)
write(tiffobj,Y,Cb,Cr)
```

Description

`write(tiffobj,imageData)` writes `imageData` to TIFF file associated with the Tiff object, `tiffobj`. The `write` method breaks the data into strips or tiles, depending on the value of the `RowsPerStrip` tag, or the `TileLength` and `TileWidth` tags.

`write(tiffobj,Y,Cb,Cr)` writes the YCbCr component data to the TIFF file. Use this syntax only for images with a YCbCr photometric interpretation.

Examples

Write Image Data

Write tags and image data to a new TIFF file.

Read sample data into an array, `imdata`. Create a `Tiff` object associated with a new file, `myfile.tif`, and open the file for writing.

```
imdata = imread('example.tif');
t = Tiff('myfile.tif','w');
```

Set tag values.

```
setTag(t,'ImageLength',size(imdata,1))
setTag(t,'ImageWidth',size(imdata,2))
setTag(t,'Photometric',Tiff.Photometric.RGB)
```

```
setTag(t,'BitsPerSample',8)
setTag(t,'SamplesPerPixel',size(imdata,3))
setTag(t,'TileWidth',128)
setTag(t,'TileLength',128)
setTag(t,'Compression',Tiff.Compression.JPEG)
setTag(t,'PlanarConfiguration',Tiff.PlanarConfiguration.Chunky)
setTag(t,'Software','MATLAB')
```

Write the image data to the TIFF file.

```
write(t,imdata)
close(t)
```

- "Exporting Image Data and Metadata to TIFF Files"

See Also

`Tiff.writeDirectory`

writeDirectory

Class: Tiff

Create new IFD and make it current IFD

Syntax

```
writeDirectory(tiffobj)
```

Description

`writeDirectory(tiffobj)` create a new image file directory (IFD) and makes it the current IFD. `Tiff` object methods operate on the current IFD. If you are creating a TIFF file that only contains one image, you do not need to use this method. With single-image TIFF files, just close the `Tiff` object to write data to the file.

Examples

Open a TIFF file for modification and create a new IFD in the file. `writeDirectory` makes the newly created IFD the current IFD. Replace the name `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r+');
dnum = currentDirectory(t);
writeDirectory(t);
dnum = currentDirectory(t);
```

References

This method corresponds to the `TIFFWriteDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.write` | `Tiff.close`

writeEncodedStrip

Class: Tiff

Write data to specified strip

Syntax

```
writeEncodedStrip(tiffobj,stripNumber,imageData)
writeEncodedStrip(tiffobj,stripNumber,Y,Cb,Cr)
```

Description

`writeEncodedStrip(tiffobj,stripNumber,imageData)` writes the data in `imageData` to the strip specified by `stripNumber`. Strip identification numbers are one-based. If `imageData` has fewer bytes than fit into a strip, `writeEncodedStrip` silently pads the strip. If `imageData` has more bytes than fit into a strip, `writeEncodedStrip` issues a warning and truncates the data. To determine the size of a strip, view the value of the `RowsPerStrip` tag.

`writeEncodedStrip(tiffobj,stripNumber,Y,Cb,Cr)` writes the YCbCr component data to the specified tile. You must set the `YCbCrSubSampling` tag.

Examples

Write Image Data to Strips

Open a `Tiff` object for writing and set tag values.

```
t = Tiff('myfile.tif','w');
setTag(t,'ImageLength',32)
setTag(t,'ImageWidth',32)
setTag(t,'Photometric',Tiff.Photometric.MinIsBlack)
setTag(t,'BitsPerSample',8)
setTag(t,'SamplesPerPixel',1)
setTag(t,'RowsPerStrip',16)
setTag(t,'PlanarConfiguration',Tiff.PlanarConfiguration.Chunky)
```

Write data to the first and second strips.

```
writeEncodedStrip(t,1,ones(16,32,'uint8'))
writeEncodedStrip(t,2,2*ones(16,32,'uint8'))
close(t)
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

References

This method corresponds to the `TIFFWriteEncodedStrip` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.writeEncodedTile`

writeEncodedTile

Class: Tiff

Write data to specified tile

Syntax

```
writeEncodedTile(tiffobj,tileNumber,imageData)
writeEncodedTile(tiffobj,tileNumber,Y,Cb,Cr)
```

Description

`writeEncodedTile(tiffobj,tileNumber,imageData)` writes the data in `imageData` to the tile specified by `tileNumber`. Tile identification numbers are one-based. If `imageData` has fewer bytes than fit into a tile, `writeEncodedTile` silently pads the tile. If `imageData` has more bytes than fit into a tile, `writeEncodedTile` issues a warning and truncates the data. To determine the size of a tile, view the value of the `tileLength` and `tileWidth` tags.

`writeEncodedTile(tiffobj,tileNumber,Y,Cb,Cr)` writes the YCbCr component data to the specified tile. You must set the `YCbCrSubSampling` tags.

Examples

Write Image Data to Tiles

Open a `Tiff` object for writing and set tag values.

```
t = Tiff('myfile.tif','w');
setTag(t,'ImageLength',32)
setTag(t,'ImageWidth',32)
setTag(t,'Photometric',Tiff.Photometric.MinIsBlack)
setTag(t,'BitsPerSample',8)
setTag(t,'SamplesPerPixel',1)
setTag(t,'TileWidth',16)
setTag(t,'TileLength',16)
```

```
setTag(t,'PlanarConfiguration',Tiff.PlanarConfiguration.Chunky)
```

Write data to four tiles.

```
writeEncodedTile(t,1,ones(16,16,'uint8'))
writeEncodedTile(t,2,2*ones(16,16,'uint8'))
writeEncodedTile(t,3,3*ones(16,16,'uint8'))
writeEncodedTile(t,4,4*ones(16,16,'uint8'))
close(t)
```

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

References

This method corresponds to the `TIFFWriteEncodedTile` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at LibTIFF - TIFF Library and Utilities.

See Also

`Tiff.writeEncodedStrip`

writeVideo

Class: VideoWriter

Write video data to file

Syntax

```
writeVideo(writerObj,img)
writeVideo(writerObj,images)
writeVideo(writerObj,frame)
writeVideo(writerObj,mov)
```

Description

`writeVideo(writerObj,img)` writes data from an image to a video file.

`writeVideo(writerObj,images)` writes a sequence of color images to a video file.

`writeVideo(writerObj,frame)` writes a frame to the video file associated with writerObj.

`writeVideo(writerObj,mov)` writes a MATLAB movie to a video file. mov is an array of frame structures.

You must call `open(writerObj)` before calling `writeVideo`.

Input Arguments

`writerObj`

`VideoWriter` object created by the `VideoWriter` function.

`img`

When creating AVI or MPEG-4 files, `img` is an array of `single`, `double`, or `uint8` values representing a grayscale or RGB color image, which `writeVideo` writes as an

RGB video frame. Data of type `single` or `double` must be in the range `[0,1]`, except when writing Indexed AVI files.

When creating Motion JPEG 2000 files, `img` is an array of `uint8`, `int8`, `uint16`, or `int16` values representing a monochrome or RGB color image.

For grayscale, monochrome, or indexed data, img must be two-dimensional: height-by-width. For color data that is not indexed, img is three-dimensional: height-by-width-by-3. The height and width must be consistent for all frames within a file. For more information, see "Image Types".

`images`

Four-dimensional array of grayscale (height-by-width-by-1-by-frames) or RGB (height-by-width-by-3-by-frames) images.

`frame`

Structure typically returned by the `getframe` function that contains two fields: `cdata` and `colormap`. If the `colormap` is not empty, `writeVideo` expects a two-dimensional (height-by-width) array `cdata`. The height and width must be consistent for all frames within a file. `colormap` can contain a maximum of 256 entries.

The `profile` of `writerObj` and the size of `cdata` determine how the `writeVideo` method uses `frame`.

`profile` of `VideoWriter` object	Size of `cdata`	Behavior of `writeVideo`
`'Indexed AVI'` or `'Grayscale AVI'`	2-dimensional (height-by-width)	Use frame as provided. For `'Grayscale AVI'`, `colormap` should be empty.
	3-dimensional (height-by-width-by-3)	Error
All other profiles	2-dimensional (height-by-width)	Construct RGB image frames using the `colormap` field
	3-dimensional (height-by-width-by-3)	`Colormap` field ignored. Construct RGB image frames using the `cdata` field

mov

1-by-F array of frame structures, where F is the number of images. Each frame structure contains fields `cdata` and `colormap`.

Examples

AVI File from Animation

Write a sequence of frames to a compressed AVI file, `peaks.avi`.

Prepare the new file.

```
writerObj = VideoWriter('peaks.avi');
open(writerObj);
```

Generate initial data and set axes and figure properties.

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
set(gcf,'Renderer','zbuffer');
```

Setting the `Renderer` property to `zbuffer` or `Painters` works around limitations of `getframe` with the OpenGL renderer on some Windows systems.

Create a set of frames and write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe;
   writeVideo(writerObj,frame);
end

close(writerObj);
```

MPG to AVI Conversion

Convert an example file, `xylophone.mp4`, to an uncompressed AVI file:

Create objects to read and write the video, and open the AVI file for writing.

```
readerObj = VideoReader('xylophone.mp4');
```

```
writerObj = VideoWriter('transcoded_xylophone.avi', ...
                        'Uncompressed AVI');

open(writerObj);
```

Read and write each frame.

```
for k = 1:readerObj.NumberOfFrames
   img = read(readerObj,k);
   writeVideo(writerObj,img);
end

close(writerObj);
```

See Also

open | close | getframe | VideoReader | VideoWriter

xlabel

Label x-axis

Syntax

```
xlabel(str)
xlabel(str,Name,Value)

xlabel(ax, ___ )

t = xlabel( ___ )
```

Description

`xlabel(str)` labels the *x*-axis of the current axes with the text specified by `str`. Reissuing the `xlabel` command replaces the old label with the new label.

`xlabel(str,Name,Value)` specifies text properties for the label using one or more `Name,Value` pair arguments. For example, `'Color','blue'` creates a blue label.

`xlabel(ax, ___ )` adds the label to the axes specified by `ax`, instead of the current axes. The option `ax` can precede any of the input argument combinations in the previous syntaxes.

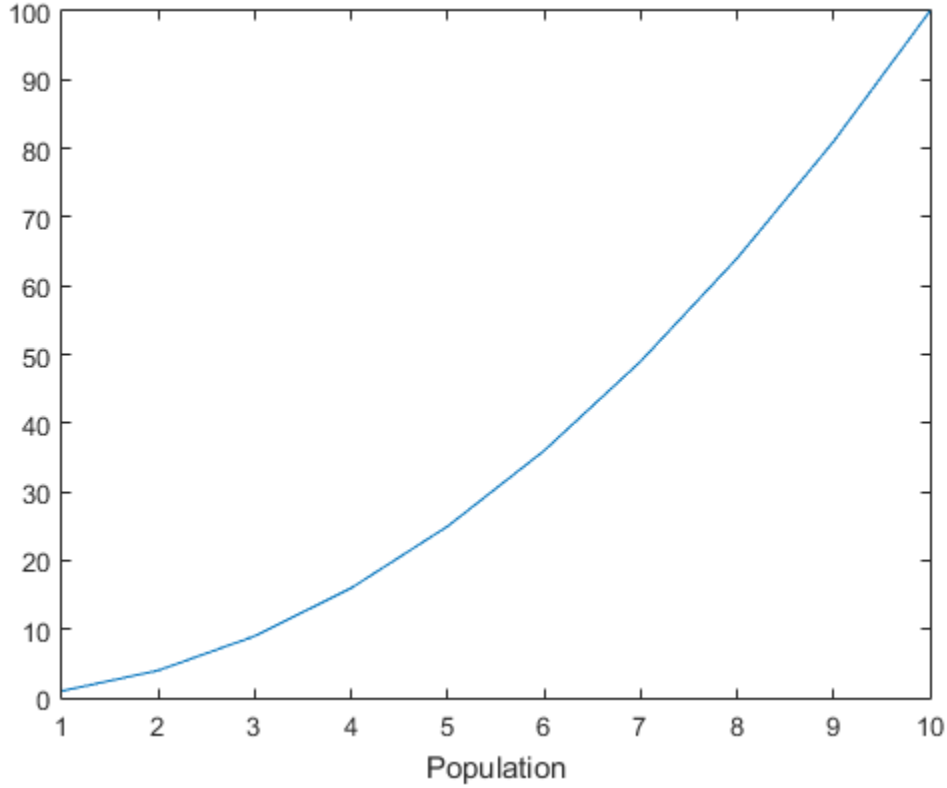
`t = xlabel( ___ )` returns the text object used as the *x*-axis label. Use `t` to make future modifications to the label after it is created.

Examples

Label x-Axis

Display `Population` beneath the *x*-axis.

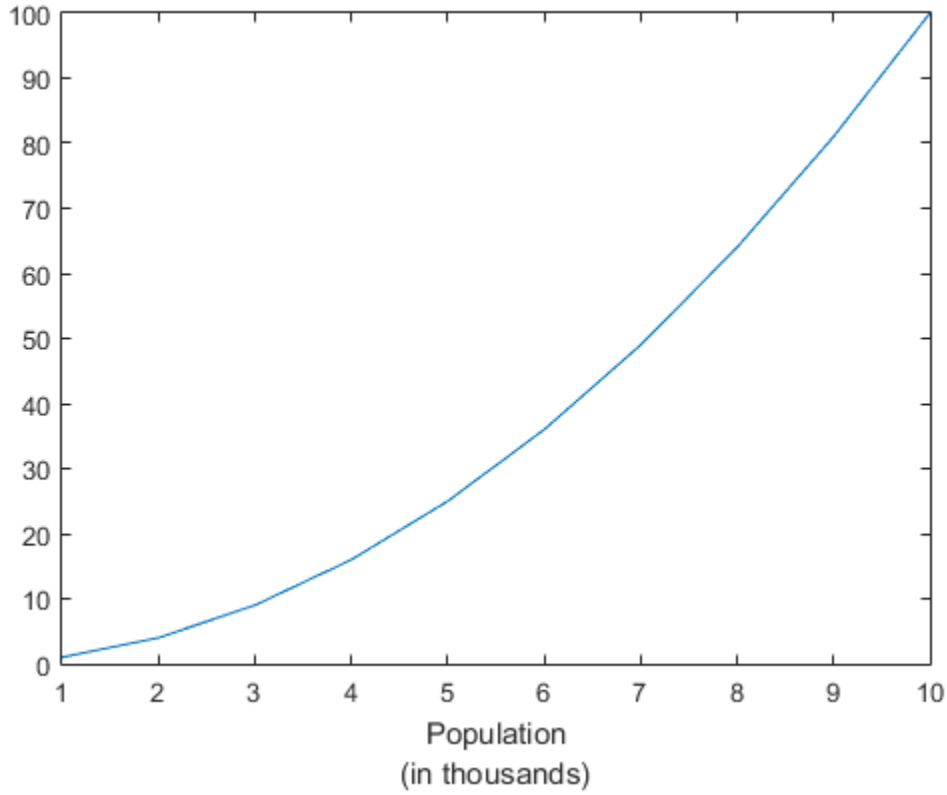
```
plot((1:10).^2)
xlabel('Population')
```



Create Multiline x-Axis Label

Create a multiline label using a cell array of strings.

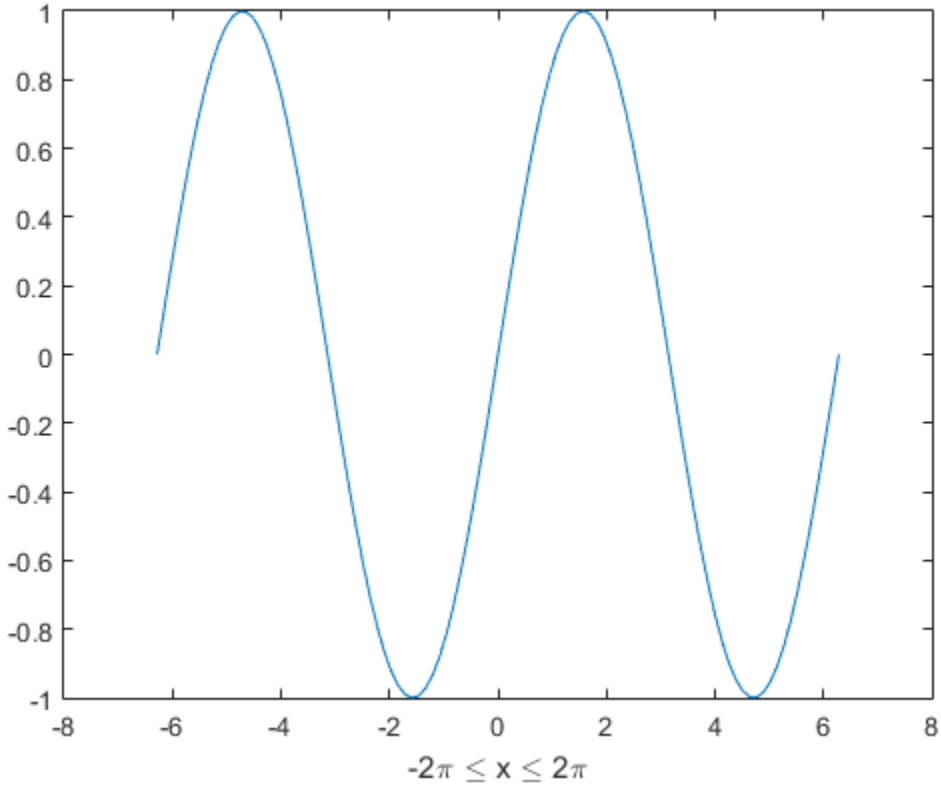
```
plot((1:10).^2)
xlabel({'Population','(in thousands)'})
```



Include Greek Letters in x-Axis Label

Include Greek letters and other special characters in the label using TeX markup.

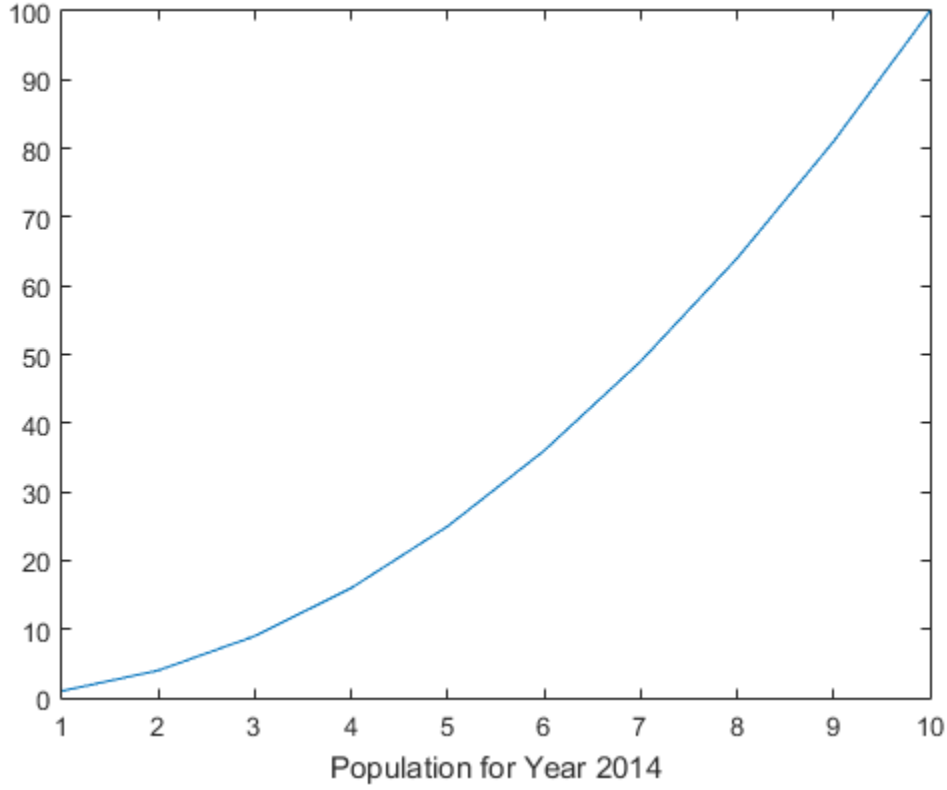
```
x = linspace(-2*pi,2*pi);
y = sin(x);
plot(x,y)
xlabel('-2\pi \leq x \leq 2\pi')
```



Include Variable Value in x-Axis Label

Add a label with text and a variable value. Use the `num2str` function to convert the variable value to a string.

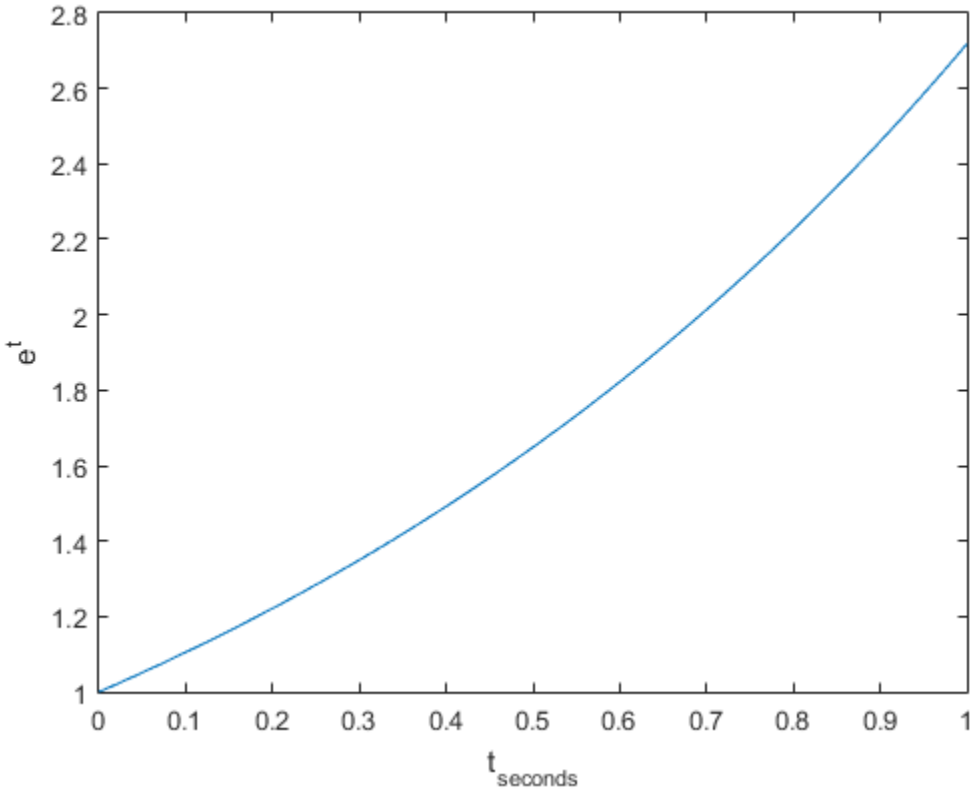
```
plot((1:10).^2)
year = 2014;
xlabel(['Population for Year ',num2str(year)])
```



Include Superscript and Subscript in Axis Labels

Use the `'^'` and `'_'` characters to include superscripts and subscripts in the axis labels. Use curly braces `{}` to modify more than one character.

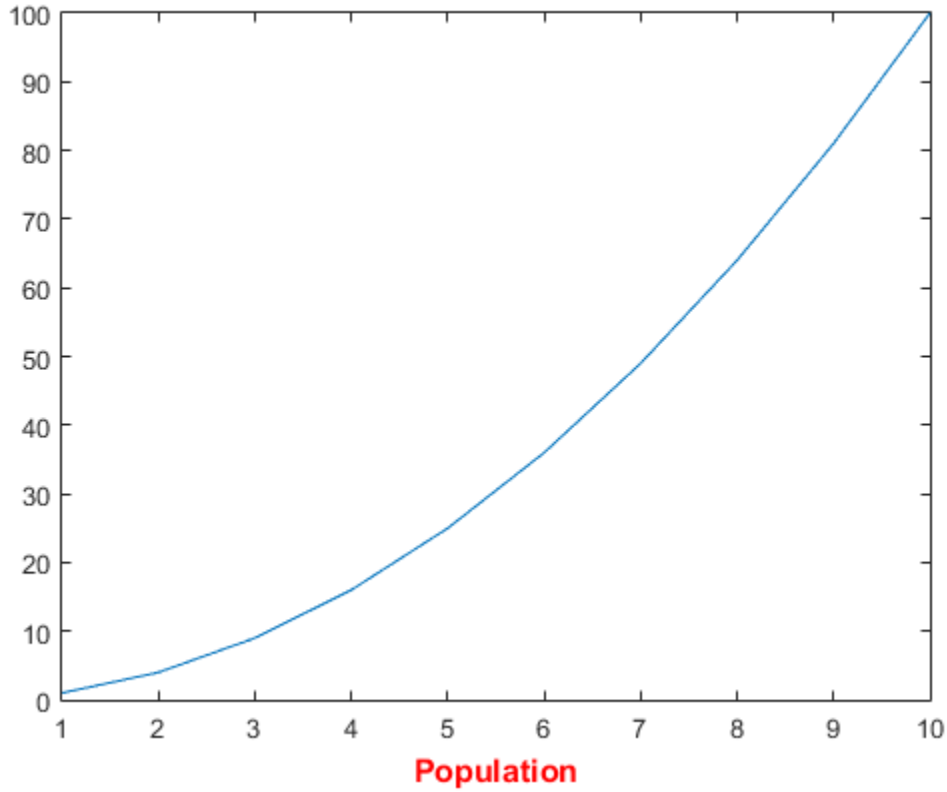
```
t = linspace(0,1);
y = exp(t);
plot(t,y)
xlabel('t_{seconds}')
ylabel('e^t')
```



Change x-Axis Label Font Size and Color

Use `Name,Value` pair arguments to set the font size, font weight, and text color properties of the *x*-axis label.

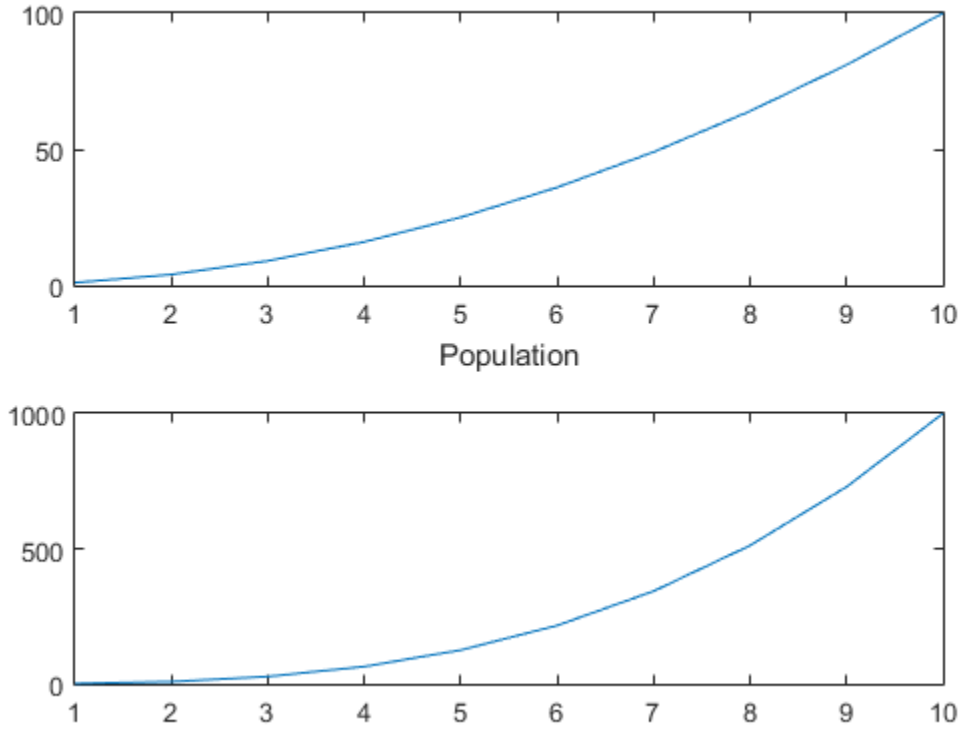
```
plot((1:10).^2)
xlabel('Population','FontSize',12,'FontWeight','bold','Color','r')
```



Label x-Axis of Specific Subplot

Create a figure with two subplots. Label the *x*-axis of the top subplot.

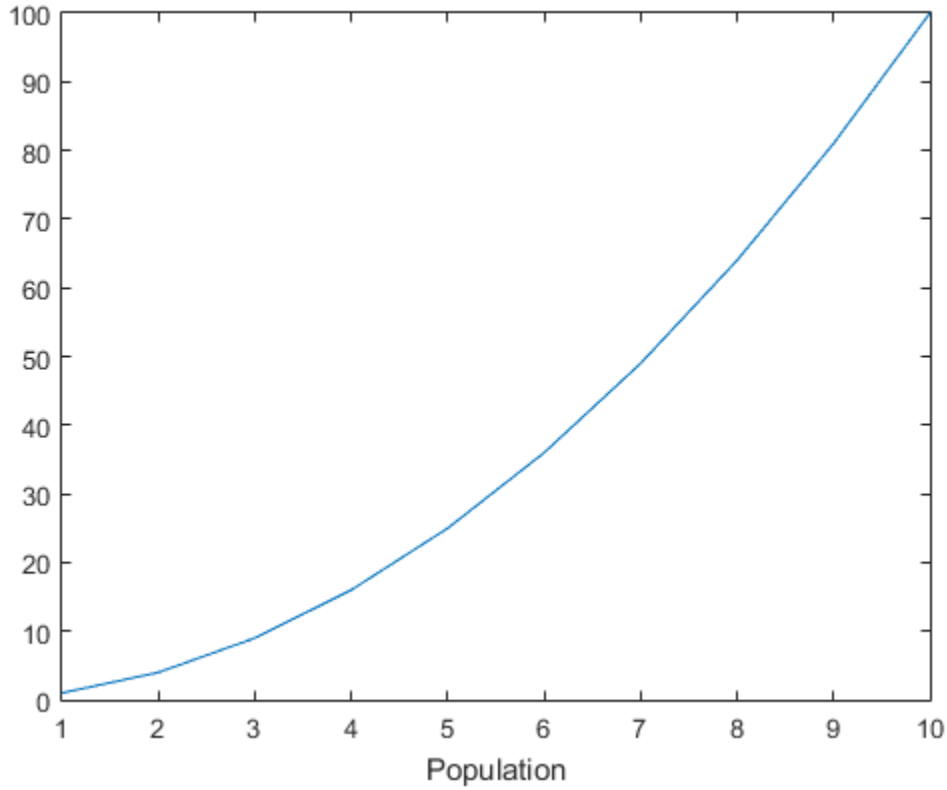
```
ax1 = subplot(2,1,1);
plot((1:10).^2)
xlabel(ax1,'Population')

ax2 = subplot(2,1,2);
plot((1:10).^3)
```



Modify x-Axis Label After Creation

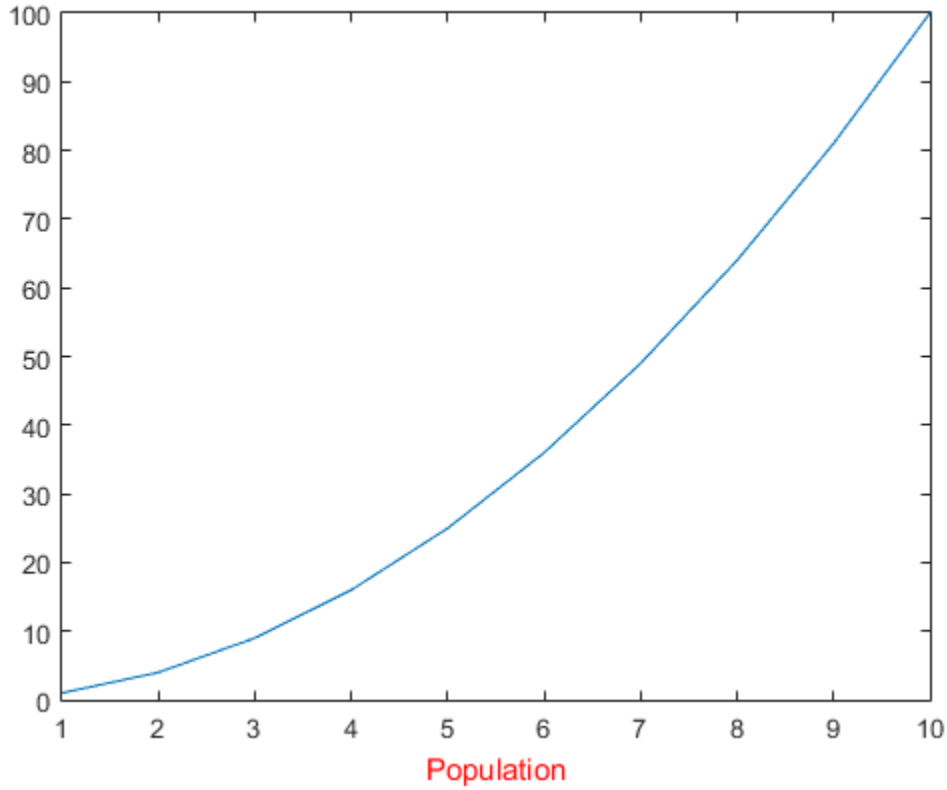
Label the *x*-axis and return the text object used as the label.

```
plot((1:10).^2)
t = xlabel('Population');
```



Use `t` to set text properties of the label after it has been created. For example, set the color of the label to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
t.Color = 'red';
```



- “Add Text to Graph Interactively”

Input Arguments

`str` — Axis label

character array | cell array | numeric value

Axis label, specified as a character array, cell array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: 123

To include numeric variables with text in a label, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the Interpreter property.

To create multiline labels:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create a string with a new line character, such as `sprintf('first line \n second line')`.

Numeric labels are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note: The words `default`, `factory`, and `remove` are reserved words that will not appear in a label when quoted as a normal string. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

---

ax — Axes object

axes object

Axes object. If you do not specify an axes, then the `xlabel` function adds the label to the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',12` specifies red, 12-point font.

The text properties listed here are only a subset. For a complete list, see Text Properties.

`'FontSize'` — Font size

`11` (default) | scalar value greater than `0`

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Setting the font size properties for the associated axes also affects the label font size. The label font size updates to equal the axes font size times the label scale factor. The `FontSize` property of the axes contains the axes font size. The `LabelFontSizeMultiplier` property of the axes contains the label scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the *x*-axis label font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'FontWeight'` — Thickness of text characters

`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

`'FontName'` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`'Color'` — Text color

`[0.15 0.15 0.15]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. If you set the color to `'none'`, then the text is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`'Interpreter'` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`

Modifier	Description	Example of String
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsui`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ε	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrighta`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	Θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

Output Arguments

`t` — Text object

text object

Text object used as the *x*-axis label. Use `t` to access and modify properties of the label after it has been created.

See Also

Functions

`num2str` | `text` | `title` | `ylabel` | `zlabel`

Properties

Text Properties

Introduced before R2006a

matlab.wsdl.createWSDLClient

Create interface to SOAP-based Web service

Syntax

```
matlab.wsdl.createWSDLClient(wsdlURL)
matlab.wsdl.createWSDLClient(wsdlURL,folder)
matlab.wsdl.createWSDLClient(wsdlURL,folder,'silent')

classname = matlab.wsdl.createWSDLClient( ___ )
```

Description

`matlab.wsdl.createWSDLClient(wsdlURL)` creates an interface to a service based on a Web Services Description Language (WSDL) document specified by wsdlURL. This function creates a MATLAB class file for each Simple Object Access Protocol (SOAP) binding in the WSDL and, if required, additional support files in the current folder. You can package and distribute these files to other users.

You must install the WSDL tools, Java JDK™ and Apache™ CXF software, then set the tool paths using the `matlab.wsdl.setWSDLToolPath` function.

You must have write permission for the current folder.

`matlab.wsdl.createWSDLClient(wsdlURL,folder)` creates the interface files in folder, which must be on the MATLAB path.

`matlab.wsdl.createWSDLClient(wsdlURL,folder,'silent')` suppresses display of generated files and folders.

`classname = matlab.wsdl.createWSDLClient( ___ )` returns a handle to the created class. The function returns a cell array of handles if multiple classes were created. You can use this syntax with any of the input arguments of the previous syntaxes.

To get information on using the class, call the MATLAB `help` function on the class name.

Examples

Get Map Name from USGSImageryOnly_MapServer Web Service

You must install the WSDL tools and set the tool paths to run this example. `jdk` and `cxf` are the paths to these tools on your system.

```
matlab.wsdl.setWSDLToolPath('JDK',jdk,'CXF',cxf)
```

Create the class files.

```
url = ...
'http://basemap.nationalmap.gov/arcgis/services/USGSImageryOnly/MapServer?wsdl';
matlab.wsdl.createWSDLClient(url);
```

```
Created USGSImageryOnly_MapServer.
  .\USGSImageryOnly_MapServer.m
  .\+wsdl

In order to use USGSImageryOnly_MapServer, you must run javaaddpath('.\+wsdl\mapserver
```

Add the jar files to the Java path.

```
javaaddpath('.\+wsdl\mapserver.jar')
```

Create the service.

```
wsdl = USGSImageryOnly_MapServer;
```

Read help for the service and its functions.

```
help USGSImageryOnly_MapServer
```

```
USGSImageryOnly_MapServer   A client to connect to the USGSImageryOnly_MapServer servic
     SERVICE = USGSImageryOnly_MapServer  connects to http://basemap.nationalmap.gov/ar

     To communicate with the service, call a function on the SERVICE:

        [...] = FUNCTION(SERVICE,arg,...)

     See doc USGSImageryOnly_MapServer for a list of functions.
```

Call one of the methods, for example `GetDefaultMapName` that returns the map name.

```
GetDefaultMapName(wsdl)
```

```
ans =

Layers
```

- “Set Up WSDL Tools”

Input Arguments

`wsdlURL` — WSDL URL or file path

string

WSDL URL or file path, specified as a string, that defines service methods, arguments, and transactions.

`wsdlURL` can be an `http` or `https` URL or a local path. `wsdlURL` cannot be a `file://` URL. On Microsoft Windows, UNC paths are not supported.

Example: `'http://www.mywebservice.com/servicename?WSDL'`

`folder` — Folder for generated files

string

Folder for generated files, specified as a string. If omitted or empty (`''`), `matlab.wsdl.createWSDLClient` uses the current folder. You must have write permission for the folder. The function overwrites existing files with the same names as the generated files.

Example: `'c:\work'`

Limitations

RPC-encoded WSDL documents are not supported. For these documents, use `createClassFromWsdl`.

The following WSDL documents are not supported:

- Some documents with messages containing multiple parts.
- Some documents with schemas containing anonymous complex types.
- Documents that the Apache CXF program cannot compile into complete code.

More About

Tips

- If you create WSDL files in multiple locations on your computer, avoid confusion by deleting the class files from duplicate locations, and then call `clear java`.

See Also

`clear` | `createClassFromWsdl` | `matlab.wsdl.setWSDLToolPath` | `webread`

Introduced in R2014b

matlab.wsdl.setWSDLToolPath

Location of WSDL tools

Syntax

```
matlab.wsdl.setWSDLToolPath(Name,Value)

paths = matlab.wsdl.setWSDLToolPath
```

Description

`matlab.wsdl.setWSDLToolPath(Name,Value)` sets the paths to Java JDK and Apache CXF software. You must download these tools to use the `matlab.wsdl.createWSDLClient` interface.

Values for the `Name` arguments, `'JDK'` and `'CXF'`, are saved across sessions in your user preferences, so you only need to specify them once. You must specify both values before calling `matlab.wsdl.createWSDLClient`.

`paths = matlab.wsdl.setWSDLToolPath` displays paths to the JDK and CXF software.

Examples

Set Path to Tools

Set `jdk` and `cxf` to valid paths on your system. For example,

```
jdk = 'E:/Program Files/win64/jdk';
cxf = 'c:\apache-cxf-2.7.10'
matlab.wsdl.setWSDLToolPath('JDK',jdk,'CXF',cxf)
```

Check if Tool Paths are Set

```
p = matlab.wsdl.setWSDLToolPath;
if (isempty(p.JDK) || isempty(p.CXF))
    disp('Install the Java Development Kit (JDK) and Apache CXF programs.')
    disp('See the Set Up WSDL Tools example link at the end of this page.')
```

```
else
    disp('Paths set to:')
    matlab.wsdl.setWSDLToolPath
end
```

- "Set Up WSDL Tools"

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'CXF','C:\apache-cxf-2.7.10'`

`'JDK'` — Path to Java JDK software

string

Path to Java Development Kit (JDK) software, specified as a string. Download the software from `http://www.oracle.com/technetwork/java/javase/downloads` and choose the latest release of 7.

Example: `'E:/Program Files/win64/jdk'`

`'CXF'` — Path to Apache CXF software

string

Path to Apache CXF software, specified as a string. CXF is an open-source services framework. Download the software from `http://cxf.apache.org/download` and choose the latest release of 2.7.

Example: `'C:\apache-cxf-2.7.10'`

Output Arguments

`paths` — Tool paths

structure

Tool paths, returned as a structure with the fields:

JDK — Path to Java JDK software
string

Path to Java JDK software, specified as a string. If the value of `JDK` is empty, you cannot call `matlab.wsdl.createWSDLClient`.

CXF — Path to Apache CXF software
string

Path to Apache CXF software, specified as a string. If the value of `CXF` is empty, you cannot call `matlab.wsdl.createWSDLClient`.

More About

- http://www.oracle.com/technetwork/java/javase/downloads
- http://cxf.apache.org/download

See Also

`matlab.wsdl.createWSDLClient`

Introduced in R2014b

year

Year number

Syntax

```
y = year(t)
y = year(t,yearType)
```

Description

`y = year(t)` returns the ISO year numbers for the datetime values in `t`. ISO year numbers include a year zero and represent years BCE using negative values. The `y` output is a `double` array the same size as `t`.

`y = year(t,yearType)` returns the type of year number specified by `yearType`.

The `year` function returns the year numbers of datetime values. To assign ISO year values to a datetime array, `t`, use `t.Year` and modify the `Year` property.

Examples

Extract Year Number from Dates

Extract the year numbers from an array of dates.

```
t = datetime(2010,05,31):calyears(1):datetime(2015,02,28)
```

```
t =

   31-May-2010    31-May-2011    31-May-2012    31-May-2013    31-May-2014
```

```
y = year(t)
```

```
y =
```

2010 2011 2012 2013 2014

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

`yearType` — Type of year values

`'iso'` (default) | `'gregorian'`

Type of year values, specified as either `'iso'` or `'gregorian'`.

- If `yearType` is `'iso'`, then `year` returns the ISO year number, which includes a year zero and represents years BCE using negative values.
- If `yearType` is `'gregorian'`, then `year` returns the Gregorian year number, which is an unsigned integer. For example, the Gregorian year number for 5 CE and 5 BCE is 5 in both cases. Gregorian year numbers do not have a year zero.

See Also

datetime Properties | `day` | `month` | `quarter` | `week` | `ymd`

Introduced in R2014b

years

Duration in years

Syntax

```
Y = years(X)
```

Description

`Y = years(X)` returns an array of years equivalent to the values in `X`.

- If `X` is a numeric array, then `Y` is a `duration` array in units of fixed-length years. A fixed-length year is equal to 365.2425 days.
- If `X` is a `duration` array, then `Y` is a `double` array with each element equal to the number of fixed-length years in the corresponding element of X.

The `years` function converts between `duration` and `double` values. To display a duration in units of years, set its `Format` property to `'y'`.

Examples

Create Duration Array of Fixed-Length Years

```
X = [1 3 5; 10 12 15]
```

```
X =

     1     3     5
    10    12    15
```

```
Y = years(X)
```

```
Y =
```

```
   1 yr    3 yrs    5 yrs
  10 yrs   12 yrs   15 yrs
```

Convert Durations to Numeric Array of Years

Find the difference between two arrays of dates. The output is a `duration` array.

```
t1 = datetime(2007:2010,10,1);
t2 = datetime(2014,05,1);
dt = t2 - t1
```

```
dt =

   57696:00:00   48912:00:00   40152:00:00   31392:00:00
```

Convert each duration in `dt` to a number of fixed-length years.

```
Y = years(dt)
```

```
Y =

    6.5819    5.5799    4.5805    3.5812
```

Input Arguments

X — Input array

numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

More About

Tips

- `years` creates fixed-length years. To create an array of calendar years that account for leap days when used in calendar calculations, use the `calyears` function.

See Also

`calyears`

Introduced in R2014b

ylabel

Label y-axis

Syntax

```
ylabel(str)
ylabel(str,Name,Value)

ylabel(ax, ___ )

h = ylabel( ___ )
```

Description

`ylabel(str)` labels the *y*-axis of the current axes with the string, `str`. Reissuing the `ylabel` command causes the new label to replace the old label. Labels appear beside the axis in a two-dimensional view and to the side or in front of the axis in a three-dimensional view.

`ylabel(str,Name,Value)` additionally specifies the text object properties using one or more `Name,Value` pair arguments.

`ylabel(ax, ___ )` adds the label to the axes specified by `ax`. This syntax allows you to specify the axes to which to add a label. `ax` can precede any of the input argument combinations in the previous syntaxes.

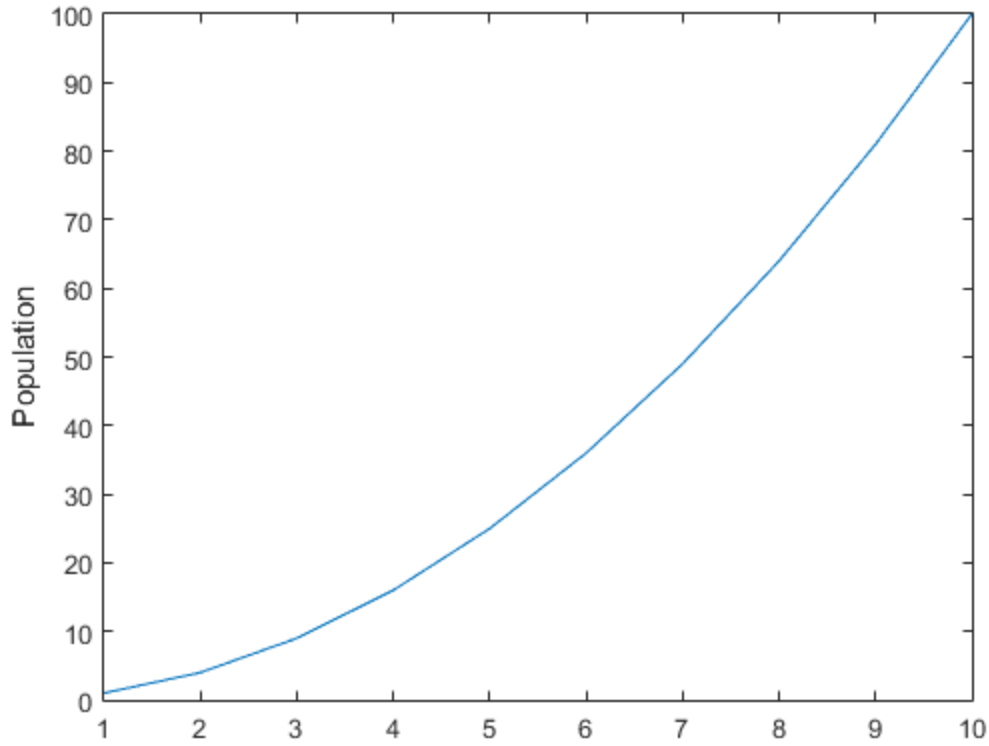
`h = ylabel( ___ )` returns the handle to the text object used as the *y*-axis label. The handle is useful when making future modifications to the label.

Examples

Label y-Axis with String

```
figure
```

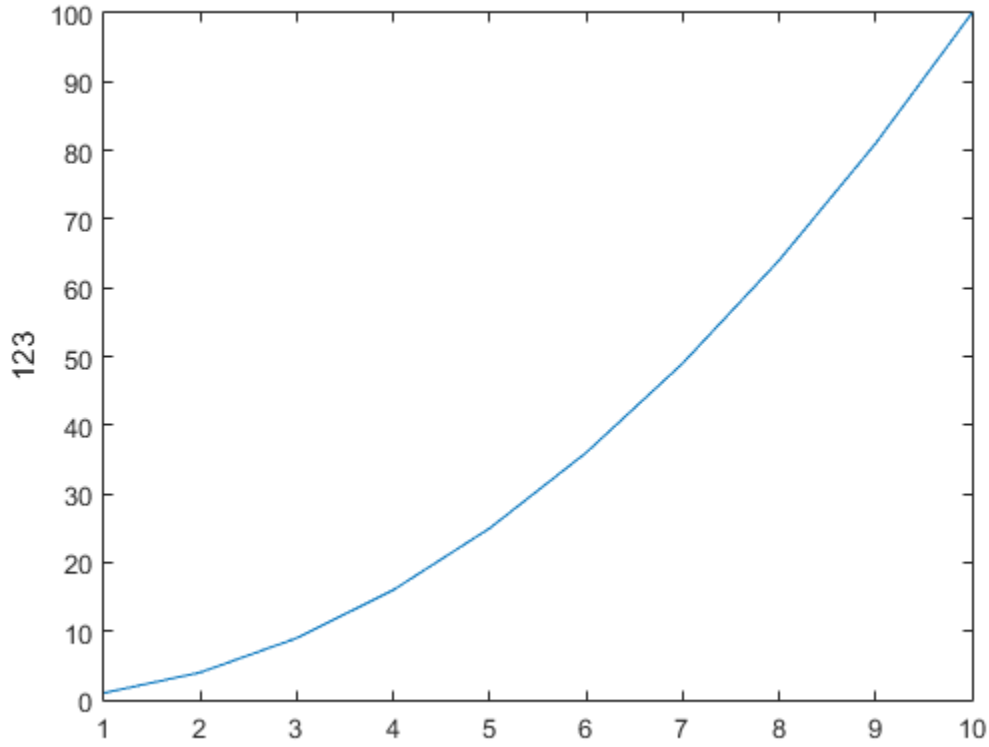
```
plot((1:10).^2)
ylabel('Population')
```



MATLAB® displays `Population` beside the *y*-axis.

Label y-Axis with Numeric Input

```
figure
plot((1:10).^2)
ylabel(123)
```

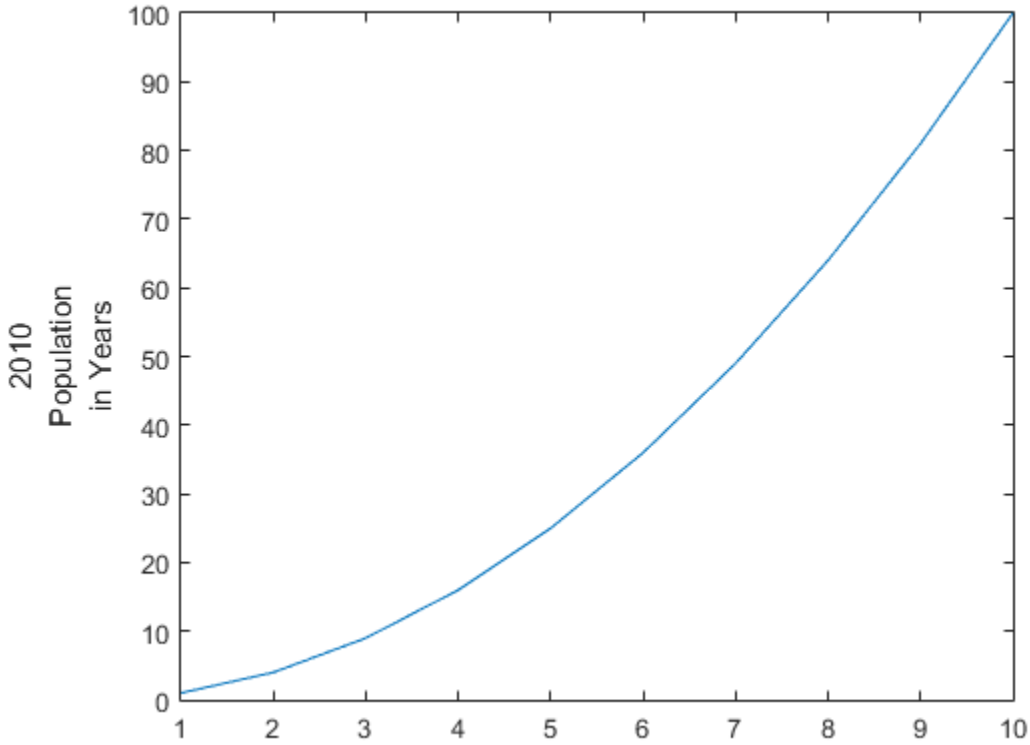


MATLAB® displays `123` beside the *y*-axis.

Create Multiline Label

Create a multiline label using a multiline cell array.

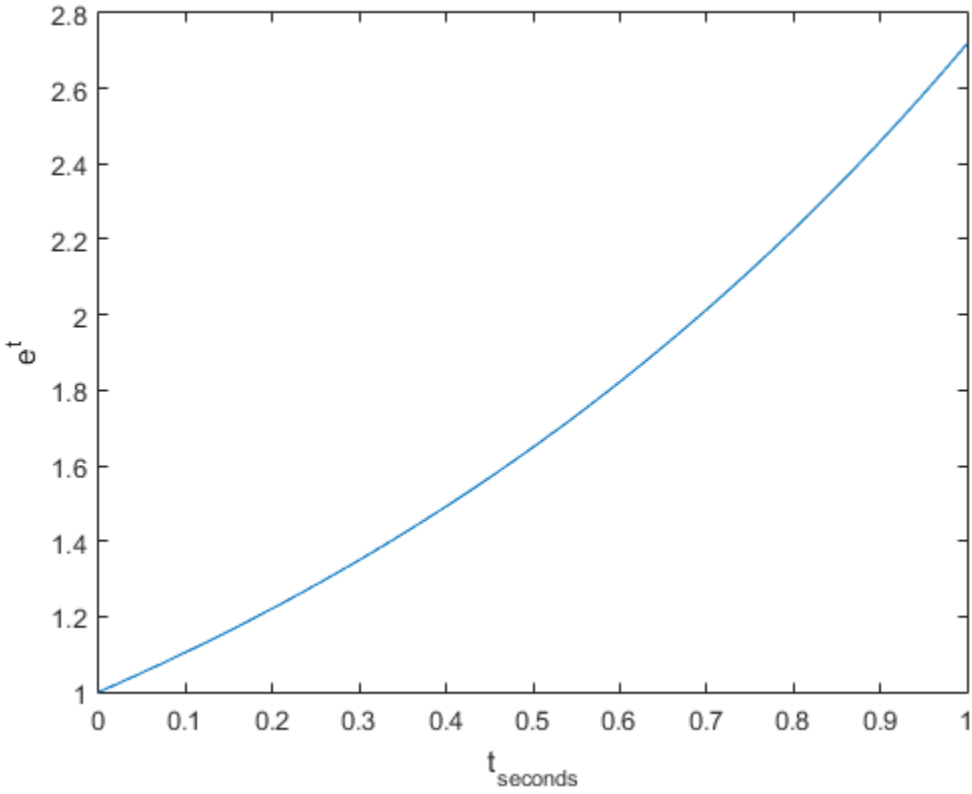
```
figure
plot((1:10).^2)
ylabel({2010;'Population';'in Years'})
```



Include Superscript and Subscript in Axis Labels

Use the `'^'` and `'_'` characters to include superscripts and subscripts in the axis labels. Use curly braces `{}` to modify more than one character.

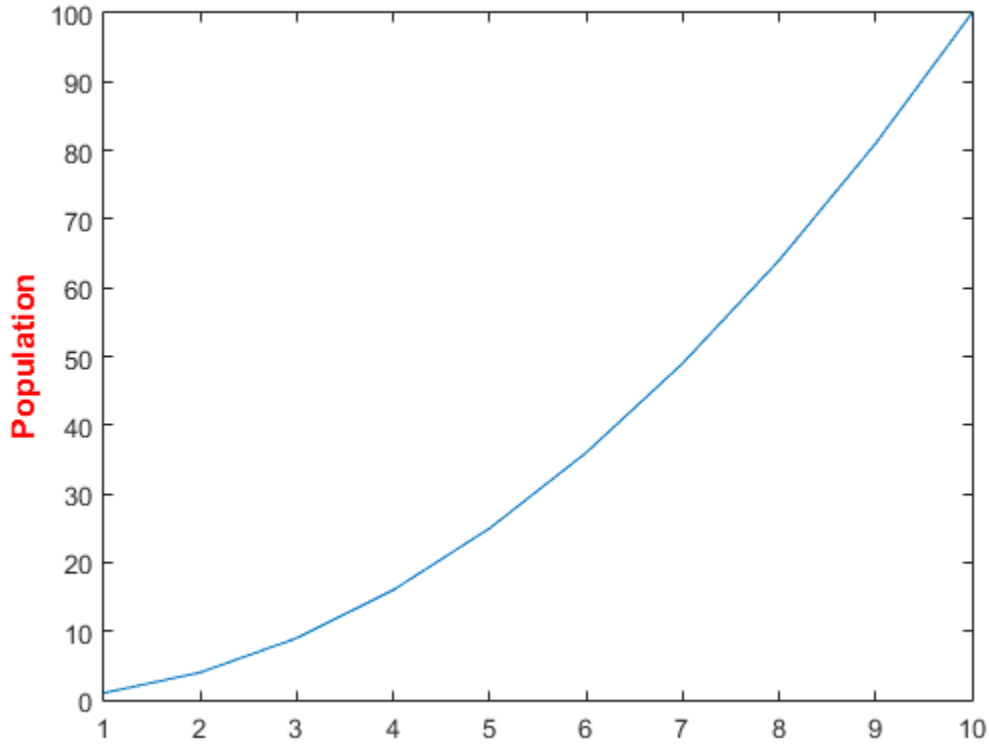
```
t = linspace(0,1);
y = exp(t);
plot(t,y)
xlabel('t_{seconds}')
ylabel('e^t')
```



Create y-Axis Label and Set Font Properties

Use `Name,Value` pairs to set the font size, font weight, and text color properties of the *y*-axis label.

```
figure
plot((1:10).^2)
ylabel('Population','FontSize',12,...
       'FontWeight','bold','Color','r')
```

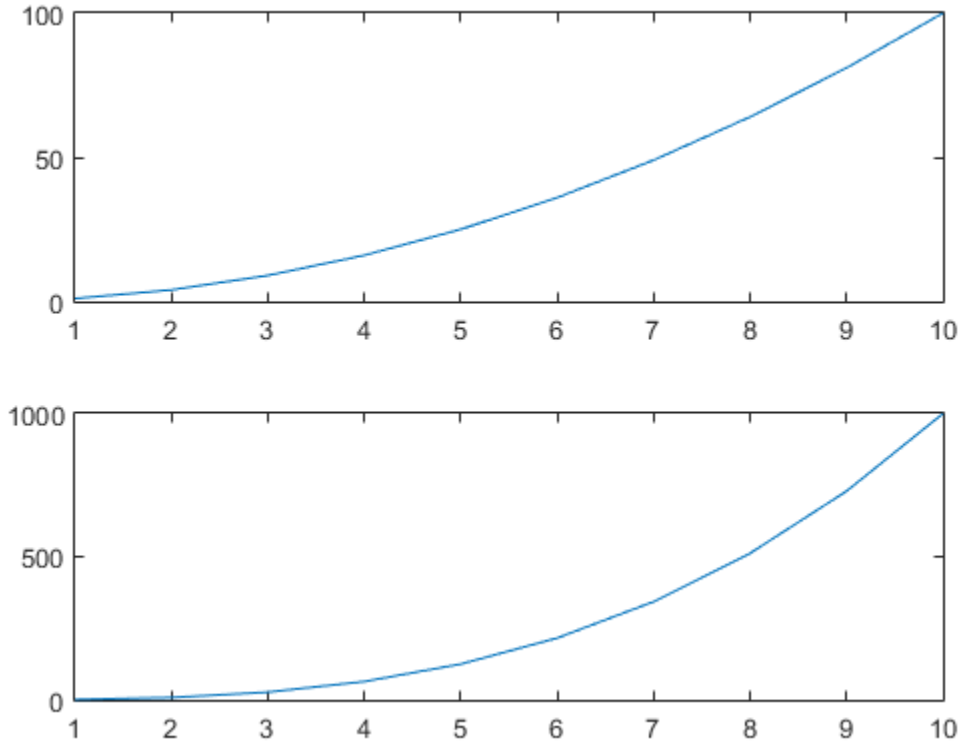


`'FontSize',12` displays the label text in 12-point font. `'FontWeight','bold'` makes the text bold. `'Color','r'` sets the text color to red.

Label y-Axis of Specific Axes

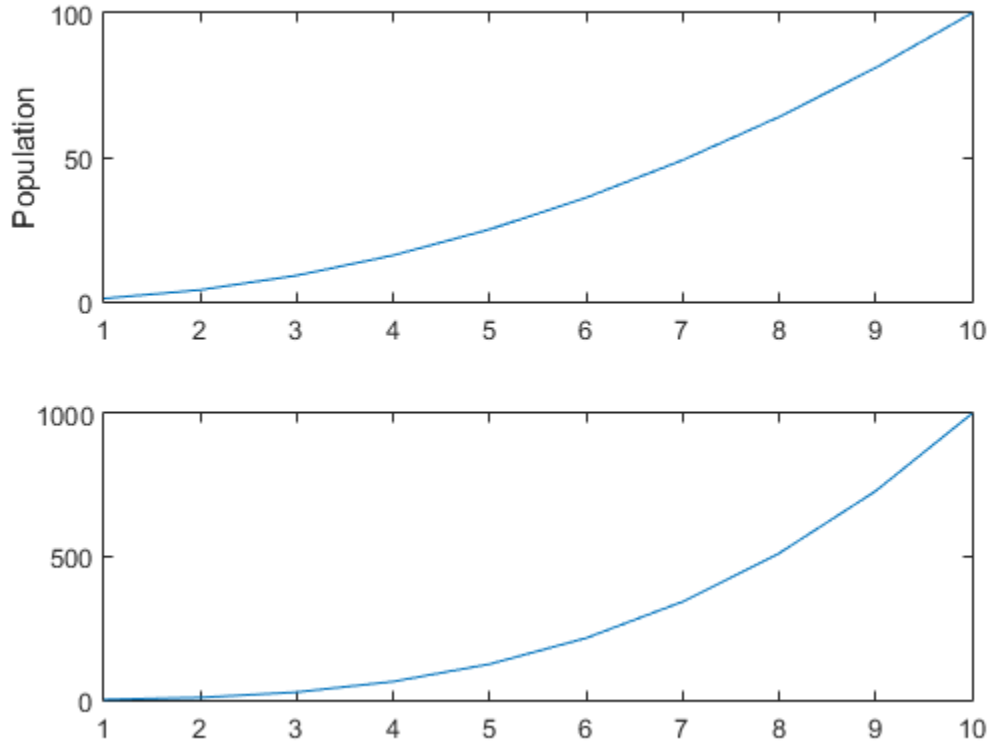
Create two subplots and return the axes handles, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
plot((1:10).^2)
s(2) = subplot(2,1,2);
plot((1:10).^3)
```



Label the *y*-axis of the top plot by referring to its axes handle, `s(1)`.

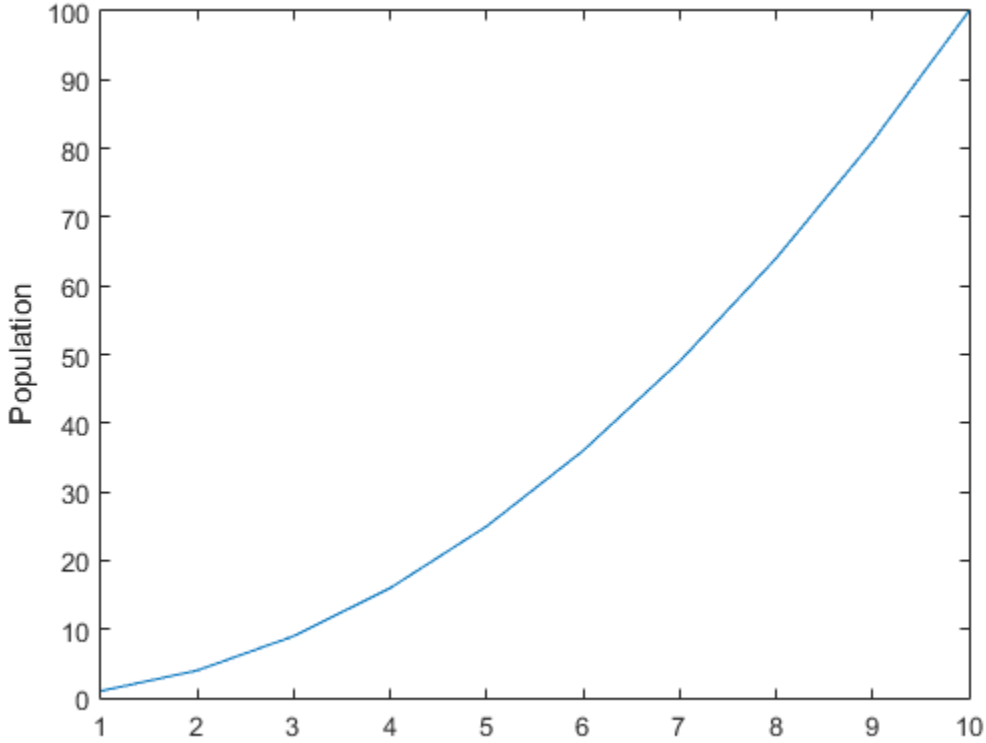
```
ylabel(s(1),'Population')
```



Label y-Axis and Return Object Handle

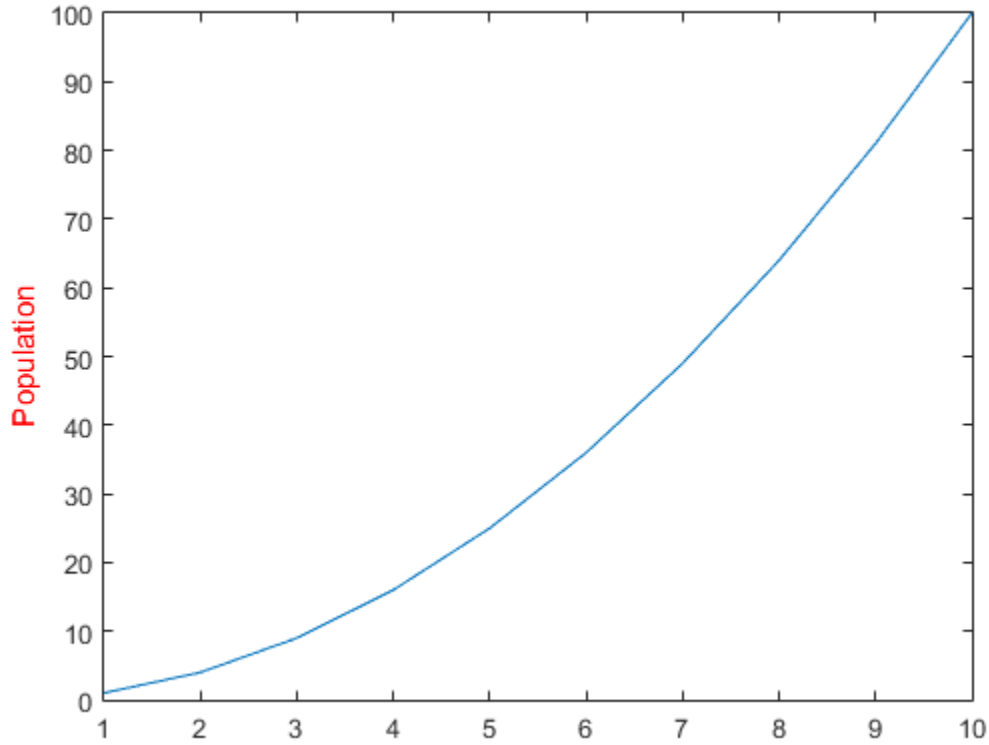
Label the *y*-axis and return the handle to the text object used as the label.

```
plot((1:10).^2)
t = ylabel('Population');
```



Set the color of the label to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
t.Color = 'red';
```



- “Add Text to Graph Interactively”

Input Arguments

`str` — Axis label

character array | cell array | numeric value

Axis label, specified as a character array, cell array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: 123

To include numeric variables with text in a label, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the Interpreter property.

To create multiline labels:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create a string with a new line character, such as `sprintf('first line \n second line')`.

Numeric labels are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note: The words `default`, `factory`, and `remove` are reserved words that will not appear in a label when quoted as a normal string. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

---

ax — Axes object

axes object

Axes object. If you do not specify an axes, then the `ylabel` function uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',12` specifies red, 12-point font.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text Properties.

`'FontSize'` — Font size

`11` (default) | scalar value greater than `0`

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Setting the font size properties for the associated axes also affects the label font size. The label font size updates to equal the axes font size times the label scale factor. The `FontSize` property of the axes contains the axes font size. The `LabelFontSizeMultiplier` property of the axes contains the label scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the *y*-axis label font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'FontWeight'` — Thickness of text characters

`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

`'FontName'` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`'Color'` — Text color

`[0.15 0.15 0.15]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. If you set the color to `'none'`, then the text is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`'Interpreter'` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`

Modifier	Description	Example of String
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsui`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ε	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrighta`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	Θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

Output Arguments

h — Text object

text object

Text object used as the *y*-axis label. Use `h` to access and modify properties of the label after its created.

See Also

Functions

`num2str` | `text` | `title` | `xlabel` | `zlabel`

Properties

Text Properties

Introduced before R2006a

ymd

Year, month, and day numbers of datetime

Syntax

```
[y,m,d] = ymd(t)
```

Description

`[y,m,d] = ymd(t)` returns the year, month, and day numbers of the datetime values in `t` as separate numeric arrays. The `y`, `m`, and `d` outputs are the same size as `t`, and contain integer values.

The `ymd` function is equivalent to calling the `year`, `month`, and `day` functions on the same datetime array.

Examples

Find Year, Month, and Day Numbers of Dates

```
t = datetime(2013,05,31):calmonths(3):datetime(2014,06,15)
```

```
t =

   31-May-2013   31-Aug-2013   30-Nov-2013   28-Feb-2014   31-May-2014
```

```
[y,m,d] = ymd(t)
```

```
y =

        2013        2013        2013        2014        2014

m =
```

```
     5     8    11     2     5

d =

    31    31    30    28    31
```

`ymd` returns the year, month, and day values in separate arrays.

Input Arguments

`t` — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

Output Arguments

`y` — ISO year numbers

scalar | vector | matrix | multidimensional array

ISO year numbers, returned as a scalar, vector, matrix, or multidimensional array of integer values. ISO year numbers include a year zero and represent years BCE using negative values. `y` is of type `double` and is the same size as `t`.

`m` — Month numbers

scalar | vector | matrix | multidimensional array

Month numbers, returned as a scalar, vector, matrix, or multidimensional array of integer values from 1 to 12. `m` is of type `double` and is the same size as `t`.

`d` — Day of month numbers

scalar | vector | matrix | multidimensional array

Day of month numbers, returned as a scalar, vector, matrix, or multidimensional array of integer values from 1 to 28, 29, 30, or 31, depending on the month and year. `d` is of type `double` and is the same size as `t`.

See Also

`day` | `hms` | `month` | `quarter` | `week` | `year`

Introduced in R2014b

yyyymmdd

Convert MATLAB datetime to YYYYMMDD numeric value

Syntax

```
d = yyyymmdd(t)
```

Description

`d = yyyymmdd(t)` returns a `double` array containing integers whose digits represent the datetime values in `t`. For example, the date July 16, 2014 is converted to the integer 20140716. The conversion is performed this way:

```
d = 10000*year(t) + 100*month(t) + day(t)
```

Examples

Convert Datetime Array to YYYYMMDD Numeric Values

Create an array of YYYYMMDD numeric values that represent dates.

```
d = [20140628 20140701 20140704]
```

```
d =

    20140628    20140701    20140704
```

Convert the dates to datetime values.

```
t = datetime(d,'ConvertFrom','yyyymmdd')
```

```
t =

   28-Jun-2014 00:00:00   01-Jul-2014 00:00:00   04-Jul-2014 00:00:00
```

Convert the datetime values back to YYYYMMDD numeric values.

```
d2 = yyyymmdd(t)
```

```
d2 =

    20140628     20140701     20140704
```

Input Arguments

t — Input date and time

`datetime` array

Input date and time, specified as a `datetime` array.

See Also

`datenum` | `datetime` | `exceltime` | `juliandate` | `posixtime`

Introduced in R2014b

zlabel

Label z-axis

Syntax

```
zlabel(str)
zlabel(str,Name,Value)

zlabel(ax, ___ )

h = zlabel( ___ )
```

Description

`zlabel(str)` labels the *z*-axis of the current axes with the string, `str`. Reissuing the `zlabel` command causes the new label to replace the old label.

`zlabel(str,Name,Value)` additionally specifies the text object properties using one or more `Name,Value` pair arguments.

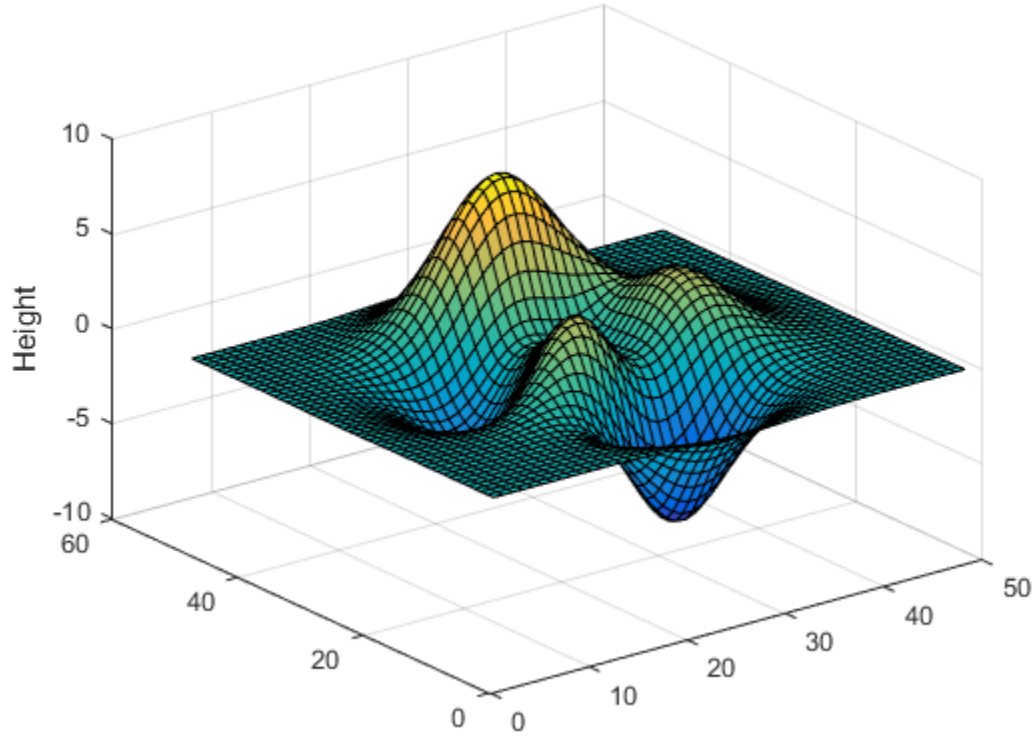
`zlabel(ax, ___ )` adds the label to the axes specified by `ax`. This syntax allows you to specify the axes to which to add a label. `ax` can precede any of the input argument combinations in the previous syntaxes.

`h = zlabel( ___ )` returns the handle to the text object used as the *z*-axis label. The handle is useful when making future modifications to the label.

Examples

Label z-Axis with String

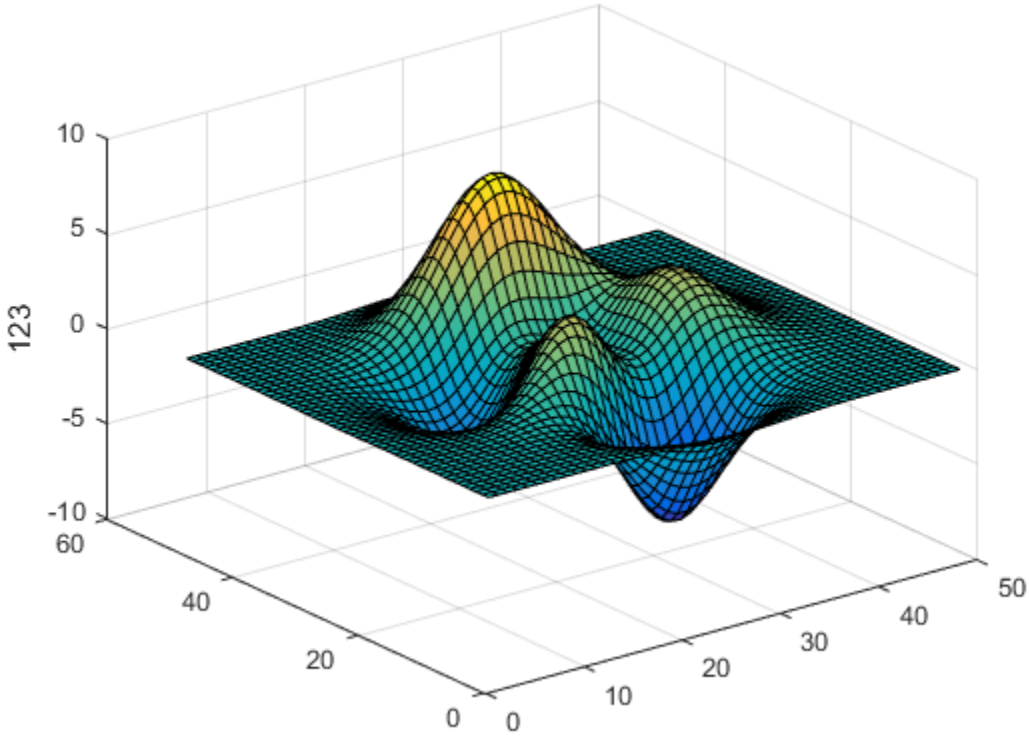
```
figure
surf(peaks)
zlabel('Height')
```



MATLAB® displays `Height` beside the *z*-axis.

Label z-Axis with Numeric Input

```
figure
surf(peaks)
zlabel(123)
```

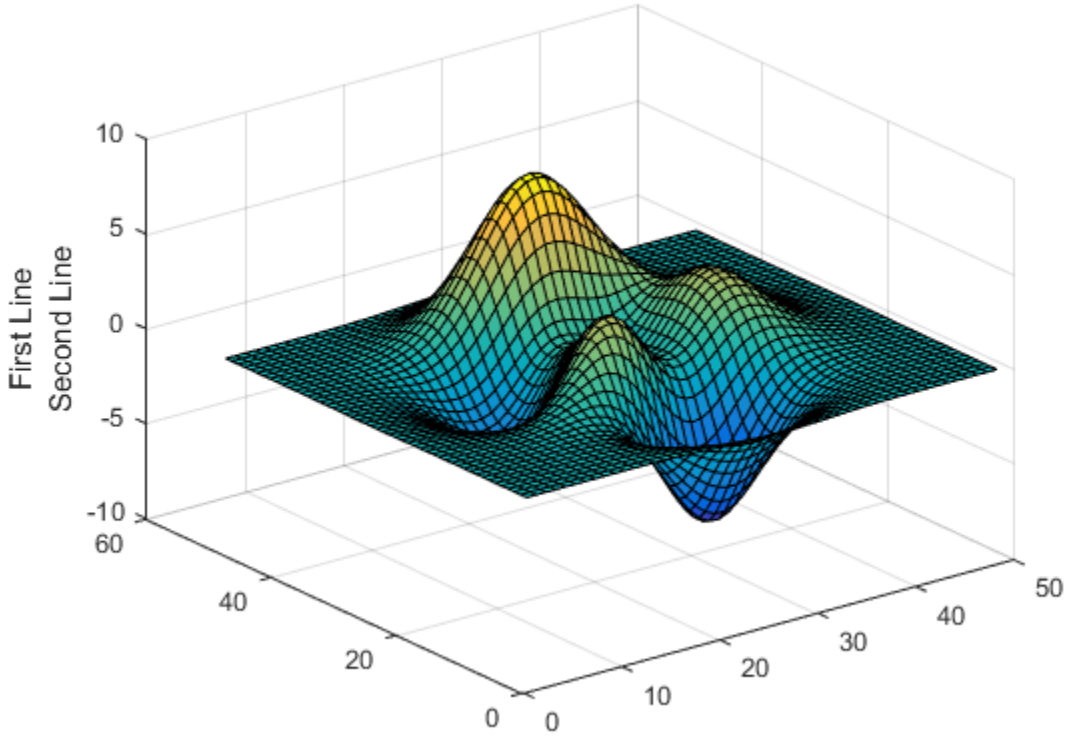


MATLAB® displays `123` beside the *z*-axis.

Create Multiline z-Axis Label

Create a multiline label using a multiline cell array.

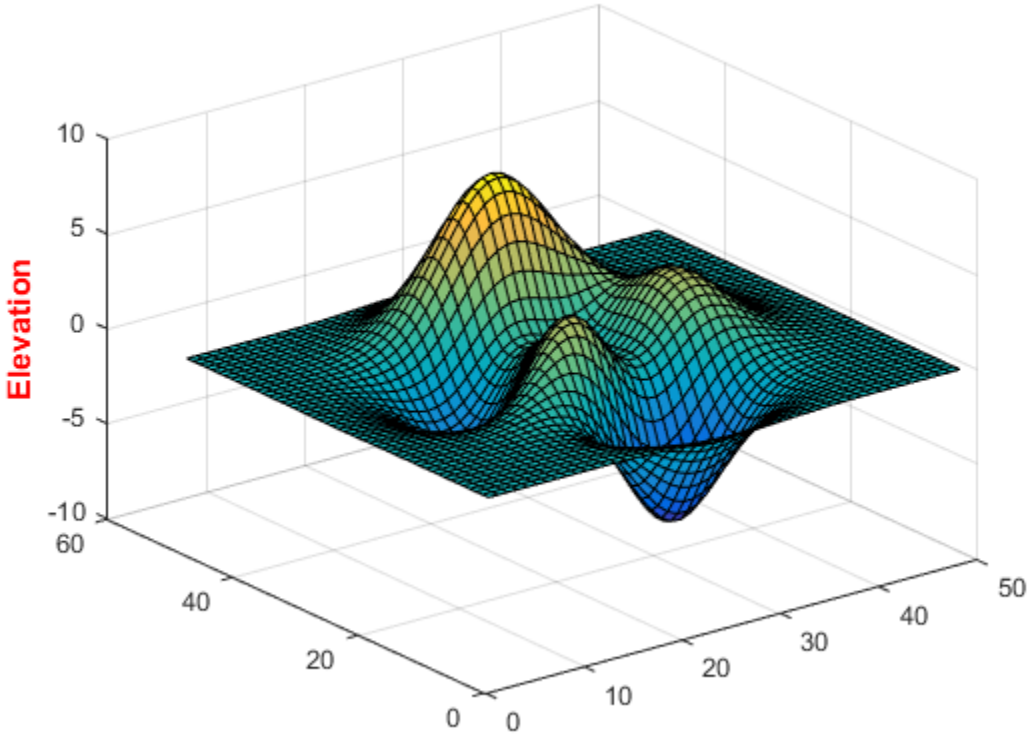
```
figure
surf(peaks)
zlabel({'First Line';'Second Line'})
```



Label z-Axis and Set Font Properties

Use `Name,Value` pairs to set the font size, font weight, and text color properties of the *z*-axis label.

```
figure
surf(peaks)
zlabel('Elevation','FontSize',12,...
       'FontWeight','bold','Color','r')
```

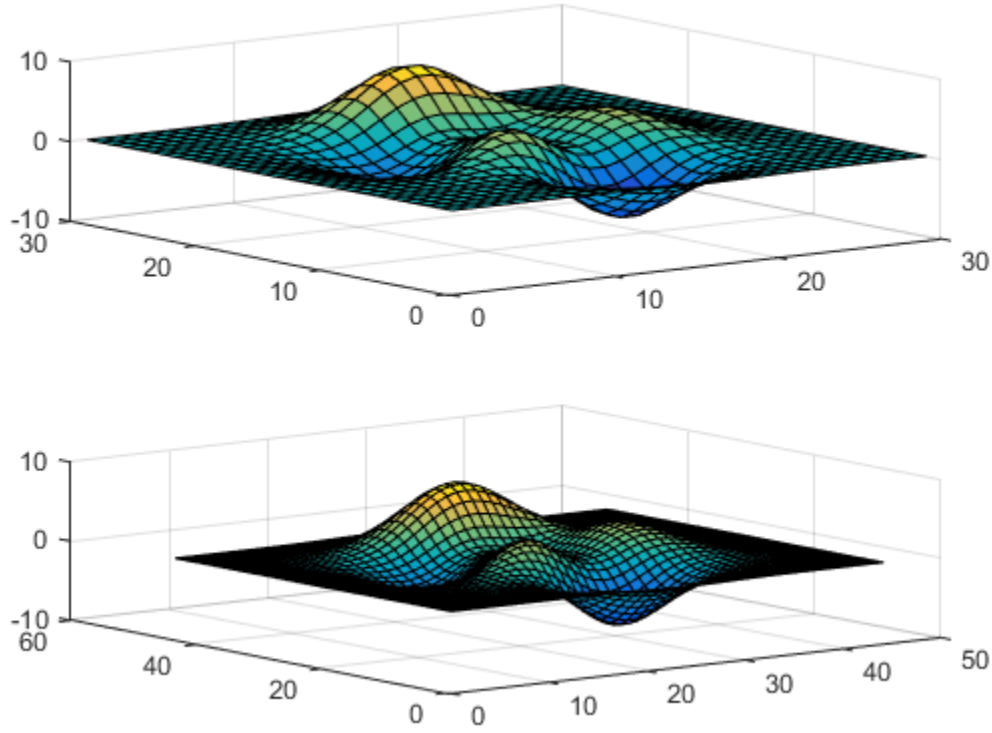


`'FontSize',12` displays the label text in 12-point font. `'FontWeight','bold'` makes the text bold. `'Color','r'` sets the text color to red.

Label z-Axis of Specific Axes

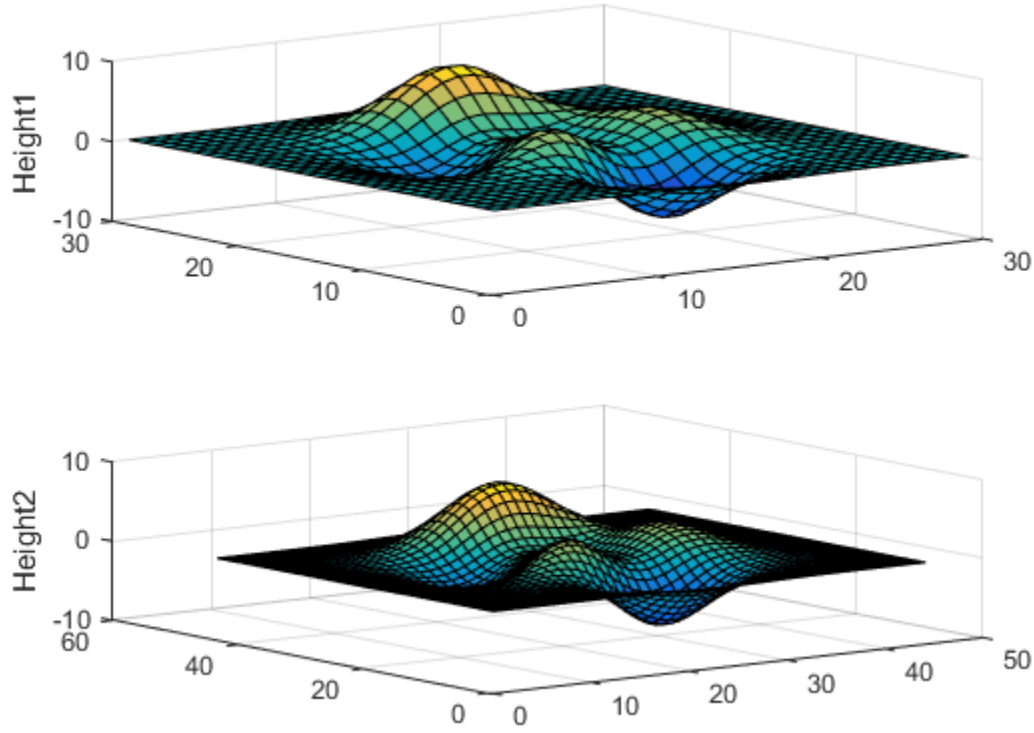
Create two subplots and return the handles to the axes objects, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
surf(peaks(30))
s(2) = subplot(2,1,2);
surf(peaks(45))
```



Label the *z*-axis of each plot by referring to the axes handles, `s(1)` and `s(2)`.

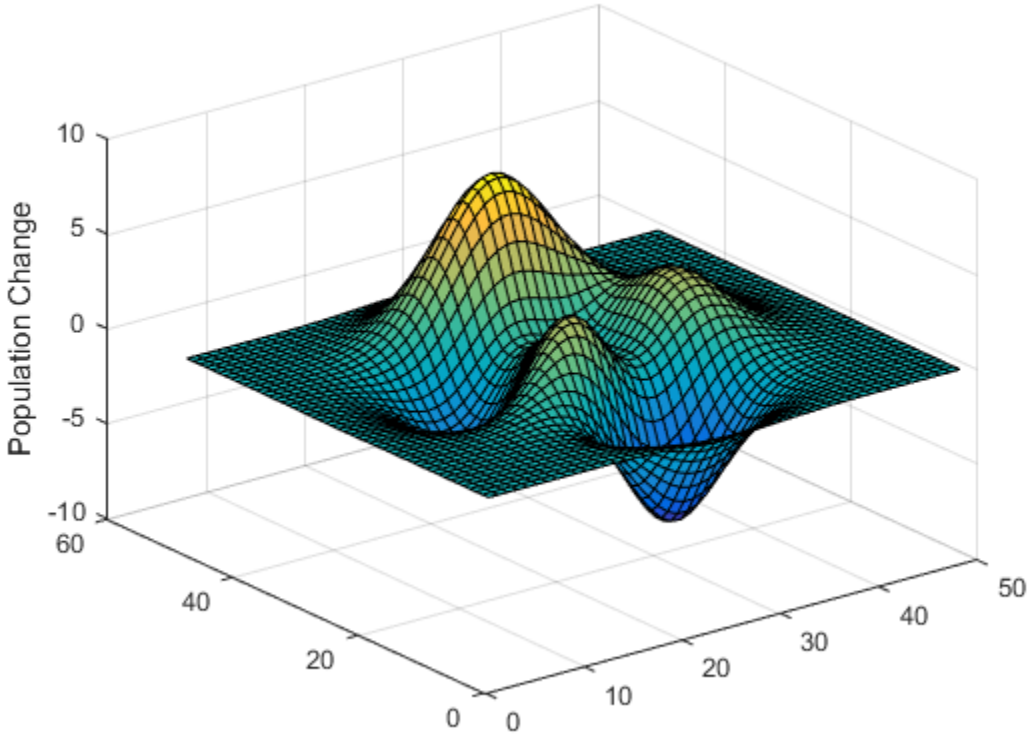
```
zlabel(s(1),'Height1')
zlabel(s(2),'Height2')
```



Label z-Axis and Return Object Handle

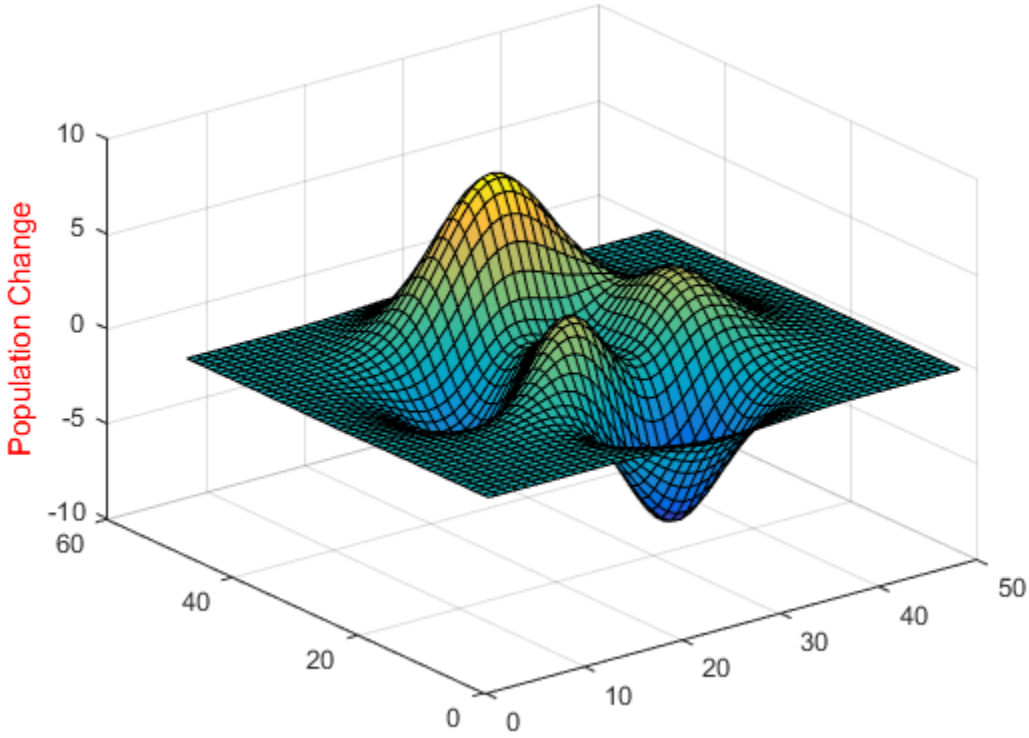
Label the *z*-axis and return the text object used as the label.

```
surf(peaks)
t = zlabel('Population Change');
```



Set the color of the label to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

```
t.Color = 'red';
```



- “Add Text to Graph Interactively”

Input Arguments

`str` — Axis label

character array | cell array | numeric value

Axis label, specified as a character array, cell array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: 123

To include numeric variables with text in a label, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the Interpreter property.

To create multiline labels:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create a string with a new line character, such as `sprintf('first line \n second line')`.

Numeric labels are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note: The words `default`, `factory`, and `remove` are reserved words that will not appear in a label when quoted as a normal string. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

---

ax — Axes object

axes object

Axes object. If you do not specify an axes, then the `zlabel` function uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',12` specifies red, 12-point font.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text Properties.

`'FontSize'` — Font size

`11` (default) | scalar value greater than `0`

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Setting the font size properties for the associated axes also affects the label font size. The label font size updates to equal the axes font size times the label scale factor. The `FontSize` property of the axes contains the axes font size. The `LabelFontSizeMultiplier` property of the axes contains the label scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the *z*-axis label font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'FontWeight'` — Thickness of text characters

`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker characters outlines than normal

MATLAB uses the FontWeight property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

Note: The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

`'FontName'` — Font name

`'Helvetica'` (default) | `'FixedWidth'` | system supported font name

Font name, specified as the name of the font to use or the string `'FixedWidth'`. To display and print properly, the font name must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

`'Color'` — Text color

`[0.15 0.15 0.15]` (default) | RGB triplet | color string | `'none'`

Text color, specified as a three-element RGB triplet, a color string, or `'none'`. If you set the color to `'none'`, then the text is invisible.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists RGB triplet values that have equivalent color strings.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black`	`'k'`	`[0 0 0]`

Example: `[0.5 0.6 0.7]`

Example: `'blue'`

`'Interpreter'` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret text strings using a subset of TeX markup. This is the default value.
- `'latex'` — Interpret text strings using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text string.

This table lists the supported modifiers when the `Interpreter` property is set to `'tex'`. Modifiers remain in effect until the end of the string, except for superscripts and subscripts, which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example of String
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`

Modifier	Description	Example of String
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters when the interpreter is set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	Φ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	ß	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsui`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ε	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrighta`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	Θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text string.

The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at http://www.latex-project.org/.

Output Arguments

h — Text object

text object

Text object used as the *z*-axis label. Use `h` to access and modify properties of the label after its created.

See Also

Functions

`num2str` | `text` | `title` | `xlabel` | `ylabel`

Properties

Text Properties

Introduced before R2006a

xlim

Set or query *x*-axis limits

Syntax

```
xlim(limits)

xlim auto
xlim manual

xl = xlim
m = xlim('mode')

___ = xlim(ax, ___ )
```

Description

`xlim(limits)` specifies the *x*-axis limits for the current axes. Specify `limits` as a two-element vector of the form `[xmin xmax]`, where `xmax` is a numeric value greater than `xmin`.

`xlim auto` lets the axes choose the *x*-axis limits. The axes chooses limits that span the range of the plotted data. This command sets the `XLimMode` property for the axes to `'auto'`.

`xlim manual` freezes the limits at the current values. Use this option if you want to retain the current limits when adding new data to the axes using the `hold on` command. This command sets the `XLimMode` property for the axes to `'manual'`.

`xl = xlim` returns a two-element vector containing the current limits.

`m = xlim('mode')` returns the current value of the limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

`___ = xlim(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify an axes with any of the input or output arguments in previous syntaxes.

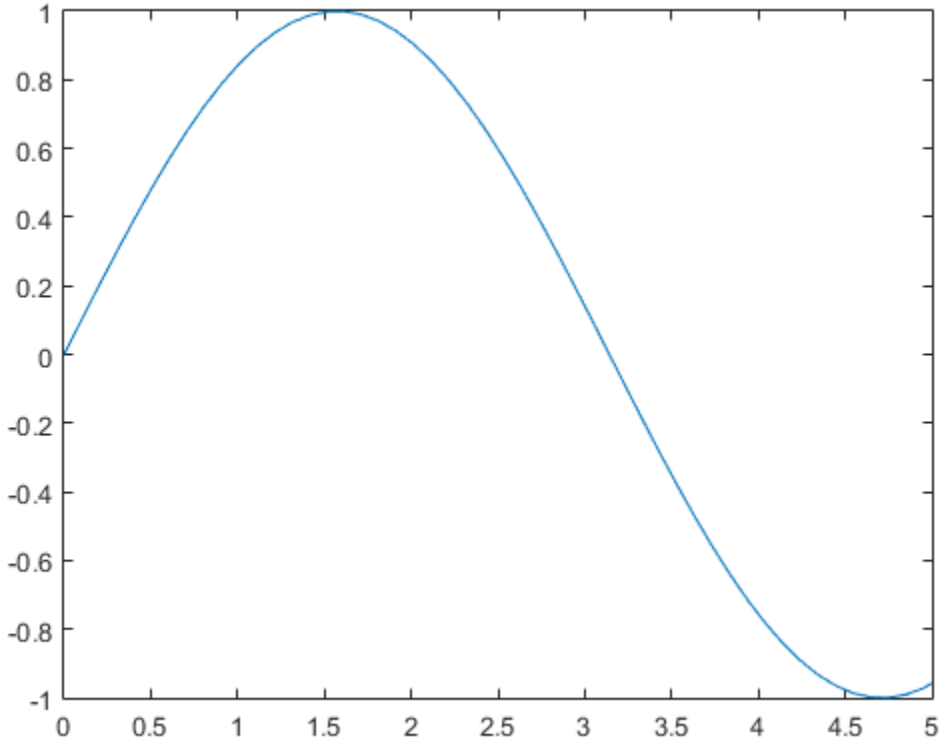
Use single quotes around input arguments that are character strings, for example, `xlim(ax,'auto')` and `xlim(ax,'manual')`.

Examples

Set *x*-Axis Limits

Plot a line and set the *x*-axis limits to range from 0 to 5.

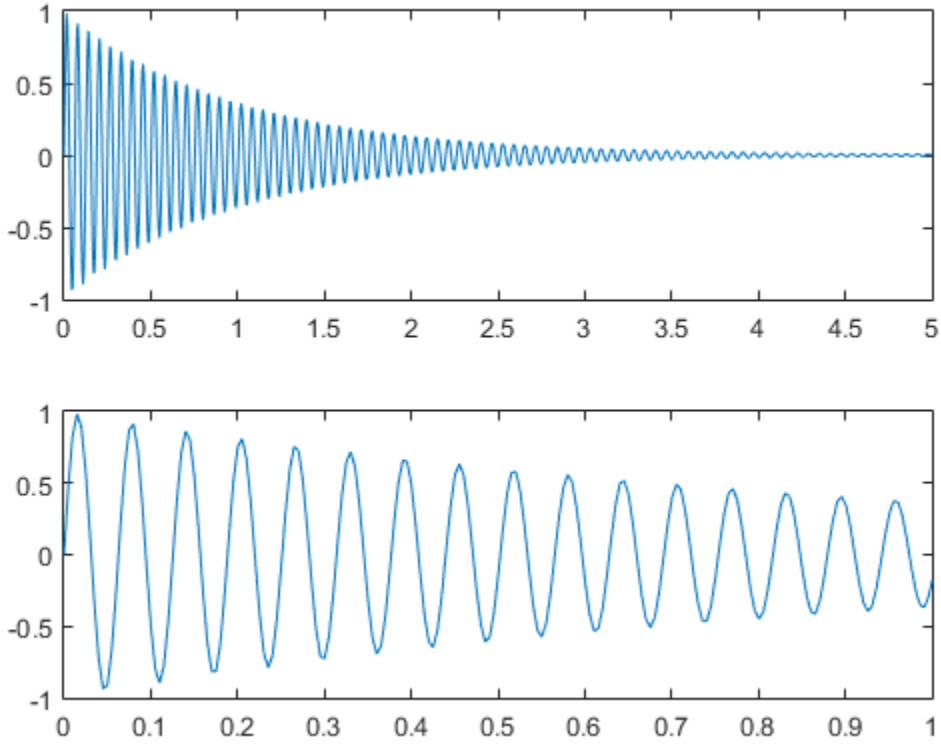
```
x = linspace(0,10);
y = sin(x);
plot(x,y)
xlim([0 5])
```



Set *x*-Axis Limits for Specific Axes

Create a figure with two subplots and plot the same data in each subplot. Set the *x*-axis limits for the bottom subplot.

```
x = linspace(0,5,1000);
y = sin(100*x)./exp(x);
ax1 = subplot(2,1,1);
plot(x,y)

ax2 = subplot(2,1,2);
plot(x,y)
xlim(ax2,[0 1])
```

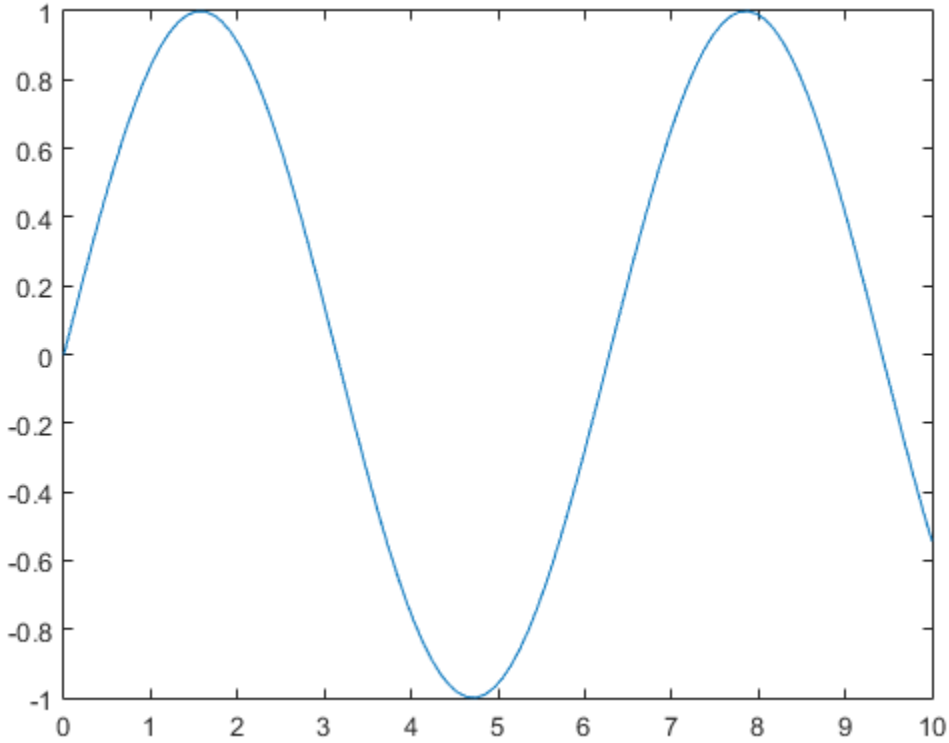


Maintain Current *x*-Axis Limits

Use manual mode to maintain the current x-axis limits when you add more plots to the axes.

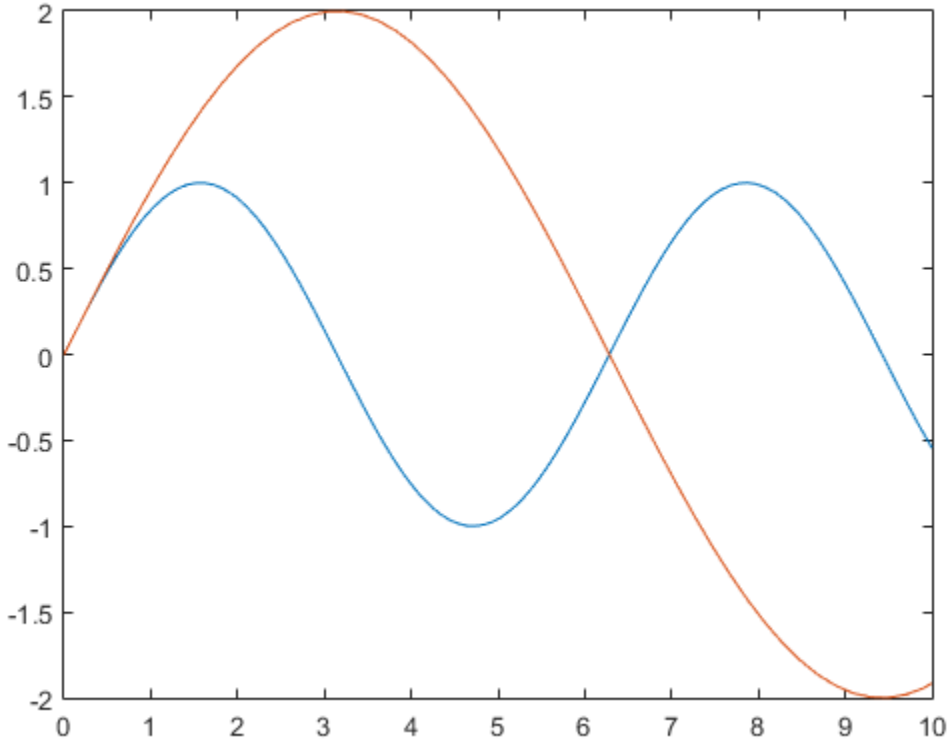
First, plot a line.

```
x = linspace(0,10);
y = sin(x);
plot(x,y);
```



Set the *x*-axis limits mode to manual so that the limits do not change. Use `hold on` to add a second plot to the axes.

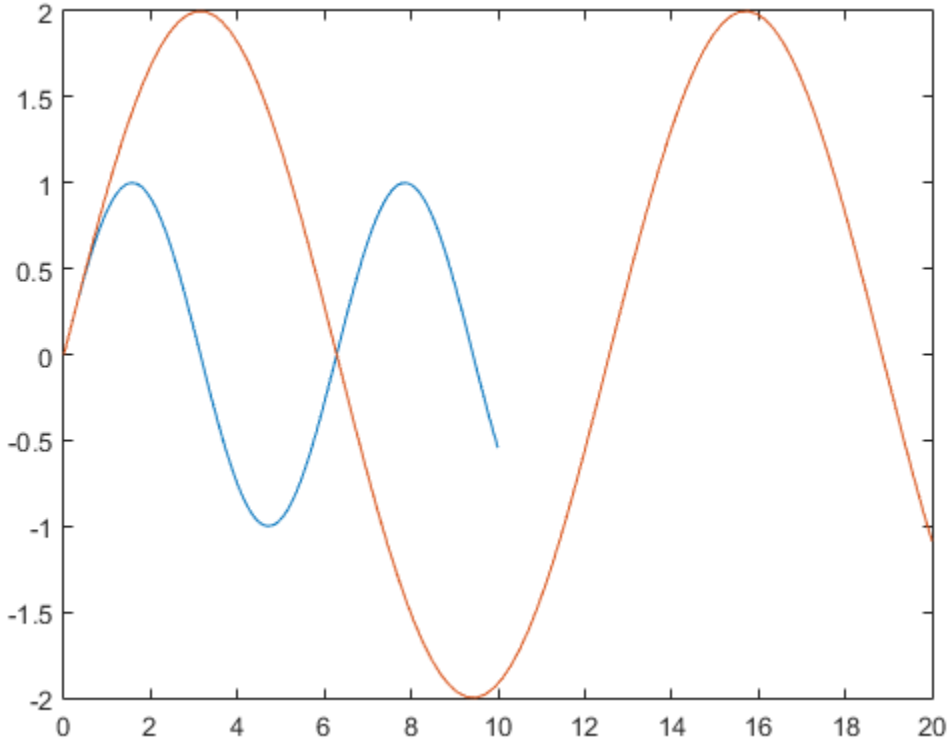
```
xlim manual
hold on
plot(2*x,2*y)
hold off
```



The *x*-axis limits do not update to incorporate the new plot.

Switch back to automatically updated limits by resetting the mode to automatic.

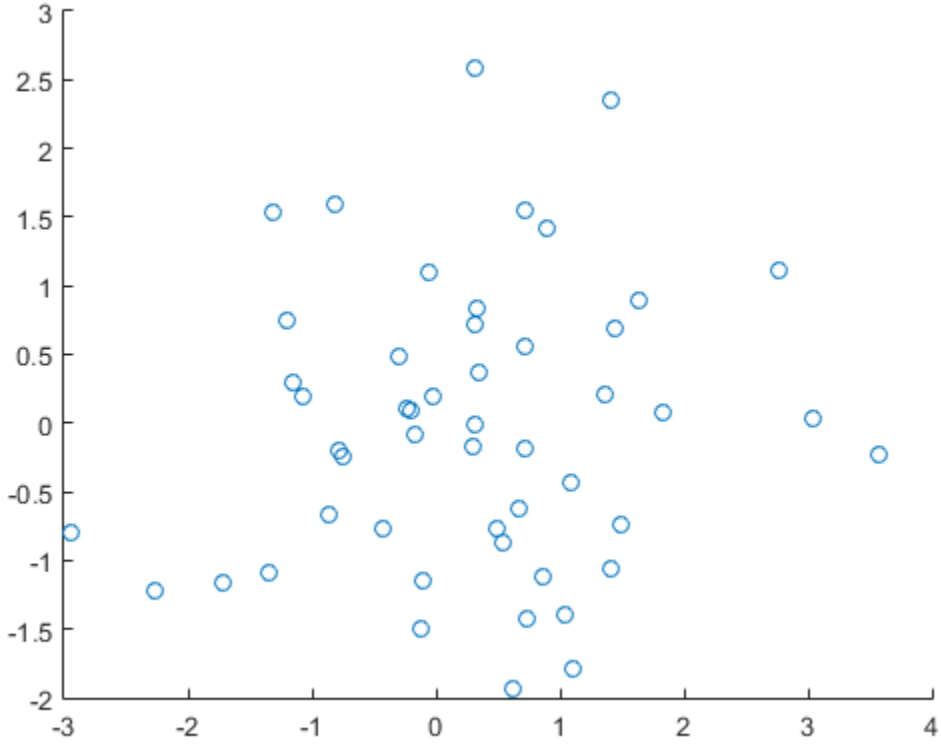
```
xlim auto
```



Return *x*-Axis Limits

Create a scatter plot of random data. Return the values of the *x*-axis limits.

```
x = randn(50,1);
y = randn(50,1);
scatter(x,y)
```




```
xl = xlim
```

```
xl =

    -3     4
```

Input Arguments

`limits` — Minimum and maximum limits

two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[xmin xmax]`. Specifying the *x*-axis limits sets the *x*-axis limits mode to manual. The `XLim` and `XLimMode` properties for the corresponding axes store the *x*-axis limits and the limits mode, respectively.

Changing the *x*-axis limits can cause other limits to change, unless their corresponding mode properties are set to manual.

Example: `[0 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ax` — Axes object

axes object

Axes object. If you do not specify an axes, then `xlim` sets the limits for the current axes (`gca`).

Output Arguments

`xl` — Current limits

two-element vector

Current limits, returned as a two-element vector of the form `[xmin xmax]`.

Querying the limits returns the `XLim` property value for the corresponding axes.

`m` — Current limits mode

`'auto'` | `'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — The limits automatically update to reflect changes in the data.
- `'manual'` — The limits do not update to reflect changes in the data.

Querying the *x*-axis limits mode returns `XLimMode` property value for the corresponding axes.

See Also

`axis` | `grid` | `hold` | `xlabel` | `ylim` | `zlim`

Introduced before R2006a

ylim

Set or query *y*-axis limits

Syntax

```
ylim(limits)

ylim auto
ylim manual

yl = ylim
m = ylim('mode')

___ = ylim(ax, ___ )
```

Description

`ylim(limits)` sets the *y*-axis limits for the current axes. Specify `limits` as a two-element vector of the form `[ymin ymax]`, where `ymax` is a numeric value greater than `ymin`.

`ylim auto` lets the axes choose the *y*-axis limits. The axes chooses limits that span the range of the plotted data. This command sets the `YLimMode` property for the axes to `'auto'`.

`ylim manual` freezes the limits at the current values. Use this option if you want to retain the current limits when adding new data to the axes using the `hold on` command. This command sets the `YLimMode` property for the axes to `'manual'`.

`yl = ylim` returns a two-element vector containing the current limits.

`m = ylim('mode')` returns the current value of the limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

`___ = ylim(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify an axes with any of the input or output arguments in previous syntaxes.

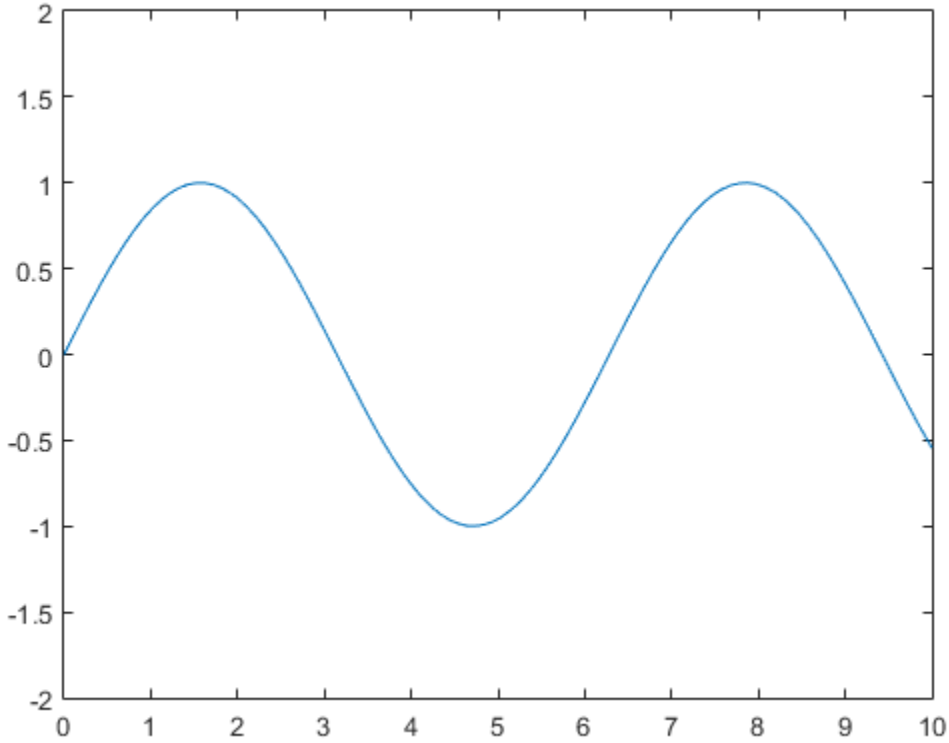
Use single quotes around input arguments that are character strings, for example, `ylim(ax,'auto')` and `ylim(ax,'manual')`.

Examples

Set *y*-Axis Limits

Plot a line and set the *y*-axis limits to range from -2 to 2.

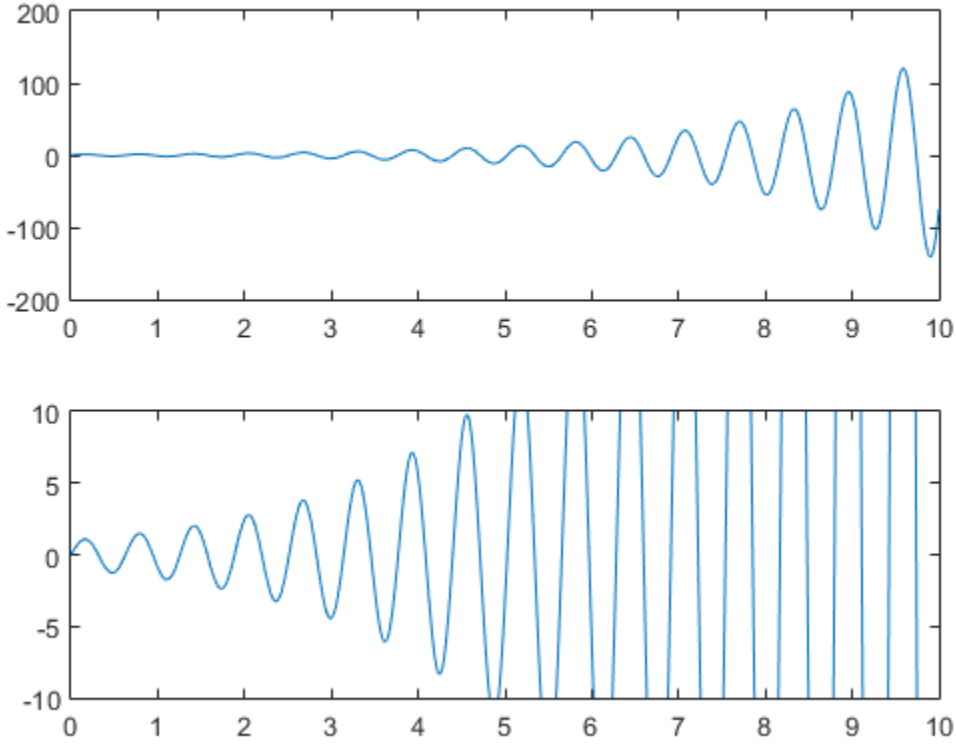
```
x = linspace(0,10);
y = sin(x);
plot(x,y)
ylim([-2 2])
```



Set *y*-Axis Limits for Specific Axes

Create a figure with two subplots and plot the same data in each subplot. Set the *y*-axis limits for the bottom subplot.

```
x = linspace(0,10,1000);
y = sin(10*x).*exp(.5*x);
ax1 = subplot(2,1,1);
plot(x,y)

ax2 = subplot(2,1,2);
plot(x,y)
ylim(ax2,[-10 10])
```

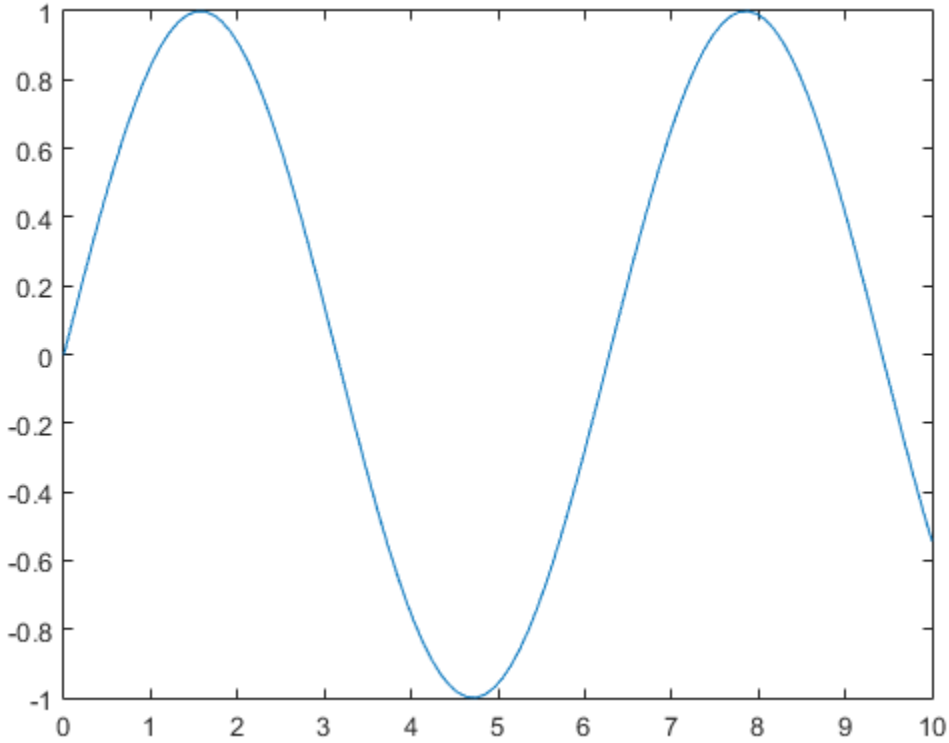


Maintain Current *y*-Axis Limits

Use manual mode to maintain the current *y*-axis limits when you add more plots to the axes.

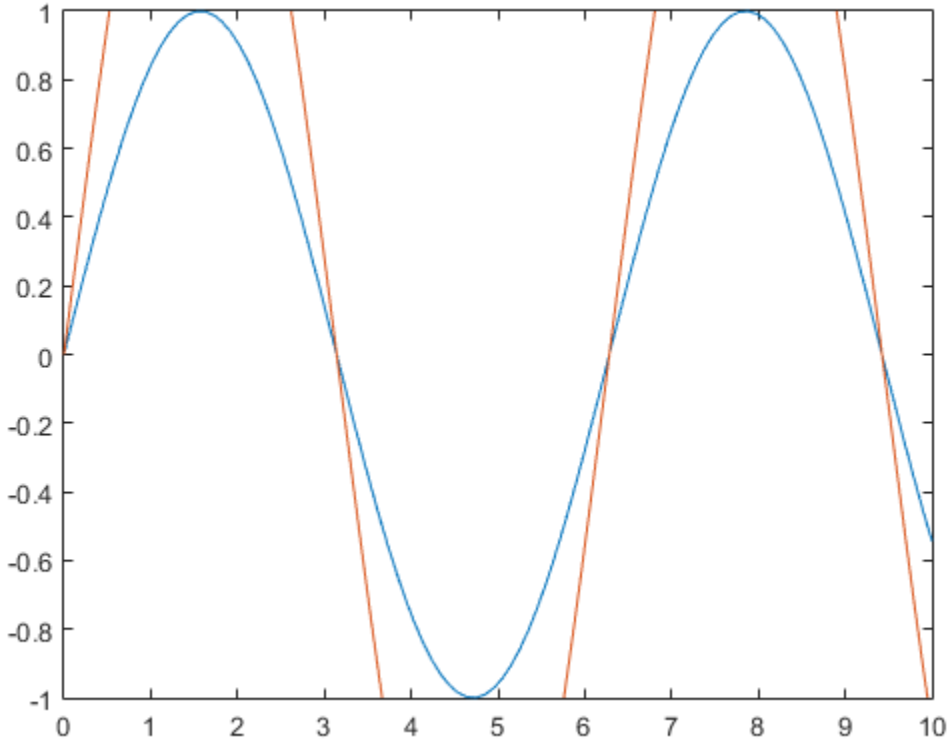
First, plot a line.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
```



Set the *y*-axis limits mode to manual so that the limits to not change. Use `hold on` to add a second plot to the axes.

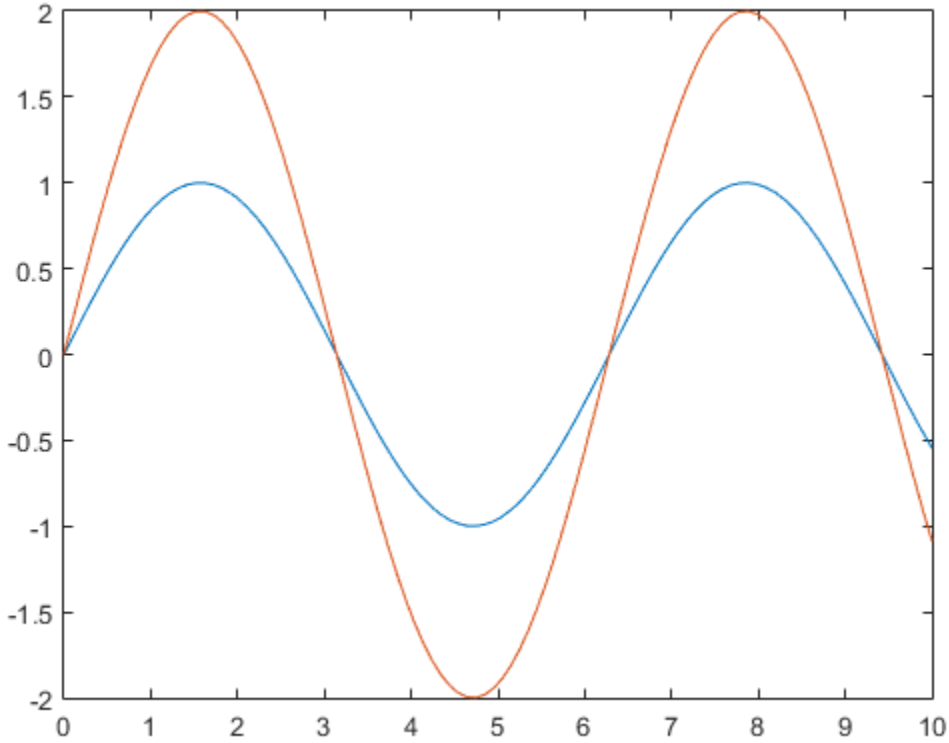
```
ylim manual
hold on
y2 = 2*sin(x);
plot(x,y2)
hold off
```



The *y*-axis limits do not update to incorporate the new plot.

Switch back to automatically updated limits by resetting the mode to automatic.

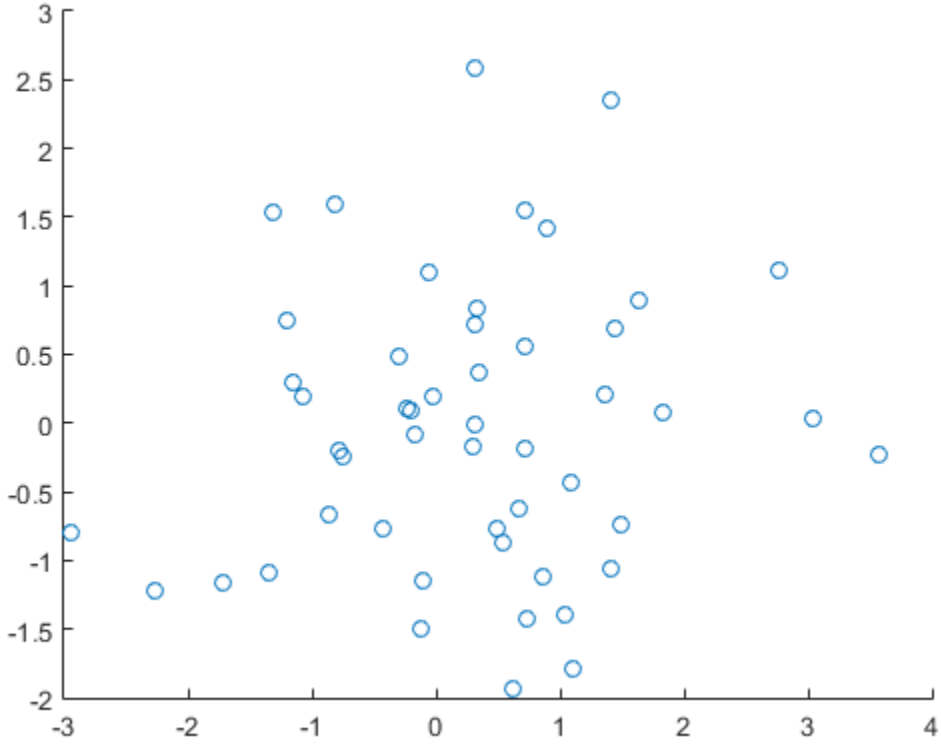
```
ylim auto
```



Return *y*-Axis Limits

Create a scatter plot of random data. Return the values of the *y*-axis limits.

```
x = randn(50,1);
y = randn(50,1);
scatter(x,y)
```




```
yl =  ylim

yl =

    -2     3
```

Input Arguments

`limits` — Minimum and maximum limits

two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[ymin ymax]`. Specifying the *y*-axis limits sets the *y*-axis limits mode to manual. The `YLim` and `YLimMode` properties for the corresponding axes store the *y*-axis limits and the limits mode, respectively.

Changing the *y*-axis limits can cause other limits to change, unless their corresponding mode properties are set to manual.

Example: `[0 1]`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

`ax` — Axes object
axes object

Axes object. If you do not specify an axes, then `ylim` sets the limits for the current axes (`gca`).

Output Arguments

`yl` — Current limits
two-element vector

Current limits, returned as a two-element vector of the form `[ymin ymax]`.

Querying the limits returns the `YLim` property value for the corresponding axes.

`m` — Current limits mode
`'auto' | 'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — The limits update to reflect changes in the data.
- `'manual'` — The limits do not update to reflect changes in the data.

Querying the *y*-axis limits mode returns `YLimMode` property value for the corresponding axes.

See Also
`axis | grid | hold | xlim | ylabel | zlim`

Introduced before R2006a

zlim

Set or query *z*-axis limits

Syntax

```
zlim(limits)

zlim auto
zlim manual

zl = zlim
m = zlim('mode')

___ = zlim(ax, ___ )
```

Description

`zlim(limits)` sets the *z*-axis limits for the current axes. Specify `limits` as a two-element vector of the form `[zmin zmax]`, where `zmax` is a numeric value greater than `zmin`.

`zlim auto` lets the axes choose the *z*-axis limits. The axes chooses limits that span the range of the plotted data. This command sets the `ZLimMode` property for the axes to `'auto'`..

`zlim manual` freezes the limits at the current values. Use this option if you want to retain the current limits when adding new data to the axes using the `hold on` command. This command sets the `ZLimMode` property for the axes to `'manual'`.

`zl = zlim` returns a two-element vector containing the current limits.

`m = zlim('mode')` returns the current value of the limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

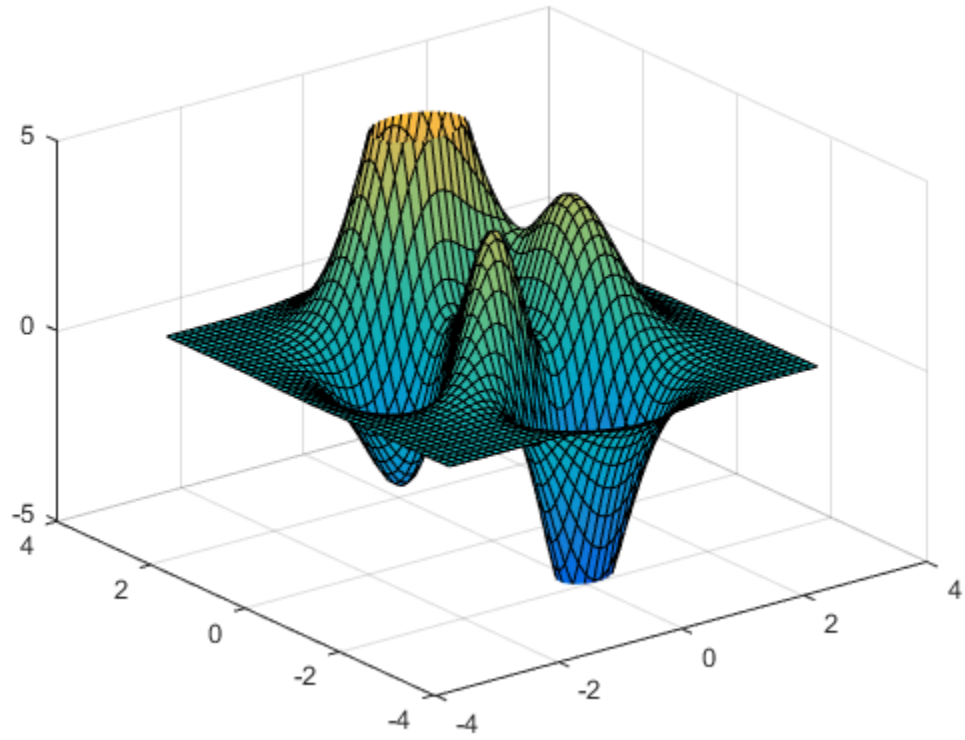
`___ = zlim(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify an axes with any of the input or output arguments in previous syntaxes. Use single quotes around input arguments that are character strings, for example, `zlim(ax,'auto')` and `zlim(ax,'manual')`.

Examples

Set *z*-Axis Limits

Plot a surface and set the *z*-axis limits to range from -5 to 5.

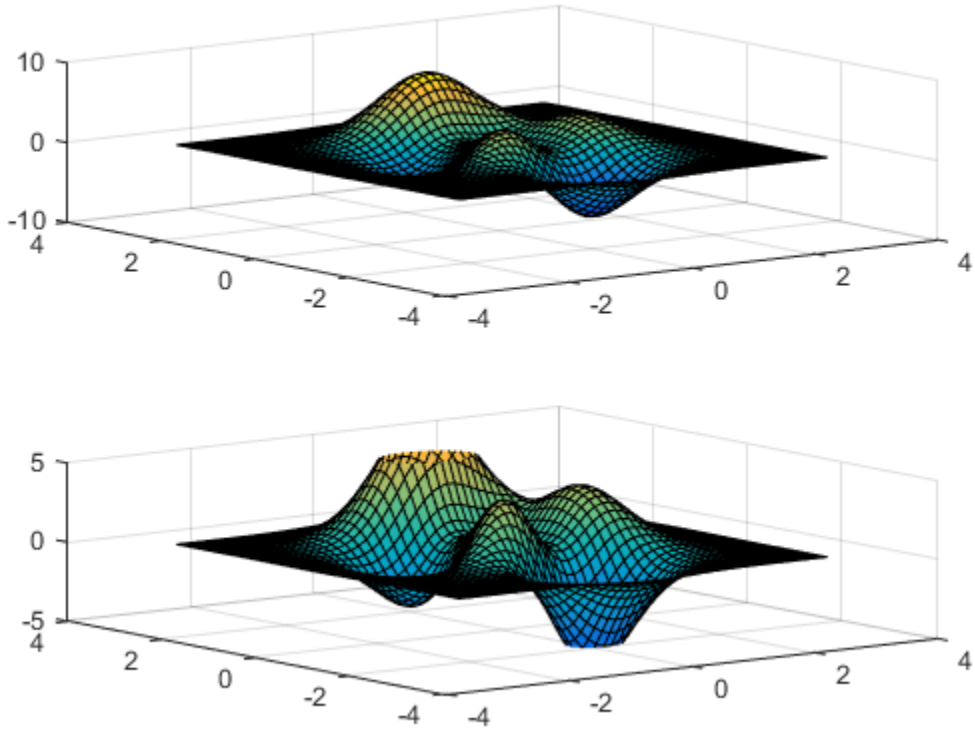
```
[X,Y,Z] = peaks;
surf(X,Y,Z);
zlim([-5 5])
```



Set *z*-Axis Limits for Specific Axes

Create a figure with two subplots and plot the same data in each subplot. Set the *z*-axis limits for the bottom subplot.

```
[X,Y,Z] = peaks;
ax1 = subplot(2,1,1);
surf(X,Y,Z)

ax2 = subplot(2,1,2);
surf(X,Y,Z)
zlim(ax2,[-5 5])
```

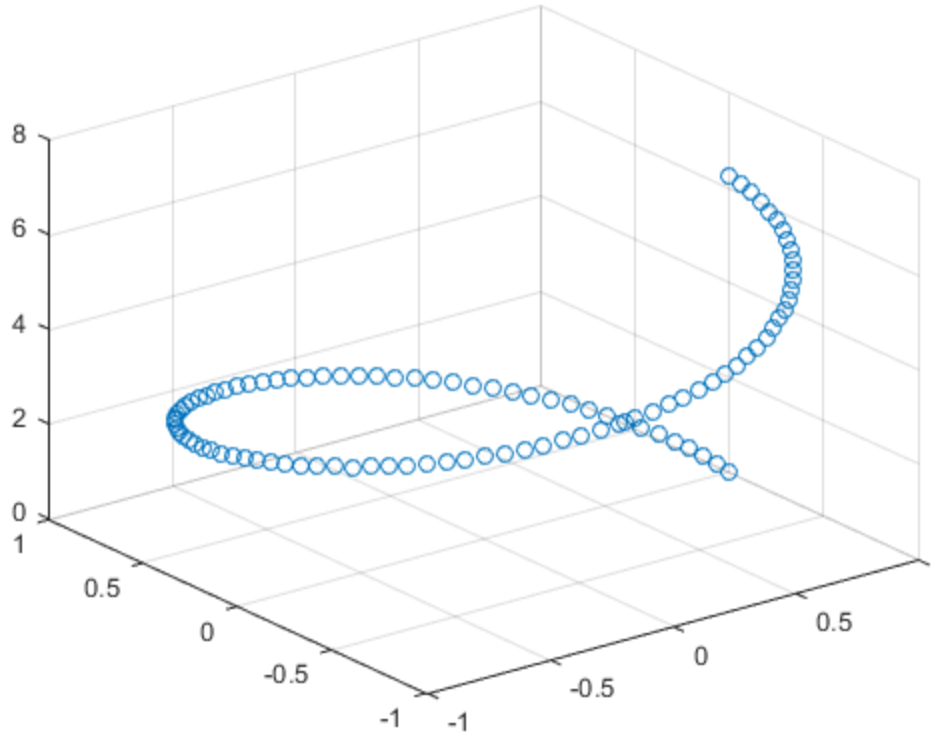


Maintain Current *z*-Axis Limits

Use manual mode to maintain the current z-axis limits when you add more plots to the axes.

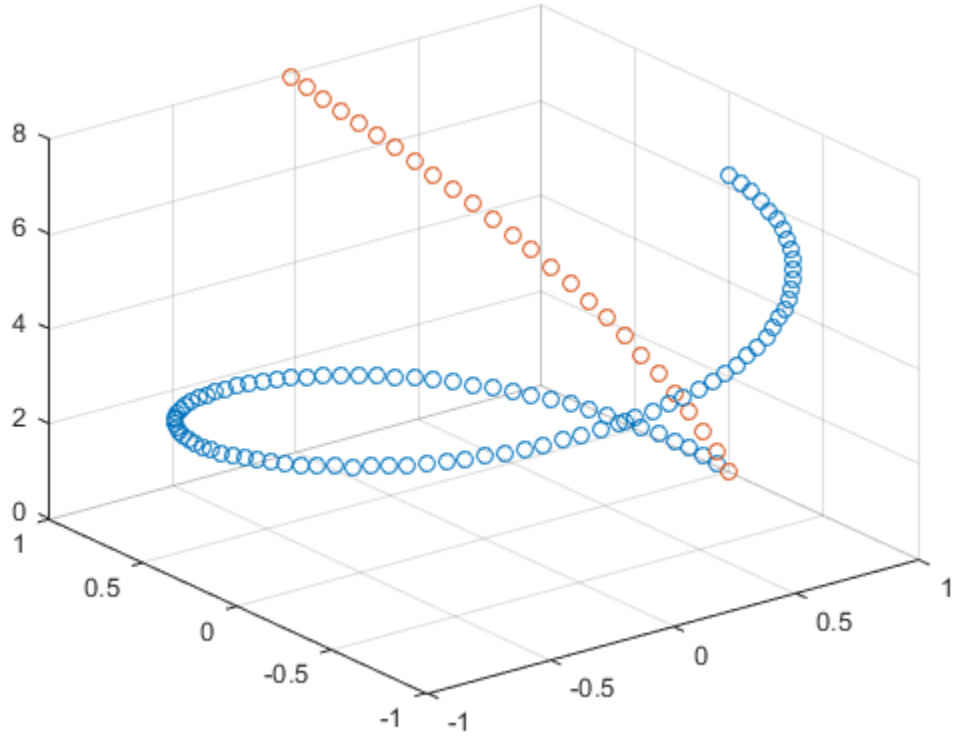
First, create a 3-D scatter plot.

```
theta = linspace(0,2*pi);
X = cos(theta);
Y = sin(theta);
Z = theta;
scatter3(X,Y,Z)
```



Set the *z*-axis limits mode to manual so that the limits to not change. Use `hold on` to add a second plot to the axes.

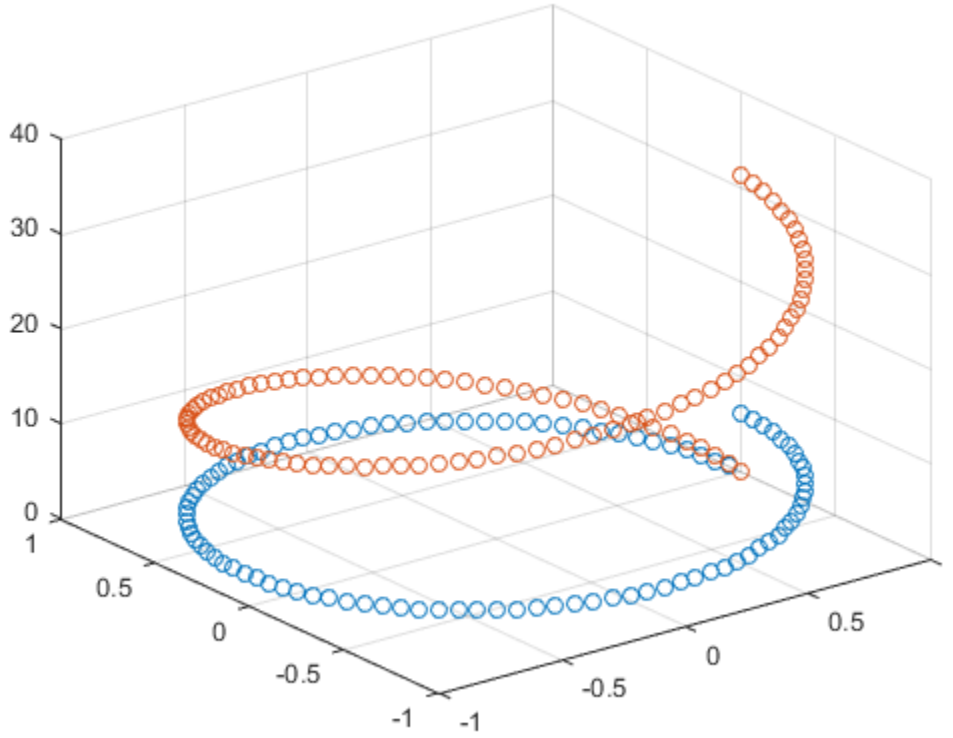
```
zlim manual
hold on
Znew = 5*theta;
scatter3(X,Y,Znew)
hold off
```



The *z*-axis limits do not update to incorporate the new plot.

Switch back to automatically updated limits by setting the mode to automatic.

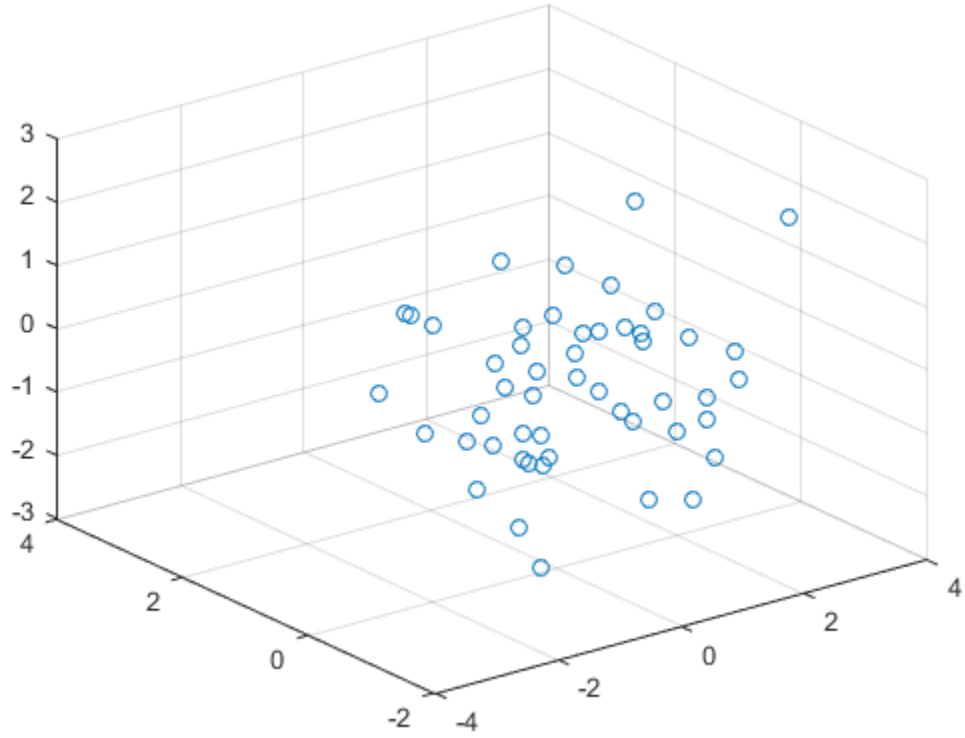
```
zlim auto
```



Return *z*-Axis Limits

Create a 3-D scatter plot of random data. Return the values of the *z*-axis limits.

```
x = randn(50,1);
y = randn(50,1);
z = randn(50,1);
scatter3(x,y,z)
```




```
zl = zlim
```

```
zl =

    -3     3
```

Input Arguments

`limits` — Minimum and maximum limits

two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[zmin zmax]`. Specifying the *z*-axis limits sets the *z*-axis limits mode to manual. The `ZLim` and `ZLimMode` properties for the corresponding axes store the *z*-axis limits and the limits mode, respectively.

Changing the *z*-axis limits can cause other limits to change, unless their corresponding mode properties are set to manual.

Example: `[0 1]`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

`ax` — Axes object

axes object

Axes object. If you do not specify an axes, then `zlim` sets the limits for the current axes (`gca`).

Output Arguments

`zl` — Current limits

two-element vector

Current limits, returned as a two-element vector of the form `[zmin zmax]`.

Querying the limits returns the `ZLim` property value for the corresponding axes.

`m` — Current limits mode

`'auto' | 'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — The limits update to reflect changes in the data.
- `'manual'` — The limits do not update to reflect changes in the data.

Querying the *z*-axis limits mode returns `ZLimMode` property value for the corresponding axes.

See Also

`axis | grid | hold | xlim | ylim | zlabel`

Introduced before R2006a

xlsfinfo

Determine if file contains Microsoft Excel spreadsheet

Syntax

```
status = xlsfinfo(filename)
[status,sheets] = xlsfinfo(filename)
[status,sheets,xlFormat] = xlsfinfo(filename)
```

Description

`status = xlsfinfo(filename)` indicates if `filename` is a file that the `xlsread` function can read.

`[status,sheets] = xlsfinfo(filename)` additionally returns the name of each spreadsheet in the file.

`[status,sheets,xlFormat] = xlsfinfo(filename)` also returns the format description that Excel returns for the file. On systems without Excel for Windows, `xlFormat` is an empty string, `''`.

Examples

View Information About Spreadsheet File

Create a sample Excel file named `myExample.xlsx`.

```
values = {1, 2, 3 ; 4, 5, 'x' ; 7, 8, 9};
headers = {'First', 'Second', 'Third'};
xlswrite('myExample.xlsx', [headers; values]);
```

Call `xlsfinfo` to get information about the file.

```
[status,sheets,xlFormat] = xlsfinfo('myExample.xlsx')
```

```
status =
```

```
Microsoft Excel Spreadsheet


sheets =

    'Sheet1'    'Sheet2'    'Sheet3'


xlFormat =

xlOpenXMLWorkbook
```

`status` is a descriptive string indicating that the `xlsread` function can read the sample file.

Input Arguments

`filename` — Name of file

string

Name of file, specified as a string.

Example: `'myFile.xlsx'`

Data Types: `char`

Output Arguments

`status` — Type of file

string

Type of file, returned as a string.

- If `filename` is a file that `xlsread` can read, then `status` is a descriptive string, such as `'Microsoft Excel Spreadsheet'`.
- If `filename` is not a file that `xlsread` can read, then `status` is an empty string, `''`.
- If MATLAB cannot find the file, then `xlsfinfo` returns an error.

`sheets` — Worksheet names

1-by-`n` cell array of strings

Worksheet names, returned as a 1-by-`n` cell array of strings, where `n` is the number of worksheets in the file. Each cell contains the name of a worksheet. If `xlsread` cannot read a particular worksheet, the corresponding cell contains an error message.

If `xlsfinfo` cannot read the file, then `sheets` contains an error message.

`xlFormat` — File format description returned by Excel

string

File format description returned by Excel, returned as a string.

On Windows systems with Excel software, `xlFormat` is a nonempty string, such as one of the following.

`'xlOpenXMLWorkbook'`	Spreadsheet in XLSX format (Excel 2007 or later)
`'xlWorkbookNormal'` or `'xlExcel8'`	Spreadsheet in XLS format (compatible with Excel 97-2003)
`'xlCSV'`	File in comma-separated value (CSV) format
`'xlHtml'` or `'xlWebArchive'`	Spreadsheet exported to HTML format

On all other systems, `xlFormat` is an empty string, `''`.

Limitations

- `xlsfinfo` supports only 7-bit ASCII characters.

More About

Tips

- If `xlsfinfo` warns that it cannot start an ActiveX server, then the COM server, which is part of the typical Excel installation, is unavailable. In this case, consider reinstalling your Excel software. On systems with Excel for Windows, `xlsfinfo` uses the COM server to obtain information.

See Also

`xlsread` | `xlswrite`

Introduced before R2006a

xlsread

Read Microsoft Excel spreadsheet file

Syntax

```
num = xlsread(filename)
num = xlsread(filename,sheet)
num = xlsread(filename,xlRange)
num = xlsread(filename,sheet,xlRange)
num = xlsread(filename,sheet,xlRange,'basic')
[num,txt,raw] = xlsread(___)

___ = xlsread(filename,-1)

[num,txt,raw,custom] = xlsread(filename,sheet,xlRange,'',processFcn)
```

Description

`num = xlsread(filename)` reads the first worksheet in the Microsoft Excel spreadsheet workbook named filename and returns the numeric data in a matrix.

`num = xlsread(filename,sheet)` reads the specified worksheet.

`num = xlsread(filename,xlRange)` reads from the specified range of the first worksheet in the workbook. Use Excel range syntax, such as `'A1:C3'`.

`num = xlsread(filename,sheet,xlRange)` reads from the specified worksheet and range.

`num = xlsread(filename,sheet,xlRange,'basic')` reads data from the spreadsheet in `basic` import mode. If your computer does not have Excel for Windows, `xlsread` automatically operates in `basic` import mode, which supports XLS, XLSX, XLSM, XLTX, and XLTM files.

If you do not specify all the arguments, use empty strings as placeholders, for example, `num = xlsread(filename,'','','basic')`.

`[num,txt,raw] = xlsread( ___ )` additionally returns the text fields in cell array txt, and both numeric and text data in cell array raw, using any of the input arguments in the previous syntaxes.

`___ = xlsread(filename,-1)` opens an Excel window to interactively select data. Select the worksheet, drag and drop the mouse over the range you want, and click **OK**. This syntax is supported only on Windows computers with Microsoft Excel software installed.

`[num,txt,raw,custom] = xlsread(filename,sheet,xlRange,'',processFcn)`, where `processFcn` is a function handle, reads from the spreadsheet, calls `processFcn` on the data, and returns the final results as numeric data in array `num`. The `xlsread` function returns the text fields in cell array `txt`, both the numeric and text data in cell array `raw`, and the second output from `processFcn` in array `custom`. The `xlsread` function does not change the data stored in the spreadsheet. This syntax is supported only on Windows computers with Excel software.

Examples

Read Worksheet Into Numeric Matrix

Create an Excel file named `myExample.xlsx`.

```
values = {1, 2, 3 ; 4, 5, 'x' ; 7, 8, 9};
headers = {'First','Second','Third'};
xlswrite('myExample.xlsx',[headers; values]);
```

`Sheet1` of `myExample.xlsx` contains:

First	Second	Third
1	2	3
4	5	x
7	8	9

Read numeric data from the first worksheet.

```
filename = 'myExample.xlsx';
A = xlsread(filename)
```

```
A =
     1     2     3
     4     5   NaN
```

```
     7     8     9
```

Read Range of Cells

Read a specific range of data from the Excel file in the previous example.

```
filename = 'myExample.xlsx';
sheet = 1;
xlRange = 'B2:C3';

subsetA = xlsread(filename,sheet,xlRange)
```

```
subsetA =
     2     3
     5   NaN
```

Read Column

Read the second column from the Excel file in the first example.

```
filename = 'myExample.xlsx';

columnB = xlsread(filename,'B:B')
```

```
columnB =
     2
     5
     8
```

For better performance, include the row numbers in the range, such as `'B1:B3'`.

Request Numeric, Text, and Raw Data

Request the numeric data, text data, and combined data from the Excel file in the first example.

```
[num,txt,raw] = xlsread('myExample.xlsx')
```

```
num =
     1     2     3
     4     5   NaN
     7     8     9

txt =
    'First'    'Second'    'Third'
    ''         ''          ''
```

```
    ''          ''          'x'

raw =
    'First'     'Second'    'Third'
    [     1]    [     2]    [     3]
    [     4]    [     5]    'x'
    [     7]    [     8]    [     9]
```

Execute a Function on a Worksheet

In the Editor, create a function to process data from a worksheet. In this case, set values outside the range `[0.2,0.8]` to `0.2` or `0.8`.

```
function [Data] = setMinMax(Data)

minval = 0.2;
maxval = 0.8;

for k = 1:Data.Count
  v = Data.Value{k};
  if v > maxval
    Data.Value{k} = maxval;
  elseif v < minval
    Data.Value{k} = minval;
  end
end
```

In the Command Window, add random data to `myExample.xlsx`.

```
A = rand(5);
xlswrite('myExample.xlsx',A,'MyData')
```

The worksheet named `MyData` contains values ranging from 0 to 1.

Read the data from the worksheet, and reset any values outside the range `[0.2,0.8]`. Specify the sheet name, but use `''` as placeholders for the xlRange and 'basic' inputs.

```
trim = xlsread('myExample.xlsx','MyData','','',@setMinMax);
```

Request Custom Output

Execute a function on a worksheet and display the custom index output.

In the Editor, modify the function `setMinMax` from the previous example to return the indices of the changed elements (custom output).

```
function [Data,indices] = setMinMax(Data)

minval = 0.2;
maxval = 0.8;
indices = [];

for k = 1:Data.Count
  v = Data.Value{k};
  if v > maxval
    Data.Value{k} = maxval;
    indices = [indices k];
  elseif v < minval
    Data.Value{k} = minval;
    indices = [indices k];
  end
end
```

Read the data from the worksheet `MyData`, and request the custom index output, `idx`.

```
[trim,txt,raw,idx] = xlsread('myExample.xlsx',...
    'MyData','','',@setMinMax);
```

Input Arguments

`filename` — File name

string

File name, specified as a string. If you do not include an extension, `xlsread` searches for a file with the specified name and a supported Excel extension. `xlsread` can read data saved in files that are currently open in Excel for Windows.

Example: `'myFile.xlsx'`

Data Types: `char`

`sheet` — Worksheet

string | positive integer

Worksheet, specified as one of the following:

- String that contains the worksheet name. The string cannot contain a colon (`:`). To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`. For XLS files in `basic` mode, `sheet` is case sensitive.

- Positive integer that indicates the worksheet index. This option is not supported for XLS files in `basic` mode.

`xlRange` — Rectangular range

string

Rectangular range, specified as a string.

Specify `xlRange` using two opposing corners that define the region to read. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `xlRange` input is not case sensitive, and uses Excel A1 reference style (see Excel help).

Range selection is not supported when reading XLS files in `basic` mode. In this case, use `''` in place of `xlRange`.

If you do not specify sheet, then `xlRange` must include both corners and a colon character, even for a single cell (such as `'D2:D2'`). Otherwise, `xlsread` interprets the input as a worksheet name (such as `'sales'` or `'D2'`).

If you specify `sheet`, then `xlRange`:

- Does not need to include a colon and opposite corner to describe a single cell.
- Can refer to a named range that you defined in the Excel file (see the Excel help).

When the specified `xlRange` overlaps merged cells:

- On Windows computers with Excel, `xlsread` expands the range to include all merged cells.
- On computers without Excel for Windows, `xlsread` returns data for the specified range only, with empty or `NaN` values for merged cells.

Data Types: `char`

`'basic'` — Flag to request reading in `basic` mode

literal string

Flag to request reading in `basic` mode, specified as the literal string, `'basic'`.

`basic` mode is the default for computers without Excel for Windows. In `basic` mode, `xlsread`:

- Reads XLS, XLSX, XLSM, XLTX, and XLTM files only.
- Does not support an `xlRange` input when reading XLS files. In this case, use `''` in place of `xlRange`.
- Does not support function handle inputs.
- Imports all dates as Excel serial date numbers. Excel serial date numbers use a different reference date than MATLAB date numbers.

`processFcn` — Handle to a custom function

function handle

Handle to a custom function. This argument is supported only on Windows computers with Excel software. `xlsread` reads from the spreadsheet, executes your function on a copy of the data, and returns the final results. `xlsread` does not change the data stored in the spreadsheet.

When `xlsread` calls the custom function, it passes a range interface from the Excel application to provide access to the data. The custom function must include this interface both as an input and output argument. (See “Execute a Function on a Worksheet” on page 1-9105)

Example: `@myFunction`

Output Arguments

`num` — Numeric data

matrix

Numeric data, returned as a matrix of `double` values. The array does not contain any information from header lines, or from outer rows or columns that contain nonnumeric data. Text data in inner spreadsheet rows and columns appear as `NaN` in the `num` output.

`txt` — Text data

cell array

Text data, returned as a cell array. Numeric values in inner spreadsheet rows and columns appear as empty strings, `''`, in `txt`.

For XLS files in `basic` import mode, the `txt` output contains empty strings, `''`, in place of leading columns of numeric data that precede text data in the spreadsheet. In all other cases, `txt` does not contain these additional columns.

Undefined values (such as `'#N/A'`) appear in the `txt` output as `'#N/A'`, except for XLS files in `basic` mode.

`raw` — Numeric and text data
cell array

Numeric and text data from the worksheet, returned as a cell array.

On computers with Excel for Windows, undefined values (such as `'#N/A'`) appear in the `raw` output as `'ActiveX VT_ERROR:'`. For XLSX, XLSM, XLTX, and XLTM files on other computers, undefined values appear as `'#N/A'`.

`custom` — Second output of the function corresponding to `processFcn`
defined by the function

Second output of the function corresponding to `processFcn`. The value and data type of `custom` are determined by the function.

Limitations

- `xlsread` reads only 7-bit ASCII characters.
- `xlsread` does not support non-contiguous ranges.

More About

Algorithms

- `xlsread` imports formatted dates as strings (such as `'10/31/96'`), except in `basic` mode and on computers without Excel for Windows.
- "Import and Export Dates to Excel Files"

See Also

`function_handle` | `importdata` | `readtable` | `uiimport` | `xlsfinfo` | `xlswrite`

Introduced before R2006a

xlswrite

Write Microsoft Excel spreadsheet file

Syntax

```
xlswrite(filename,A)
xlswrite(filename,A,sheet)
xlswrite(filename,A,xlRange)
xlswrite(filename,A,sheet,xlRange)

status = xlswrite( ___ )
[status,message] = xlswrite( ___ )
```

Description

`xlswrite(filename,A)` writes matrix `A` to the first worksheet in the Microsoft Excel spreadsheet workbook `filename` starting at cell `A1`.

`xlswrite(filename,A,sheet)` writes to the specified worksheet.

`xlswrite(filename,A,xlRange)` writes to the rectangular region specified by `xlRange` in the first worksheet of the workbook. Use Excel range syntax, such as `'A1:C3'`.

`xlswrite(filename,A,sheet,xlRange)` writes to the specified worksheet and range.

`status = xlswrite( ___ )` returns the status of the write operation, using any of the input arguments in previous syntaxes. When the operation is successful, status is `1`. Otherwise, status is `0`.

`[status,message] = xlswrite( ___ )` additionally returns any warning or error message generated by the write operation in structure `message`.

Examples

Write Vector to Spreadsheet

Write a 7-element vector to an Excel file.

```
filename = 'testdata.xlsx';
A = [12.7 5.02 -98 63.9 0 -.2 56];
xlswrite(filename,A)
```

Write to Specific Sheet and Range in Spreadsheet

Write mixed text and numeric data to an Excel file starting at cell `E1` of `Sheet2`.

```
filename = 'testdata.xlsx';
A = {'Time','Temperature'; 12,98; 13,99; 14,97};
sheet = 2;
xlRange = 'E1';
xlswrite(filename,A,sheet,xlRange)
```

Input Arguments

`filename` — File name

string

File name, specified as a string.

If `filename` does not exist, `xlswrite` creates a file, determining the format based on the specified extension. To create a file compatible with Excel 97-2003 software, specify an extension of `.xls`. To create files in Excel 2007 formats, specify an extension of `.xlsx`, `.xlsb`, or `.xlsm`. If you do not specify an extension, `xlswrite` uses the default, `.xls`.

Example: `'myFile.xlsx'`

Example: `'C:\myFolder\myFile.xlsx'`

A — Input matrix

matrix

Input matrix, specified as a two-dimensional numeric or character array, or, if each cell contains a single element, a cell array.

If `A` is a cell array containing something other than a scalar numeric or a string, then `xlswrite` silently leaves the corresponding cell in the spreadsheet empty.

The maximum size of array `A` depends on the associated Excel version. For more information on Excel specifications and limits, see the Excel help.

Example: `[10,2,45;-32,478,50]`

Example: `{92.0,'Yes',45.9,'No'}`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `cell`

`sheet` — Worksheet name

string | positive integer

Worksheet name, specified as one of the following:

- String that contains the worksheet name. The string cannot contain a colon (`:`). To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`.
- Positive integer that indicates the worksheet index.

If `sheet` does not exist, `xlswrite` adds a new sheet at the end of the worksheet collection. If `sheet` is an index larger than the number of worksheets, `xlswrite` appends empty sheets until the number of worksheets in the workbook equals `sheet`. In either case, `xlswrite` generates a warning indicating that it has added a new worksheet.

`xlRange` — Rectangular range

string

Rectangular range, specified as a string.

Specify `xlRange` using two opposing corners that define the region to write. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `xlRange` input is not case sensitive, and uses Excel A1 reference style (see Excel help). `xlswrite` does not recognize named ranges.

- If you do not specify `sheet`, then `xlRange` must include both corners and a colon character, even for a single cell (such as `'D2:D2'`). Otherwise, `xlswrite` interprets the input as a worksheet name (such as `'D2'`).
- If you specify sheet, then `xlRange` can specify only the first cell (such as `'D2'`). `xlswrite` writes input array `A` beginning at this cell.
- If `xlRange` is larger than the size of input array `A`, Excel software fills the remainder of the region with `#N/A`. If `xlRange` is smaller than the size of `A`, then `xlswrite` writes only the subset that fits into `xlRange` to the file.

Output Arguments

`status` — Status of the write operation

`1 | 0`

Status of the write operation, returned as either `1` (`true`) or `0` (`false`). When the write operation is successful, `status` is `1`. Otherwise, `status` is `0`.

`message` — Error or warning generated during the write operation

structure array

Error or warning generated during the write operation, returned as a structure array containing two fields:

`message`	Text of the warning or error message, returned as a string.
`identifier`	Message identifier, returned as a string.

Limitations

- If your computer does not have Excel for Windows, or if the COM server (part of the typical installation of Excel) is unavailable, then the `xlswrite` function:
  - Writes array A to a text file in comma-separated value (CSV) format. `A` must be a numeric matrix.
  - Ignores the sheet and xlRange arguments.

More About

Tips

- If your computer has Microsoft Office 2003 software, but you want to create a file in an Excel 2007 format, install the Office 2007 Compatibility Pack.
- Excel and MATLAB can store dates as strings (such as `'10/31/96'`) or serial date numbers (such as `729329`). If your array includes serial date numbers, convert the dates to strings using `datestr` before calling `xlswrite`. Alternatively, see “Import and Export Dates to Excel Files”.

- To write data to Excel files with custom formats (such as fonts or colors), access the Windows COM server directly using `actxserver` rather than `xlswrite`. For example, this MathWorks Support Answer uses `actxserver` to establish a connection between MATLAB and Excel, writes data to a worksheet, and specifies the colors of the cells.

Algorithms

Excel converts `Inf` values to `65535`. MATLAB converts `NaN` values to empty cells.

See Also

`writetable` | `xlsfinfo` | `xlsread`

Introduced before R2006a

xmlread

Read XML document and return Document Object Model node

Syntax

```
DOMnode = xmlread(filename)
```

Description

`DOMnode = xmlread(filename)` reads the specified XML file and returns a Document Object Model node representing the document.

Input Arguments

`filename`

String enclosed in single quotation marks that specifies the name of the local file or URL.

Output Arguments

`DOMnode`

Document Object Model node, as defined by the World Wide Web consortium. For more information, see "What Is an XML Document Object Model (DOM)?".

Examples

The root element in an XML file sometimes includes an `xsi:noNamespaceSchemaLocation` attribute. The value of this attribute is the name of the preferred schema file. Call the `getAttribute` method to get this value:

```
xDoc = xmlread(fullfile(matlabroot,'toolbox',...
               'matlab','general','info.xml'));

xRoot = xDoc.getDocumentElement;
```

```
schema = char(xRoot.getAttribute('xsi:noNamespaceSchemaLocation'))
```

This code returns:

```
schema =
http://www.mathworks.com/namespace/info/v1/info.xsd
```

Create functions that parse data from an XML file into a MATLAB structure array with fields `Name`, `Attributes`, `Data`, and `Children`:

```
function theStruct = parseXML(filename)
% PARSEXML Convert XML file to a MATLAB structure.
try
   tree = xmlread(filename);
catch
   error('Failed to read XML file %s.',filename);
end

% Recurse over child nodes. This could run into problems
% with very deeply nested trees.
try
   theStruct = parseChildNodes(tree);
catch
   error('Unable to parse XML file %s.',filename);
end


% ----- Local function PARSECHILDNODES -----
function children = parseChildNodes(theNode)
% Recurse over node children.
children = [];
if theNode.hasChildNodes
   childNodes = theNode.getChildNodes;
   numChildNodes = childNodes.getLength;
   allocCell = cell(1, numChildNodes);

   children = struct(             ...
      'Name', allocCell, 'Attributes', allocCell,    ...
      'Data', allocCell, 'Children', allocCell);

    for count = 1:numChildNodes
        theChild = childNodes.item(count-1);
        children(count) = makeStructFromNode(theChild);
    end
end
```

```
% ----- Local function MAKESTRUCTFROMNODE -----
function nodeStruct = makeStructFromNode(theNode)
% Create structure of node info.

nodeStruct = struct(                        ...
   'Name', char(theNode.getNodeName),       ...
   'Attributes', parseAttributes(theNode),  ...
   'Data', '',                              ...
   'Children', parseChildNodes(theNode));

if any(strcmp(methods(theNode), 'getData'))
   nodeStruct.Data = char(theNode.getData);
else
   nodeStruct.Data = '';
end

% ----- Local function PARSEATTRIBUTES -----
function attributes = parseAttributes(theNode)
% Create attributes structure.

attributes = [];
if theNode.hasAttributes
   theAttributes = theNode.getAttributes;
   numAttributes = theAttributes.getLength;
   allocCell = cell(1, numAttributes);
   attributes = struct('Name', allocCell, 'Value', ...
                       allocCell);

   for count = 1:numAttributes
      attrib = theAttributes.item(count-1);
      attributes(count).Name = char(attrib.getName);
      attributes(count).Value = char(attrib.getValue);
   end
end
```

More About

Tips

The display for a properly parsed document is `[#document: null]`. For example,

```
xDoc = xmlread('info.xml')
```

returns

```
xDoc =
[#document: null]
```

- “What Is an XML Document Object Model (DOM)?”
- “Example — Finding Text in an XML File”
- DOM Package Summary (methods and properties for nodes)

See Also

`xmlwrite` | `xslt`

Introduced before R2006a

xmlwrite

Write XML Document Object Model node

Syntax

```
xmlwrite(filename,DOMnode)
str = xmlwrite(DOMnode)
```

Description

`xmlwrite(filename,DOMnode)` writes the Document Object Model (DOM) node `DOMnode` to the file `filename`.

`str = xmlwrite(DOMnode)` serializes the DOM node to a string.

Input Arguments

`filename`

String enclosed in single quotation marks that specifies the name of a local file or a URL.

`DOMnode`

Document Object Model node, as defined by the World Wide Web consortium. For more information, see "What Is an XML Document Object Model (DOM)?"

Output Arguments

`str`

String that contains the serialized DOM node as it appears in an XML file.

Examples

Create and view an XML document:

```
docNode = com.mathworks.xml.XMLUtils.createDocument...
    ('root_element')
docRootNode = docNode.getDocumentElement;
docRootNode.setAttribute('attr_name','attr_value');
for i=1:20
    thisElement = docNode.createElement('child_node');
    thisElement.appendChild...
        (docNode.createTextNode(sprintf('%i',i)));
    docRootNode.appendChild(thisElement);
end
docNode.appendChild(docNode.createComment('this is a comment'));

xmlFileName = [tempname,'.xml'];
xmlwrite(xmlFileName,docNode);
type(xmlFileName);
```

More About

- "What Is an XML Document Object Model (DOM)?"
- "Creating an XML File"
- DOM Package Summary (methods and properties for nodes)

See Also

`xmlread` | `xslt`

Introduced before R2006a

xor

Logical exclusive-OR

Syntax

```
C = xor(A, B)
```

Description

`C = xor(A, B)` performs an exclusive OR operation on the corresponding elements of arrays `A` and `B`. The resulting element `C(i,j,...)` is logical true (`1`) if `A(i,j,...)` or `B(i,j,...)`, but not both, is nonzero.

A	B	C
Zero	Zero	0
Zero	Nonzero	1
Nonzero	Zero	1
Nonzero	Nonzero	0

Examples

Given `A = [0 0 pi eps]` and `B = [0 -2.4 0 1]`, then

```
C = xor(A,B)
C =
   0   1   1   0
```

To see where either `A` or `B` has a nonzero element and the other matrix does not,

```
spy(xor(A,B))
```

More About

- "Truth Table for Logical Operations"

See Also

`all` | `and` | `any` | `find` | `Logical Operators: Short Circuit` | `not` | `or`

Introduced before R2006a

xslt

Transform XML document using XSLT engine

Syntax

```
result = xslt(source, style, dest)
[result,style] = xslt( ___ )
xslt( ___ ,'-web')
```

Description

`result = xslt(source, style, dest)` transforms an XML document using a stylesheet and returns the resulting document's URL. The function uses these inputs, the first of which is required:

- `source` is the filename or URL of the source XML file. `source` can also specify a DOM node.
- `style` is the filename or URL of an XSL stylesheet.
- `dest` is the filename or URL of the desired output document. If `dest` is absent or empty, the function uses a temporary filename. If `dest` is `'-tostring'`, the function returns the output document as a MATLAB string.

`[result,style] = xslt( ___ )` returns a processed stylesheet appropriate for passing to subsequent XSLT calls as `style`. This prevents costly repeated processing of the stylesheet.

`xslt( ___ ,'-web')` displays the resulting document in the Help Browser.

Examples

This example converts the file `info.xml` using the stylesheet `info.xsl`, writing the output to the file `info.html`. It launches the resulting HTML file in the MATLAB Web Browser.

```
xslt('info.xml', 'info.xsl', 'info.html', '-web')
```

More About

Tips

MATLAB uses the Saxon XSLT processor, version 6.5.5, which supports XSLT 1.0 expressions. For more information, see http://saxon.sourceforge.net/saxon6.5.5/

For additional information on writing XSL stylesheets, see the World Wide Web Consortium (W3C®) web site, http://www.w3.org/Style/XSL/.

See Also

`xmlread` | `xmlwrite`

Introduced before R2006a

zeros

Create array of all zeros

Syntax

```
X = zeros
X = zeros(n)
X = zeros(sz1,...,szN)
X = zeros(sz)

X = zeros(___,typename)
X = zeros(___,'like',p)
```

Description

`X = zeros` returns the scalar `0`.

`X = zeros(n)` returns an `n`-by-`n` matrix of zeros.

`X = zeros(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of zeros where `sz1,...,szN` indicate the size of each dimension. For example, `zeros(2,3)` returns a 2-by-3 matrix.

`X = zeros(sz)` returns an array of zeros where size vector `sz` defines `size(X)`. For example, `zeros([2 3])` returns a 2-by-3 matrix.

`X = zeros(___,typename)` returns an array of zeros of data type `typename`. For example, `zeros('int8')` returns a scalar, 8-bit integer `0`. You can use any of the input arguments in the previous syntaxes.

`X = zeros(___,'like',p)` returns an array of zeros like `p`; that is, of the same data type (class), sparsity, and complexity (real or complex) as `p`. You can specify `typename` or `'like'`, but not both.

Examples

Matrix of Zeros

Create a 4-by-4 matrix of zeros.

```
X = zeros(4)
```

```
X =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

3-D Array of Zeros

Create a 2-by-3-by-4 array of zeros.

```
X = zeros(2,3,4);
size(X)
```

```
ans =

     2     3     4
```

Clone Size from Existing Array

Create an array of zeros that is the same size as an existing array.

```
A = [1 4; 2 5; 3 6];
sz = size(A);
X = zeros(sz)
```

```
X =

     0     0
     0     0
     0     0
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = zeros(size(A));
```

Specify Data Type of Zeros

Create a 1-by-3 vector of zeros whose elements are 32-bit unsigned integers.

```
X = zeros(1,3,'uint32')
X =
           0           0           0
class(X)
ans =
uint32
```

Clone Complexity from Existing Array

Create a scalar `0` that is complex like an existing array instead of real valued.

First, create a complex vector.

```
p = [1+2i 3i];
```

Create a scalar `0` that is complex like `p`.

```
X = zeros('like',p)
X =
   0.0000 + 0.0000i
```

Clone Sparsity from Existing Array

Create a 10-by-10 sparse matrix.

```
p = sparse(10,10,pi);
```

Create a 2-by-3 matrix of zeros that is sparse like `p`.

```
X = zeros(2,3,'like',p)
X =
   All zero sparse: 2-by-3
```

Clone Size and Data Type from Existing Array

Create a 2-by-3 array of 8-bit unsigned integers.

```
p = uint8([1 3 5 ; 2 4 6]);
```

Create an array of zeros that is the same size and data type as `p`.

```
X = zeros(size(p),'like',p)
```

```
X =

    0    0    0
    0    0    0
```

```
class(X)
```

```
ans =

uint8
```

Clone Distributed Array

If you have Parallel Computing Toolbox, create a 1000-by-1000 distributed array of zeros with underlying data type `int8`. For the `distributed` data type, the `'like'` syntax clones the underlying data type in addition to the primary data type.

```
p = zeros(1000,'int8','distributed');
```

Create an array of zeros that is the same size, primary data type, and underlying data type as `p`.

```
X = zeros(size(p),'like',p);
```

```
class(X)
```

```
ans =

distributed
```

```
classUnderlying(X)
```

```
ans =

int8
```

Input Arguments

n — Size of square matrix

integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension (as separate arguments)

integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `zeros` ignores trailing dimensions with a size of `1`. For example, `zeros([3,1,1,1])` produces a 3-by-1 vector of zeros.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)

integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `zeros` ignores trailing dimensions with a size of `1`. For example, `zeros([3,1,1,1])` produces a 3-by-1 vector of zeros.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`typename` — Data type (class) to create

`'double'` (default) | `'single'` | `'int8'` | `'uint8'` | ...

Data type (class) to create, specified as the string `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, `'uint64'`, or the name of another class that provides `zeros` support.

p — Prototype of array to create

numeric array

Prototype of array to create, specified as a numeric array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

More About

- "Class Support for Array-Creation Functions"
- " Preallocating Arrays"

See Also

`complex` | `eye` | `false` | `ones` | `rand` | `size`

Introduced before R2006a

zip

Compress files into zip file

Syntax

```
zip(zipfile,files)
zip(zipfile,files,rootfolder)
entrynames = zip(zipfile,files,rootfolder)
```

Description

`zip(zipfile,files)` creates a zip file with the name `zipfile` from the list of files and folders specified in `files`. Folders recursively include all of their content.

`zip(zipfile,files,rootfolder)` specifies the path for `files` relative to `rootfolder` instead of the current folder.

`entrynames = zip(zipfile,files,rootfolder)` returns a string cell array of the names of the files contained in `zipfile`. Specifying `rootfolder` is optional.

Input Arguments

`zipfile`

String that specifies the name of the zip file. If `zipfile` has no extension, MATLAB appends the `.zip` extension.

If `files` includes relative paths, the zip file also contains relative paths. The zip file does not include absolute paths.

`files`

String or cell array of strings containing the list of files or folders to include in `zipfile`.

Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current folder or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~username/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`rootfolder`

String that specifies the root of the paths for the files to zip.

Relative paths in the zip file reflect the relative paths in `files`, and do not include path information from `rootfolder`.

Default: current folder (`'.'`)

Output Arguments

`entrynames`

Cell array of strings that contain the paths to the files in `zipfile`. If files includes relative paths, `entrynames` also contains relative paths.

Examples

Zip a File

Create a zip file of the file `membrane.m`, which is in the MATLAB `general` folder. Save the zip file `tmwlogo.zip` in the current folder.

```
file = fullfile(matlabroot,'toolbox','matlab','general','membrane.m');
zip('tmwlogo',file);
```

Zip Selected Files

Suppose that your system has a folder named `d:/myfiles`. Zip the files `membrane.m` and `logo.m`, which are on the MATLAB search path, into a file named `tmwlogo.zip`.

```
myfile = fullfile('d:','myfiles','tmwlogo.zip');
zip(myfile,{'membrane.m','logo.m'});
```

Zip all `.m` and `.mat` files in the current folder to the file `backup.zip`.

```
zip('backup',{'*.m','*.mat'});
```

Zip a Folder

Suppose that your current folder contains a subfolder named `mywork`. Zip the contents of all subfolders of `mywork`, and store the relative paths in the zip file.

```
zip('myfiles.zip','mywork');
```

Zip Between Folders

Suppose that you have files `thesis.doc` and `defense.ppt` in `d:/PhD`. Zip these files into `thesis.zip`, one level up from the current folder.

```
zip('../thesis.zip',{'thesis.doc','defense.ppt'},'d:/PhD');
```

Create Zip Archive of Web Page

Create a zip archive of a Web page.

Locate the list of files at the MATLAB Central File Exchange uploaded within the past 7 days, that contain "Simulink."

```
filex = 'http://www.mathworks.com/matlabcentral/fileexchange/';
params = {'duration','7','term','simulink'};
```

Save the Web content to a file.

```
urlwrite(filex,'contains_simulink.html','get',params);
```

Create a zip archive of the retrieved Web page, using the `zip` function.

```
zip('simulink_matches.zip','contains_simulink.html');
```

`zip` creates a zip archive named `simulink_matches.zip` that contains the file, `contains_simulink.html`.

Alternatives

To zip files in the Current Folder browser, select the files, right-click to open the context menu, and then select **Create Zip File**.

See Also

`gzip` | `gunzip` | `tar` | `untar` | `unzip`

Introduced before R2006a

zoom

Turn zooming on or off or magnify by factor

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig, option)
h = zoom(figure_handle)
```

Description

`zoom on` turns on interactive zooming. When interactive zooming is enabled in a figure, pressing a mouse button while your cursor is within an axes zooms into the point or out from the point beneath the mouse. Zooming changes the axes limits. When using zoom mode, you

- Zoom in by positioning the mouse cursor where you want the center of the plot to be and either
  - Press the mouse button or
  - Rotate the mouse scroll wheel away from you (upward).
- Zoom out by positioning the mouse cursor where you want the center of the plot to be and either
  - Simultaneously press **Shift** and the mouse button, or
  - Rotate the mouse scroll wheel toward you (downward).

Each mouse click or scroll wheel click zooms in or out by a factor of 2.

Clicking and dragging over an axes when zooming in is enabled draws a rubberband box. When you release the mouse button, the axes zoom in to the region enclosed by the rubberband box.

Double-clicking over an axes returns the axes to its initial zoom setting in both zoom-in and zoom-out modes.

`zoom off` turns interactive zooming off.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive `zoom` mode is enabled, will return to this zoom level.

`zoom` toggles the interactive zoom status between off and on (restoring the most recently used zoom tool).

`zoom xon` and `zoom yon` set `zoom on` for the *x*- and *y*-axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by `1/factor`.

`zoom(fig, option)` Any of the preceding options can be specified on a figure other than the current figure using this syntax.

`h = zoom(figure_handle)` returns a zoom *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Zoom Mode Objects

You can access the following properties.

- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure
- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set
- *`Motion`* `'horizontal'|'vertical'|'both'` — The type of zooming enabled for the figure

- *`Direction`* `'in'|'out'` — The direction of the zoom operation
- *`RightClickAction`* `'InverseZoom'|'PostContextMenu'` — The behavior of a right-click action

  A value of `'InverseZoom'` causes a right-click to zoom out. A value of `'PostContextMenu'` displays a context menu. This setting persists between MATLAB sessions.
- `UIContextMenu <handle>` — Specifies a custom context menu to be displayed during a right-click action

  This property is ignored if the `RightClickAction` property has been set to `'on'`.

Zoom Mode Callbacks

You can program the following callbacks for zoom mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  The application can inhibit the zoom operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

```
function [res] = myfunction(obj,event_obj)
% obj          handle to the object clicked on
% event_obj    struct for event data (empty in this release)
% res [output] a logical flag determines whether the zoom
%              operation should take place(for 'res' set
%              to 'false' or the 'ButtonDownFcn' property
%              of the object should take precedence (when
%              'res' is 'true')
```

- `ActionPreCallback <function_handle>` — Function to execute before zooming

  Set this callback if you want to execute code when a zoom operation starts. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj          handle to the figure clicked on
% event_obj    object containing struct of event data
```

The event data has the following field.

`Axes`	The handle of the axes that is being zoomed

- `ActionPostCallback <function_handle>` — Function to execute after zooming

  Set this callback if you want to execute code when a zoom operation finishes. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj         handle to the figure clicked on
% event_obj   object containing struct of event data
%             (same as the event data of the
%             'ActionPreCallback' callback)
```

Zoom Mode Utility Functions

The following functions in zoom mode query and set certain of its properties.

- `flags = isAllowAxesZoom(h,axes)` — Function querying permission to zoom axes

  Calling the function `isAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a zoom operation is permitted on the axes objects.

- `setAllowAxesZoom(h,axes,flag)` — Function to set permission to zoom axes

  Calling the function `setAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a zoom operation on the axes objects.

- *`info`* `= getAxesZoomMotion(h,axes)` — Function to get style of zoom operations

  Calling the function `getAxesZoomMotion` on the zoom object, `H`, with a vector of axes handles, `axes`, as input returns a character cell array of the same dimension as the axes handle vector, which indicates the type of zoom operation for each axes. Possible values for the type of operation are `'horizontal'`, `'vertical'`, or `'both'`.

- `setAxesZoomMotion(h,axes,style)` — Function to set style of zoom operations

Calling the function `setAxesZoomMotion` on the zoom object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, sets the style of zooming on each axes.

Examples

Example 1 — Entering Zoom Mode

Plot a graph and turn on Zoom mode:

```
plot(1:10);
zoom on
% zoom in on the plot
```

Example 2 — Constrained Zoom

Create zoom mode object and constrain to *x*-axis zooming:

```
plot(1:10);
h = zoom;
h.Motion = 'horizontal';
h.Enable = 'on';
% zoom in on the plot in the horizontal direction.
```

Example 3 — Constrained Zoom in Subplots

Create four axes as subplots and set zoom style differently for each by setting a different property for each axes handle:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = zoom;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesZoom(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesZoomMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesZoomMotion(h,ax4,'vertical');
```

```
% Zoom in on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for zoom mode objects to trigger. Copy the following code to a new file, execute it, and observe zooming behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
hLine.ButtonDownFnc = 'disp(''This executes'')';
hLine.Tag = 'DoNotIgnore';
h = zoom;
h.ButtonDownFilter = @mycallback;
h.Enable = 'on';
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = obj.Tag;
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-buttonDown events for zoom mode objects to trigger. Copy the following code to a new file, execute it, and observe zoom behavior:

```
function demo
% Listen to zoom events
plot(1:10);
h = zoom;
h.ActionPreCallback = @myprecallback;
h.ActionPostCallback = @mypostcallback;
h.Enable = 'on';
%
function myprecallback(obj,evd)
disp('A zoom is about to occur.');
%
function mypostcallback(obj,evd)
```

```
newLim = evd.Axes.XLim;
msgbox(sprintf('The new X-Limits are [%.2f %.2f].',newLim));
```

Example 6 — Creating a Context Menu for Zoom Mode

Coding a context menu that lets the user to switch to Pan mode by right-clicking:

```
figure
plot(magic(10))
hCMZ = uicontextmenu;
hZMenu = uimenu('Parent',hCMZ,'Label','Switch to pan',...
'Callback','pan(gcbf,''on'')');
hZoom = zoom(gcf);
hZoom.UIContextMenu = hCMZ;
zoom('on')
```

You cannot add items to the built-in zoom context menu, but you can replace it with your own.

Alternatives

Use the Zoom tools  on the figure toolbar to zoom in or zoom out on a plot, or select **Zoom In** or **Zoom Out** from the figure's **Tools** menu. For details, see “Zooming in Graphs”.

More About

Tips

`zoom` changes the axes limits by a factor of 2 (in or out) each time you press the mouse button while the cursor is within an axes. You can also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

You can create a zoom mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

Note: Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change

any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a UI that updates a figure's callbacks, the UI should somehow keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different zoom behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

See Also

`linkaxes` | `pan` | `rotate3d`

Introduced before R2006a